JOURNAL OF COMPUTER AND SYSTEM SCIENCES **38**, 360–379 (1989)

Deleting Completed Transactions

THANASIS HADZILACOS

Computer Technology Institute, Patras, Greece

AND

MIHALIS YANNAKAKIS

AT & Bell Laboratories, Murray Hill, New Jersey

Received February 19, 1988

We derive necessary and sufficient conditions on when it is safe to forget (and remove) a completed transaction in several versions of conflict-graph-based schedulers. We show that the conditions can be applied repeatedly and analyze their complexity. © 1989 Academic Press, Inc.

1. INTRODUCTION

An important point in the life of a transaction, besides commit time, is when its existence no longer influences the correct scheduling of present or future transactions, and information about it (its name, the items it read or wrote) can be forgotten. [SR] calls this *closing* the transaction. Of course, when a transaction can be closed depends on the particular algorithm used to control concurrency.

One of the most widely used criteria for ensuring correctness of interleaved transactions is *conflict serializability*. From a schedule of transactions a graph is formed (the *conflict graph* of the schedule, sometimes called also serialization or dependency graph), whose nodes correspond to the transactions and whose arcs reflect the order in which the transactions executed conflicting steps. Acyclicity of the graph guarantees the correctness of the schedule [EGLT]. Locking protocols provide a simple and efficient way for ensuring correctness but capture only a subset of the conflict serializable schedules. A way to implement (all) conflict serializable schedules is to keep track of the conflict graph and output a step only if it does not produce a cycle in the graph. The use of portions or variants of the conflict graph has been also proposed in conjunction with multiple versions, locking, and timestamps [BOG, BHR, HP, SR].

If pure locking is used to control concurrency (i.e., the scheduler just manages locks), then it is easy to see that transactions can be closed at commit time. First,

0022-0000/89 $3.00

observe that once a transaction T completes and releases all its locks, it no longer influences the scheduling of future steps. That is, as far as serializability is concerned, the transaction does not play any role from that time on. However, we may still have to keep information about T for reliability purposes: if T has read values from some transactions that have not committed yet, then T has to wait for these transactions to commit before it can also commit. (The reason is that if one of these transactions aborts, then T must also abort because it has read invalid data). Once T commits, it can be closed; i.e., all information about it can be removed from the database. This observation is valid for all locking protocols, including the protocols that operate on structured databases, such as the ones in [SK, Y].

In contrast to the case of schedulers based on locking, it is well known that, for schedulers that use the conflict graph, it is not correct to close transactions at commit time. Even though a transaction has committed, its presence in the graph may be necessary to detect future cycles. On the other hand, of course, we cannot keep transactions indefinitely. The problem of characterizing when a transaction can be safely removed in conflict-graph based schedulers has been floating around for some time.

In this paper we study necessary and sufficient conditions for removing transactions in several versions of conflict graph schedulers. These versions differ depending (1) on the model of the transactions and (2) on the information available to the scheduler. In the basic version, (1) a transaction is a sequence of read steps followed by a final atomic write step, and (2) the scheduler does not know ahead of time what data the transactions will access. In Section 2 we describe the model and define in more detail the basic conflict graph scheduler.

In Section 3 we prove a necessary and sufficient condition under which a completed transaction can be removed from the conflict graph without affecting our ability to detect cycles in the future. Of course, in order for this to be useful, we must extend it so that multiple nodes can be removed from the graph, one by one as more steps come in and more transactions become irrelevant. Section 4 examines this problem. The issues involved are subtle and require a careful formalization of the problem. A counterintuitive phenomenon is that we may have two transactions each one of which can be removed, but such that they cannot be both removed simultaneously: removing one of them disables the criterion for the other. In general, when several transactions are eligible for removal we can determine in polynomial time whether a given subset of them can be (simultaneously) removed. However, we show that finding such a subset with the maximum number of transactions is NP-complete.

Section 5 examines two variants of the basic model. In the first variant, transactions may have multiple write steps, while in the second variant, transactions predeclare their steps. Surprisingly, we show that in the multiple write case, the problem of deciding whether a transaction can be removed from the graph is NP-complete. In the case of predeclared transactions we prove a necessary and sufficient condition which can be tested in polynomial time.

2. The Model

We employ the usual model of concurrency control [Pa]. A database is a set of *entities*. A transaction is a sequence of steps; each step reads some entity x (denoted rx for short) or writes x (wx for short). The value written by a transaction in a write step is an uninterpreted function of the values read so far. A *schedule* of a set τ of transactions is an execution of the transactions of τ in a (possibly) interleaved fashion. A schedule is *serial* if there is no interleaving. A schedule S is *serializable* if there is a serial schedule S' such that no difference is "visible" between S and S'; formally, for every initial state of the database and every interpretation of the functions computed by the transactions, each transaction reads the same values, and the final state of the database is the same in both schedules.

Serializability is an intricate concept (as manifested by its NP-completeness), so that a stronger but simpler criterion is usually employed. Two steps of two (different) transactions *conflict* if they involve the same entity and at least one of them is a write step. If S is a schedule of a set τ of transactions, the *conflict graph* CG(S) is a directed graph with set of nodes τ, and arcs $T_i \rightarrow T_j$ if a step of T_i precedes a conflicting step of T_j. The schedule is *conflict serializable* (CSR for short) if the conflict graph is acyclic. This is a sufficient condition for serializability.

We make the following two assumptions about the transactions:

(1) We will assume that all values written by a transaction are installed atomically at the end, i.e., that the execution of a transaction is a sequence of read steps followed by a final write step. The effect of this assumption is that no transaction reads dirty data, cascading aborts do not happen, and transactions may commit upon completion.

(2) We assume that the future of an *active* transaction (one that has not completed) is unknown; i.e., the scheduler does not know what entities it will read or write.

In Section 5 we relax these assumptions.

The full freedom of CSR can be achieved using either a certification (optimistic) or a preventive scheduling algorithm. In the first case, the conflict graph of the completed transactions is maintained. The active transactions are left free to run. When an active transaction is ready to terminate, a certification phase takes place, in which it is tested whether the transaction can be added to the conflict graph without creating cycles; if so, it is certified and completed, otherwise it aborts (and is restarted). In the second case, the conflict graph of the schedule seen so far of the completed and active transactions is maintained step-by-step. A new step of a transaction is accepted only if it does not create a cycle; otherwise, the transaction aborts. The issues are very similar in the two cases, so we will restrict ourselves to the second one.

In more detail, the conflict scheduler operates as follows. We assume that every transaction starts with a BEGIN step and completes with its final write step. The rules of the scheduler are as follows. Initially the conflict graph CG is empty. When the next step of a transaction arrives the graph is modified as follows, provided that no cycle is created; if a cycle is created the transaction aborts and is removed from the graph:

Rule 1. If the step is the BEGIN step of a new transaction T_i, a node T_i is added to the graph.

Rule 2. If the step is a read x step of transaction T_i an arc is added from every node (transaction) in the graph that has written x to T_i.

Rule 3. If the step is the write step of transaction T_i, then for every entity x that is written and for every node of the graph (transaction) T_j that has previously read or written x, an arc $T_j \rightarrow T_i$ is added to the graph.

The sequence s of steps that have arrived up to a certain time may contain steps of transactions which have in the meantime aborted and may not contain all the steps of some transactions (namely, the active transactions). Still, we will use the term "schedule" also for s. The *accepted* subschedule of s is its projection on the nonaborted transactions; i.e., the subsequence of s consisting of the steps of transactions that have not aborted. We denote the graph that has been constructed by the scheduler on input s by CG(s) and call it the conflict graph of s; in terms of our previous definition, CG(s) is the conflict graph of the accepted subschedule. According to Rules 1–3 of the scheduler, the current graph CG(s) is always acyclic; thus, if no more steps are to be executed, the accepted subschedule (of the completed and active transactions) is serializable.

In the following, for ease of notation, we will say that CG(r) is cyclic to mean that if the last step of r is executed, then a cycle will be created in the graph. Of course, the scheduler will abort the transaction that wants to execute the last step to maintain the acyclicity of the conflict graph.

3. Deleting a Single Transaction

Recall first some graph-theoretic definitions. A transaction T_i is a *predecessor* (resp. *immediate predecessor*) of transaction T_j in the graph if there is a path (resp. arc) from T_i to T_j. We say also that T_j is a *successor* (resp. *immediate successor*) of T_i.

From Rules 1–3 according to which the conflict graph is built, it follows that a new arc $T_i \rightarrow T_j$ is added to the graph because of some step of T_j; i.e., T_j is active when the arc is added. Therefore, once a transaction completes, it will never acquire any new immediate predecessors. It follows inductively that if a completed transaction T_i has no active predecessors, then its set of predecessors will remain the same forever. In particular, if T_i is not a predecessor of itself, it will never become one. Thus,

Lemma 1. *If a completed transaction T_i has no active predecessors, then T_i will not participate in any cycle in any future conflict graph.*

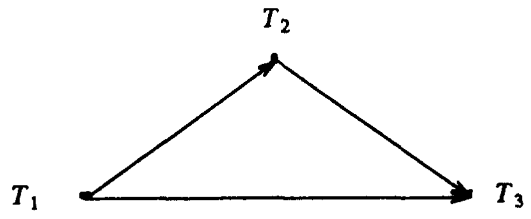

FIGURE 1

The converse to the lemma is also true. A *continuation* of a schedule p is a sequence of steps of the active transactions of p and possibly some new transactions. It is easy to show that for any schedule p with an (acyclic) conflict graph CG(p) and any completed transaction T_i of p which has some active predecessor, there is a continuation r such that CG(pr) contains a cycle passing through T_i. This does not mean, however, that T_i is essential to detecting the fact that CG(pr) is cyclic.

EXAMPLE 1. Suppose that p is the following schedule. Transaction T_1 first reads (among other things) entity x. Subsequently, before T_1 terminates, in a serial order T_2 and T_3 read and write x and complete. The conflict graph of p is shown in Fig. 1. Transaction T_2 has an active predecessor (namely, T_1). However, it should be intuitively clear that any cycle passing through T_2 which is formed in the future can equally well go through T_3; therefore, T_2 can be safely deleted.

We define now how a transaction is removed. Let p be a schedule and T_i a completed transaction of it. The *reduced conflict graph* of p by T_i, denoted RCG(p, T_i), is CG(p) with node T_i deleted and arcs to and from it replaced by arcs from all its immediate predecessors to all its immediate successors. The arcs from immediate predecessors to immediate successors of T_i are added so that paths going currently through T_i do not get lost. If the cycle-checking algorithm keeps track of the transitive closure of the graph (to facilitate testing whether a new arc can be inserted), then removing a transaction is equivalent to simply deleting the corresponding node and incident edges from the transitive closure.

Removal of a transaction T_i means that the scheduler replaces CG(p) by RCG(p, T_i) and continues applying the same rules 1–3 as usual. If r is a continuation of p, the graph resulting from RCG(p, T_i) after applying the rules for r is denoted by RCG(p, T_i, r). We want to delete a transaction only if correctness is not jeopardized, that is, only if the reduced scheduler (the scheduler with the reduced information) still accepts only conflict serializable schedules.

LEMMA 2. *Suppose that a (completed) transaction T_i is removed from the conflict graph after schedule p. The following conditions are equivalent:*

(1) *For all continuation r,* RCG(p, T_i, r) *acyclic implies* CG(pr) *acyclic; i.e., if*

a step is accepted by the reduced scheduler then it is also accepted by the original conflict scheduler.

(2) *For all continuations r,* RCG(p, T_i, r) *acyclic if and only if* CG(pr) *acyclic; i.e., the reduced and the conflict scheduler behave exactly the same way.*

(3) *For all continuations r, the subschedule of pr accepted by the reduced scheduler is conflict serializable.*

Proof. It is obvious that condition (2) implies both (1) and (3). We have to show that (3) implies (2) and (1) implies (2). Consider a shortest continuation r that violates condition (2). We shall argue that r violates also both (1) and (3). Let t be the last step of r, and let $r = st$. During s the two schedulers, the reduced and the original one, have behaved the same way. Therefore, the same transactions have been aborted after s. If RCG(p, T_i, r) contains an arc $T_k \to T_j$, then both nodes T_k, T_j are present in CG(ps). This arc was either generated from some step, in which case CG(ps) contains the same arc, or was generated in the process of removing T_i, in which case CG(ps) contains arcs $T_k \to T_i$, $T_i \to T_j$. Thus, if there is a path between two transactions in RCG(p, T_i, s) then there is such a path also in CG(ps). It follows that if step t creates a cycle in RCG then it creates a cycle also in CG. Since the two schedulers disagree at step t, we conclude that t creates a cycle in the conflict graph but not in the reduced graph. Thus, r violates condition (1). To see that r violates also (3), note that t is accepted by the reduced scheduler while CG(pr) is cyclic. If all active transactions complete now, the reduced scheduler has accepted a non-CSR schedule. ▌

We define now what it means for a condition to be necessary and sufficient. A condition C is *sufficient* for the removal of transaction T_i if C implies any one of the equivalent conditions of Lemma 2; it is *necessary* if the conditions of Lemma 2 imply C. To state our necessary and sufficient condition we need the following definition. Transaction T_i is a *tight predecessor* of T_j (and T_j is a *tight successor* of T_i) if there is a path from T_i to T_j that uses only completed transactions as intermediate nodes. We say also that a write access of an entity by a transaction is *stronger* than a read access. A necessary and sufficient condition is given in the following theorem.

THEOREM 1. *Let p be a schedule and T_i a completed transaction. The following condition is necessary and sufficient for the removal of T_i:*

(C1) *For all active tight predecessors T_j of T_i and for all entities x accessed by T_i there is a completed tight successor T_k $(\neq T_i)$ of T_j that accesses x at least as strongly as T_i.*

Proof. Sufficiency) Suppose that C1 is true, but condition (2) of Lemma 2 is false. Let $r = st$ be a shortest continuation for which the reduced and the conflict scheduler disagree. From the proof of Lemma 2, the last step t creates a cycle in the conflict graph but not in the reduced graph. If the cycle does not contain T_i, then the same cycle would be created in the reduced graph RCG(p, T_i, r). Therefore, the cycle contains T_i.

The transactions of the cycle can be partitioned according to their state right after p into (a) completed, (b) active, and (c) those that had not started yet. Not all transactions are of type (a) because of the acyclicity of CG(p). Starting from T_i walk backwards in the cycle until a transaction T_j not of type (a) is encountered for the first time. Clearly, the graph cannot have an arc from a transaction of type (c) to a transaction of type (a). Therefore, T_j was active right after p. All arcs in the path from T_j to T_i enter transactions which had completed in p. Therefore, they are all present in CG(p), and T_j was an active tight predecessor of T_i in CG(p).

Let T_l be the immediate successor of T_i on the cycle; see Fig. 2. If the arc $T_i \rightarrow T_l$ is present in CG(p), then the reduced graph RCG can use the arc from the immediate predecessor of T_i on the cycle to T_l to avoid T_i. Therefore, the arc $T_i \rightarrow T_l$ must have been added to the conflict graph at some step of the continuation r. At that step, transaction T_l performed an access of some entity x conflicting with some access of T_i. Let T_k be the tight successor of T_j for x guaranteed by C1. Since T_k accesses x at least as strongly as T_i in p, there is an arc from T_k to T_l. The reduced conflict graph can use the path from T_j to T_k (still present since all transactions are completed) and the arc $T_k \rightarrow T_l$ to form a cycle avoiding T_i.

(Necessity) Suppose that C1 does not hold: i.e., there is an active tight predecessor T_j of T_i and an entity x such that there is no tight completed successor of T_j accessing x at least as strongly as T_i. We shall find a continuation r such that RCG(p, T_i, r) is acyclic while CG(pr) contains a cycle.

The continuation r has the form $r = st$, where s is constructed so that it has the effect of aborting all active transactions except T_j. Let y be any entity other than x. The sequence s is as follows. First, all active transactions except T_j read y; then a new transaction T_m writes y, and finally all active transactions except T_j try to write y. Clearly, the last writes will fail and all active transactions except T_j will be aborted, both by the reduced and by the conflict scheduler.

The last step t of r is as follows. If T_i reads but does not write x then T_j writes x; if T_i writes x then T_j reads x. Thus, the last step t tries to introduce an arc $T_i \rightarrow T_j$ in the conflict graph, which of course forms a cycle. Let us see, however, what happens in the reduced graph. After s the only nodes are the transactions that

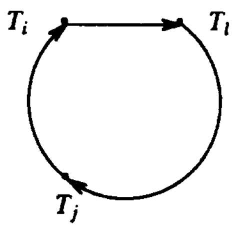

FIGURE 2

completed in p (except T_i), T_j and the new transaction T_m. The reduced graph RCG(p, T_i, s) has possibly some arcs directed into the new transaction T_m; all of its other arcs were present in the reduced graph RCG(p, T_i) before r. Therefore, the successors of T_j in RCG(p, T_i, s) are exactly the completed tight successors of T_j in CG(p) and possibly T_m. The last step t introduces arcs to T_j from all nodes T_k which have performed a conflicting step on x. Any such node accessed x at least as strongly as T_i, and therefore, is not a successor of T_j. It follows that no cycle is created in the reduced graph. ▌

Clearly, the condition of Theorem 1 can be tested in polynomial time. The following corollary gives another, easy to check, sufficient condition. The condition is incomparable to the one of Lemma 1. Say that a completed transaction is *current* if it has read or written the current value of some entity (i.e., the entity has not been subsequently overwritten). For example, transaction T_3 of Example 1 is current, but T_2 is not.

COROLLARY 1. *A noncurrent transaction can be removed.*

Proof. Let T_i be a noncurrent transaction and x an entity accessed by it. Let T_k be the last transaction to write x. Since T_i is not current, T_k wrote x after the access of T_i, and therefore the conflict graph has an arc $T_i \rightarrow T_k$. Also, since transactions write atomically at the end, T_k has completed. Consequently, every active tight predecessor of T_i has a completed tight successor, namely T_k, which has accessed x at least as strongly as T_i. ▌

4. REPEATEDLY DELETING TRANSACTIONS

In the last section we proved a necessary and sufficient condition for removing a transaction from the conflict graph. We would like conditions that allow us to do this reduction repeatedly as the schedule progresses and more and more transactions become irrelevant. Can we use the condition of Theorem 1 to repeatedly remove transactions? This is not immediately clear. One reason is that after removing a transaction, we do not deal any more with the conflict graph but with a reduced graph. To illustrate this point, consider Corollary 1. According to the corollary we can remove a noncurrent transaction from the *conflict* graph; in fact it can be shown that we can remove all of them. This does not mean that we can remove a noncurrent transaction from a *reduced* graph that has resulted after deleting other transactions. For instance, in Example 1, both T_2 and T_3 satisfy condition C1 of Theorem 1, and therefore either of them can be safely removed; however, after T_3 is deleted, it is wrong to remove the noncurrent transaction T_2.

Also, it is not immediately clear what "necessary and sufficient condition" means in the dynamic environment. In its generic form, what we have here is an algorithm which operates continuously, is fed (transaction) steps as input, and depending on the past history makes a decision (to accept or reject them). The algorithm encodes incrementally the history in a data structure (a graph) on which it bases its decisions. We want a policy which as information about the past becomes redundant reduces the data structure, keeping only relevant information. What information is relevant? First, it is information sufficient for the algorithm to keep making the correct decision in the future. Second, we need information sufficient to keep reducing the data structure. It turns out here that if the information is sufficient for the algorithm to keep working correctly, then it is sufficient also for deducing all possible subsequent reductions.

More concretely, Rules 1–3 of Section 2 specify a function F from conflict graphs and transaction steps to conflict graphs: $F(\mathrm{CG}(p), t) = \mathrm{CG}(pt)$. F is extended naturally to sequences of steps:

$$F(\mathrm{CG}(p), tr) = F(F(\mathrm{CG}(p), t), r) = F(\mathrm{CG}(pt), r) = \mathrm{CG}(ptr),$$

where t is a step and r a sequence of steps. The conflict graph of p is $\mathrm{CG}(p) = F(E, p)$, where E is the empty graph.

The graph obtained by removing one or more completed transactions (possibly at different times) has the following properties: (1) it is acyclic, (2) its nodes are transactions of the schedule p executed so far including all active transactions, and (3) whenever two transactions present in the graph have executed two conflicting steps, there is an arc indicating the order; the graph may have also arcs connecting nonconflicting transactions (because of the previous removal of other transactions). Call any graph satisfying these properties a *reduced* graph of p. A schedule has only one conflict graph but may have many reduced graphs. The same rules (1–3) are applied to reduced graphs: when a transaction step arrives, the graph is modified according to the appropriate rule provided that the graph remains acyclic; as before, if a cycle is created, then the step is rejected, the transaction aborts, and it is deleted from the graph. Thus, F is well defined on the larger domain of reduced graphs. As in the previous section, for ease of notation, we will say that $F(G, r)$ is cyclic to mean that if the last step of r is executed, then a cycle will be created; of course, the last step will be rejected and its transaction will be aborted and deleted from the graph.

Removal of a completed transaction defines a transformation on reduced graphs G: $D(G, T_i)$ is G with node T_i deleted and all arcs to and from T_i replaced by arcs from the immediate predecessors of T_i in G to all its immediate successors. D can be naturally extended to sets of transaction N: $D(G, N) = D(D(G, T_i), N - \{T_i\})$. It is obvious that the order of deletion of nodes in N is immmaterial.

A *deletion policy* P is an algorithm which given reduced graph G (the current graph) outputs a set of (completed) nodes to be deleted. A deletion policy together with F (Rules 1–3) specify the behavior of the scheduling algorithm (thus also the set of schedules accepted): when a new transaction step arrives, the function F is applied to the current graph giving a new graph G; then the set of nodes $P(G)$ is removed. Call a deletion policy *correct* if the scheduling algorithm accepts only CSR schedules.

If p is a schedule, we denote by $R'_P(p)$ the graph formed by the scheduling algorithm after processing p *before* the removal of the (completed) transactions dictated by the deletion policy P, and we will denote by $R_P(p)$ the graph formed *after* the removal of these nodes. Formally, if λ is the empty schedule, $R_P(\lambda) = R'_P(\lambda) = E$, the empty graph. If p is a schedule and t a step, then $R'_P(pt) = F(R_P(p), t)$, and $R_P(pt) = D(R'_P(pt), N)$, where $N = P(R'_P(pt))$.

The dynamic problem of characterizing correct deletion policies will be reduced to a static problem. If G is a reduced graph and N a set of (completed) transactions of G, we say that the deletion of N from G is *safe* if for all continuations r, $F(D(G, N), r)$ acyclic $\Rightarrow F(G, r)$ acyclic. We say that a deletion policy P *performs only safe deletions* if for any schedule p, the deletion of $P(R'_P(p))$ from $R'_P(p)$ is safe.

LEMMA 3. *Let G be a reduced graph and N a set of completed transactions of G. The following are equivalent:*

(1) *For all continuations r, $F(D(G, N), r)$ acyclic implies $F(G, r)$ acyclic.*

(2) *For all continuations r, $F(D(G, N), r)$ acyclic if and only if $F(G, r)$ acyclic.*

Proof. The proof is similar to that of Lemma 2. The implication $(2) \Rightarrow (1)$ is obvious. To prove $(1) \Rightarrow (2)$ we take a shortest continuation $r = st$ which violates (2). Argue as in Lemma 2 that if there is a path between two transactions in $F(D(G, N), s)$, then there is such a path also in $F(G, s)$. As a consequence, the only way that (2) can be violated at step t, is if $F(G, r)$ has a cycle, but $F(D(G, N), r)$ is acyclic; i.e., (1) is also violated. ▌

The definition of safe deletions is justified by the following.

THEOREM 2. *A deletion policy is correct iff it performs only safe deletions.*

Proof. (if) Suppose that the deletion policy performs only safe deletions. We show that the reduced scheduling algorithm using this policy behaves exactly the same as the original conflict scheduler. The proof is by induction on the length of a schedule p. The inductive hypothesis is as follows: Let $G = R_P(p)$ be the current graph that has resulted after p, using the deletion policy P. For all continuations r, $F(G, r)$ acyclic $\Leftrightarrow \mathrm{CG}(pr)$ acyclic.

The basis ($|p| = 0$) is trivial: G is the empty graph E and $F(E, r) = \mathrm{CG}(pr)$. For the induction step let $p = st$, where t is the last step. Let $G_1 = R_P(S)$ be the current graph after s. From the induction hypothesis for G_1 and continuation t, the step t is rejected by the reduced scheduler (forms a cycle in G_1) iff it is rejected by the conflict scheduler. Let G_2 be the graph resulting after applying step t to G_1: $G_2 = F(G_1, t) = R'_P(p)$. Let $G = D(G_2, P(G_2)) = R_P(p)$ be the graph resulting from G_2 after removing the transactions dictated by the deletion policy. Let r be any continuation. From the induction hypothesis for G_1 and tr, $F(G_2, r) = F(G_1, tr)$ acyclic $\Leftrightarrow \mathrm{CG}(pr)$ acyclic. Since the deletion of $P(G_2)$ is safe, the resulting graph G satisfies: $F(G, r)$ acyclic $\Leftrightarrow F(G_2, r)$ acyclic $\Leftrightarrow \mathrm{CG}(pr)$ acyclic.

(only if) Suppose that the deletion policy P performs in some cases nonsafe deletions. Let p be a smallest schedule such that P performs a nonsafe deletion after p. Let r be a shortest continuation of p witnessing the nonsafety of the deletion. That is, if $G = R'_P(p)$ is the graph that has resulted after p before the deletion, $N = P(G)$, and $G_1 = D(G, N) = R_P(p)$, then r is a shortest continuation such that $F(G_1, r)$ is acyclic but $F(G, r)$ has a cycle. From our choice of p, all deletions performed during p were safe. We can argue as in the (if) part that during p the reduced and the conflict scheduler have behaved the same way, and that furthermore, for all continuations s of p, $F(G, s)$ acyclic $\Leftrightarrow$ CG(ps) acyclic. Therefore, CG(pr) has a cycle.

Consider now how the reduced scheduler (with deletion policy P) operates on input pr. During p it agrees with the conflict scheduler. We shall argue that there is a step t of r (not necessarily the last step) in which the two schedulers disagree. Furthermore, the earliest such step t is accepted by the reduced scheduler but rejected by the conflict scheduler. From this it will follow then that, if all active transactions completed at that point, the reduced scheduler would accept a non-CSR schedule; that is, the deletion policy is not correct.

If the reduced algorithm did not make deletions during the continuation r, then our claim about the existence of the step t as above would be true by Lemma 3: t is just the last step of r. To show that further deletions after the unsafe one will not help, we use repeatedly the argument of Lemma 3. Let n be the length of r, and r_i its ith step. Let S_i be the scheduler which uses the deletion policy up to step r_i of r, and then stops making deletions any more. We use induction on i to show that the claim is true of S_i; that is, there is a step t of r on which S_i and the conflict scheduler disagree, and the earliest such step is accepted by S_i.

The basis $i = 1$ is our previous observation. For the induction step, assume the induction hypothesis for i. If S_i and the conflict scheduler disagree in one of the first i steps of r, then the claim for S_{i+1} follows from the induction hypothesis. Thus, assume that S_i and the conflict scheduler agree on the first i steps, and let r_j be the earliest step on which they disagree; $j > i$. The schedulers S_i and S_{i+1} are in the same state right after processing of the step r_i; the difference is that at this point S_{i+1} performs a reduction whereas S_i does not. If S_i and S_{i+1} agree on the continuation $r_{i+1} \cdots r_j$, then the claim for S_{i+1} follows from the induction hypothesis. On the other hand, if they do not agree and r_k, $k \leqslant j$ is the first disagreement, then by Lemma 3, S_{i+1} accepts r_k whereas S_i rejects it. Since S_i accepts r_j, we must have $k < j$. Therefore, the first disagreement between S_{i+1} and the conflict scheduler is at r_k. ▌

A careful reading of the proof of the theorem shows that it does not depend on the particular rules (1–3) for adding edges. Also, no correct deletion policy can perform nonsafe deletions even if it knows the complete past schedule (not only the current graph). That is, all the information relevant to deciding whether some transactions can be safely deleted is contained in the reduced graph.

THEOREM 3. *Let G be a reduced graph. The deletion of a (completed) transaction T_i from G is safe iff T_i satisfies condition* C1 *of Theorem* 1 *with respect to G.*

Proof. The reader can go through the proof of Theorem 1 and verify that it works for any reduced graphs, not only conflict graphs. ▌

Even though two transactions may satisfy C1 (with respect to the current graph), it may not be safe to delete both of them. For instance, in Example 1, both T_2 and T_3 satisfy C1. However, only one of them (either one) can be safely deleted. We can use condition C1 repeatedly to prove:

THEOREM 4. *Let G be a reduced graph and N a subset of completed transactions. The deletion of N from G is safe iff the following condition is satisfied:*

(C2) *For all T_i in N, for all tight active predecessors T_j of T_i and for all entities x accessed by T_i, there is a completed tight successor of T_j not in N which accesses x at least as strongly as T_i.*

Proof. Let $N = \{T_1, ..., T_n\}$, and let G_i be the graph obtained by deleting $T_1, ..., T_i$ from G. It is not hard to see that the deletion of N from G is safe if and only if the deletion of T_i from G_{i-1} is safe for all i (where we let G_0 be G); the proof is essentially contained in Theorem 2. Also, from the definition of deletion it is obvious that the predecessor and tight predecessor relations in G_i are just the restrictions of these relations in G to the nodes of G_i. From these facts and Theorem 3 it is straightforward to show that (C2) implies the safety of the deletion.

For the converse, suppose that (C2) does not hold, and let T_i be a transaction in N with maximum index i for which (C2) is violated. Thus, there is an active tight predecessor T_j of T_i and an entity x assessed by T_i such that there is no completed tight successor of T_j outside N, which accesses x at least as strongly as T_i. We claim that T_i violates condition (C1) in G_{i-1}. For, suppose that T_j has a completed tight successor T_k in G_{i-1} which accesses x at least as strongly as T_i. Then T_k must be in N and $k > i$. By our choice of i, T_k does not violate (C2). Thus, there is a completed tight successor T_l of T_j not in N which accesses x at least as strongly as T_k, and therefore, at least as strongly as T_i. Thus, T_i does not violate (C2), a contradiction. ▌

Let M be the set of transactions which satisfy C1. Clearly, every set N which can be safely deleted is a subset of M. There may be many such safe subsets; in fact a single step may make many transactions candidates for deletion. Choosing the best safe subset is a difficult problem.

THEOREM 5. *Let G be a reduced graph. Finding the maximum subset of transactions which can be safely deleted is NP-complete.*

Proof. From Theorem 4, we can determine in polynomial time if a given subset can be safely removed. This implies membership in NP. The NP-hardness part uses a reduction from the set cover problem.

The set cover problem is the following problem. We are given a family F of subsets $S_1, ..., S_m$ of a set $X = \{x_1, ..., x_n\}$, and a number k. A *cover* of X is a collection of sets whose union is X. The set cover problem is to determine if F contains a cover of size at most k. This is a well-known NP-complete problem [GJ]. Given an instance of the set cover problem we will construct a schedule p and a number l such that

(1) no transaction can be removed safely from the conflict graph before the last step of p, and

(2) after the last step of p, we can remove safely at least l transactions from the graph iff there is a cover of size at most k.

We have transactions $T_1, ..., T_m$, one for every set in the family F, and two more transactions, T_0 and T_{m+1}. The structure of the schedule p is as follows. First, T_0 reads. Then, T_1, $T_2, ..., T_m, T_{m+1}$ execute to completion serially in this order. All these transactions become successors of the active transaction T_0; furthermore, none of them satisfies condition (C1) until the last step (when T_{m+1} writes), at which point $T_1, ..., T_m$ satisfy (C1). We shall construct the transactions in such a way that there is a 1–1 correspondence between a cover and the transactions that remain after a safe deletion.

In more detail we have one entity x_i for every element of X. In addition there are entities y, and $z_1, ..., z_m$. Transaction T_0 reads y and all elements of X. Transaction T_i with $1 \leqslant i \leqslant m$ reads z_i and writes the elements of S_i. Finally, T_{m+1} reads $z_1, ..., z_m$ and writes y. Because of S_i there is an arc from T_0 to T_i for all $1 \leqslant i \leqslant m$. At the last step, an arc is added from T_0 to T_{m+1} because of entity y. Up to that step, each T_i, $1 \leqslant i \leqslant m$ has accessed an entity, namely z_i, which has not been accessed by any successor of T_0. Therefore, no transaction can be safely removed. After the last step, transactions $T_1, ..., T_m$ satisfy C1. It follows easily from condition C2, that a subset N of these transactions ca be safely removed iff the remaining ones correspond to a cover. Therefore, we can remove at least $l = m - k$ transactions iff there is a cover of size at most k. ▌

To summarize, condition C1 (or C2 for sets of transactions) gives a necessary and sufficient condition for the removal of transactions. At any given point we may have several choices which may lead to reduced graphs of different sizes. Determining the best choice is an intractable problem. We note however, that if the number of active transactions and the size of the database are bounded, then any irreducible graph (graph from which no transaction can be removed) has also bounded size. To see this, associate with every completed transaction T_i in the graph the set of pairs (T_j, x) that witness the fact that T_i does not satisfy condition C1; i.e., T_j is an active tight predecessor of T_i, x is an entity accessed by T_i, and T_j does not have a completed tight successor that accesses x at least as strongly as T_i. Suppose that two completed transactions T_i and T_k had a common witness (T_j, x), and assume without loss of generality that T_k accesses x at least as strongly as T_i. It would follow then that (T_j, x) is not a witness for T_i because T_j has a completed tight successor (namely, T_k) that accesses x at least as strongly as T_i. We conclude that no two completed transactions in the graph have a common witness. Therefore, if the number of active transactions is a and the number of entities is e, an irreducible graph can have no more than $a \cdot e$ completed transactions (and, of course, a active transactions).

5. Variants of the Model

We axamine now how the criterion is affected if we relax the assumptions of Section 2. In each case, by Theorem 2, it suffices to find a necessary and sufficient condition for a single safe deletion.

Multiple Write Steps

In this model a transaction is an arbitrary sequence of read and write steps. At a consequence, a transaction A may read an entity written by an *active* transaction B. In this case we say that A *depends directly* on B. If for some reason transaction B aborts in the future, then transaction A must also abort. The abortion of A may in turn cause another transaction to abort (if it has read an entity from A), and so on. That is, if we let *depends* be the transitive closure of the "depends directly" relation, the abort of a transaction B causes the abortion of all transactions that depend on it. Therefore, a transaction cannot commit upon completion; it has to wait until it does not depend any more on any active transactions. Thus, at any point during a schedule we have now three types of transactions:

(A) Active.

(F) Finished but not committed yet: depend on some active transactions, and therefore may abort in the future.

(C) Committed: do not depend on active transactions, only on committed ones.

How does this new situation affect the necessary and sufficient condition C1 for closing transactions? First we need a clarification of the concept of *tight predecessor*. We shall use the letters A, F, and C to restrict the types of transactions that are allowed as intermediate nodes of a path; for example, an FC-path is a path all of whose intermediate nodes have completed (are of type F or C). Condition C1 of the atomic write model has the word "tight" in two places. In the first place it makes the condition more liberal: it says that we have to worry only for paths from an active transaction T_j to T_i that use completed transactions as intermediate nodes. In the second place it makes the condition more restrictive: we must find a second path from T_j to another node T_k whose nodes are completed. In the multiple writes model, in the first place the word "tight" must be replaced by FC; i.e., we have to worry about paths from to T_j to T_i that use both type F and C nodes. However, there is no right replacement for the second occurrence of "tight": what nodes should be allowed to appear on the second path, from T_j to T_k, depends on the nodes of the first path, from T_j to T_i. This fact introduces complications which make the problem NP-complete.

We shall introduce some notation and then state the necessary and sufficient condition for this model. If M is a set of active transactions, we let M^+ be the set of all transactions that depend on (transactions in) M. If G is the current graph and N a set of type A and F transactions, we let $G-N$ be the graph obtained by aborting the transactions of N (i.e., deleting the nodes in N and their incident edges from G). The necessary and sufficient condition for the removal of a committed transaction T_i from the graph G is as follows:

(C3) *For each set M of active transactions, for each entity x accessed by T_i, if $G-M^+$ has a FC-path from an active transaction T_j to T_i, then it has also a path from T_j to some other transaction T_k that accesses x at least as strongly as T_i.*

The nodes of the second path from T_j to T_k (including T_k) may be of any type, even active. The condition remains the same whether we require the first path to be arbitrary or contain only completed nodes.

LEMMA 4. *Condition* C3 *is necessary and sufficient for the deletion of a committed transaction T_i to be safe.*

Proof. Similar to the proof of Theorem 1. ▮

THEOREM 6. *Let G be a conflict graph. It is* NP-*complete to decide* (i) *whether G cannot be reduced and* (ii) *whether a particular transaction T_i cannot be safely deleted from G.*

Proof. We can test in NP if a particular transaction T_i violates C3: we just have to guess the "right" set M of active transactions. Once M is fixed, we can construct M^+ and check the rest of the condition in polynomial time. The graph G cannot be reduced if no transaction can be safely removed.

For the NP-hardness part we use a reduction from 3-SAT [GJ]. Let f be a formula in conjunctive normal form with n variables $x_1, ..., x_n$ and m clauses $c_1, ..., c_m$ with 3 literals each. We construct a conflict graph G as follows (see Fig. 3). For each variable x_i, we include two type F transactions x_i, $\bar{x}_i$, and two type A transactions A_i, $\bar{A}_i$. For each clause c_j we have three type F transactions c_{j1}, c_{j2}, c_{j3}, one for each literal. In addition we have an active transaction A and three committed transactions B, C, D.

Our graph G has two kinds of arcs: arcs caused by write–write conflicts, and arcs caused by write–read conflicts. Each arcs is labeled by a distinct entity which is not accessed by any other transaction besides the two endpoints of the arc. Thus, the write–read arcs show actually the direct dependencies among transactions; the transaction at the head of such an arc reads an entity from the transaction at the tail. These arcs are drawn as dashed arrows in Fig. 3. The graph has the following write–write arcs. For each $i = 1, ..., n-1$, arcs from x_i and $\bar{x}_i$ to x_{i+1} and $\bar{x}_{i+1}$; from A to x_1 and $\bar{x}_1$; from x_n and $\bar{x}_n$ to B; from B to C; from A_i and $\bar{A}_i$ (for all i) to D; for each clause c_j, arcs forming a path $A \to c_{j1} \to c_{j2} \to c_{j3} \to D$ from A to D. The

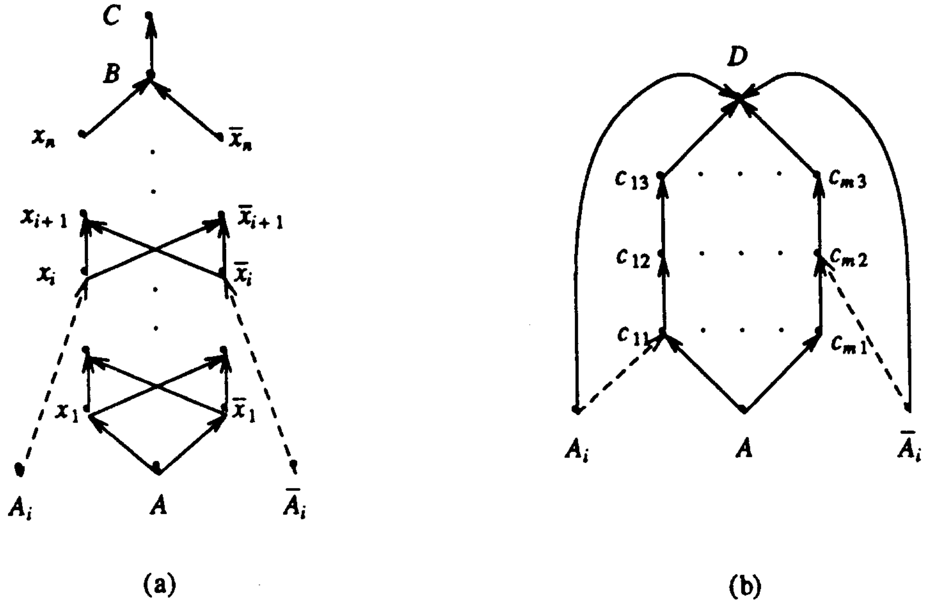

FIGURE 3

write–read arcs are $A_i \to x_i$, $\bar{A}_i \to \bar{x}_i$ for $i = 1, ..., n$; $A_i \to c_{jk}$ if the kth literal of clause c_j is x_i, and $\bar{A}_i \to c_{jk}$ if it is $\bar{x}_i$.

Each transaction accesses the entities which label its incident arcs. In addition, all transactions except C have written an entity which is not accessed by any other transaction. Transaction C has read an entity y which has been read only by D.

Note that the graph G is acyclic. A schedule p, whose conflict graph is G, can be obtained by executing the steps of the transactions serially in a topological order of G. We shall argue that the only (committed) transaction whose deletion may be safe is transaction C, and that the deletion of C is safe if and only if the formula f is not satisfiable.

The fact that every transaction except C writes a private entity implies immediately that it violates condition C3. Transaction C accesses two entities: y, and the entity, call it z, labeling the arc $B \to C$. Note that any path in G from an active transaction to C must pass through B. Therefore, for any set M of active transactions, if $G - M^+$ has a path from an active transaction T_j to C then it has also a path from T_j to another transaction (namely, B) which accesses z at least as strongly as C. Therefore, in order to check if condition C3 holds for transaction C, we only have to worry about entity y. Thus, the role of T_k in C3 will be played by transaction D. Another observation is that C3 holds if the role of the active transaction T_j is played by A_i or $\bar{A}_i$, regardless of the choice for M. This is because of the arcs $A_i \to D$ and $\bar{A}_i \to D$. To summarize, in checking C3 for transaction C we may substitute y for x, D for T_k and A for T_j.

Suppose now that f is satisfiable and pick a satisfying truth assignment. Let M consist of those transactions A_i whose corresponding variable x_i is true, and $\bar{A}_i$ for which x_i is false. The set M^+ includes all true literals c_{jk} from the clauses. Since the truth assignment satisfies f, no path is left in $G - M^+$ from A to D. However, there is clearly a path from A to C, and therefore C3 does not hold.

Conversely, assume that C3 does not hold and pick a set M that violates it. Since $G - M^+$ contains a path from A to C, for each i, either x_i or $\bar{x}_i$ is not in M^+. That is, either A_i or $\bar{A}_i$ is not in M. Consider the assignment which sets a variable x_i to true if $A_i \in M$ and to false otherwise. If $\bar{A}_i$ is in M, then A_i is not, and therefore x_i is false. Since $G - M^+$ has no path from A to D, for each clause at least one of the literals depends on a member of M, that is, at least one of the literals is true, and the formula is satisfied. ▌

Predeclared Transactions

If transactions predeclare the entities they are going to read and write, then aborts can be avoided. The conflict scheduler can use the extra information to predict future cycles in the conflict graph and prevent them from happening by delaying steps. It does so by adding an arc to the graph as soon as the first of the two conflicting steps takes place. In more detail, the rules are as follows.

RULE 1. *When a new transaction T_i starts, a node T_i is added to the graph. For every other transaction T_j which has executed a step conflicting with a (future) step of T_i add an arc $T_j \rightarrow T_i$.*

RULE 2 AND 3. *Suppose T_i wants to read or write x. For every other transaction T_k which will perform in the future a conflicting step on x, add an arc $T_i \rightarrow T_k$, provided that no directed cycle is formed; if the arc would create a cycle, then T_i has to wait for T_k to execute its conflicting step.*

Note that there is no danger of deadlock (cyclic wait among transactions). To see this, observe that if T_i waits for T_k then the graph has a path from T_k to T_i. Since the graph is acyclic at all times, the same is true of the "waits-for" graph.

The necessary and sufficient condition for the safe deletion of a transaction T_i is somewhat more complicated than C1, although it can still be tested in polynomial time. The condition holds even in the multiple write model, and is as follows:

(C4) *For all active predecessors T_j of T_i and for all entities x accessed by T_i, either*

1. *T_j has another successor T_k ($\neq T_i, T_j$) which has accessed x at least as strongly as T_i, or*
2. *every entity y that T_j will access in the future has already been accessed at least as strongly by some successor T_k ($\neq T_i$) of T_j.*

The second clause was omitted from a preliminary version of this paper that appeared in the PODS 86 conference. As we shall see, active transactions which

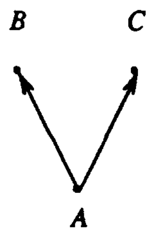

FIGURE 4

satisfy this clause behave essentially as completed, in the sense that they will not acquire any more immediate predecessors in the future. The following example illustrates this.

EXAMPLE 2. Consider the following schedule p of three transactions A, B, C. First A reads entities u, z; then B reads y, writes u and completes; then C writes x and z and completes. Transaction A is still active with one remaining step which reads y. The graph at this point is shown in Fig. 4.

Transactions B and C have both an active predecessor, A. Transaction B does not satisfy C4, but C does and can be safely removed. The reason is that the only way A can acquire a new immediate predecessor D, is if this new transaction D writes y before the read step of A. But as soon as D starts and declares its steps, an arc $B \rightarrow D$ will be added to the graph. As a consequence, D will be prevented from writing y before A reads it.

THEOREM 7. *Condition* C4 *is necessary and sufficient for the safe deletion of a (completed) transaction* T_i *in the case of predeclared transactions*

Proof. Let p be a schedule, $G(p)$ the current graph after p, T_i a completed transaction, and $R(p, T_i)$ the reduced graph with T_i removed. For a continuation r, we will denote by $G(pr)$ the graph that results after pr from the original scheduler (without the deletion of T_i), and by $R(p, T_i, r)$ the reduced graph.

(Sufficiency) Assume C4 and consider a shortest continuation r for which the original and the reduced scheduler disagree. As before, $G(pr)$ has a cycle but $R(p, T_i, r)$ is acyclic. First we argue that any active transaction T_j satisfying clause (2) of C4 has not acquired any new immediate predecessors. For suppose that $D \rightarrow T_j$ is a new arc. This arc must have been caused by a step t of D in r which conflicts with a later step t' of T_j. By (2), some successor T_k of T_j has executed in p a step which conflicts with the step t of D. Therefore, right before t, there is an arc $T_k \rightarrow D$ in both the unreduced and the reduced graph. Thus, T_j is a predecessor of D before the step t, and t will not be accepted (it will be delayed).

Consider now the cycle that is formed in $G(pr)$. As in Theorem 1, it must contain T_i. Also, the arc from T_i to its immediate successor T_l on the cycle must be new (i.e., not present in $G(p)$), because otherwise the reduced graph would contain an arc from the immediate predecessor of T_i to T_l. Starting at T_i walk backwards on the cycle until a new arc (not in $G(p)$) is first encountered, and let T_j be the head of this arc. Transaction T_j must have been active at the end of p, and from the previous observation, it does not satisfy clause (2). The argument from here on is identical to Theorem 1.

(Necessity) Suppose that C4 does not hold. That is, there is an active predecessor T_j of T_i and an entity x accessed by T_i such that neither clause (1) nor (2) is satisfied. We shall construct a continuation r such that $G(pr)$ contains a cycle but $RG(p, T_i, r)$ is acyclic.

Let M be the set of all transactions which are not successors of T_j. Clearly, a predecessor of a node in M is itself in M. The continuation r at first completes all active transactions in M serially in a topological order. These steps will be accepted without delay both by the original and the reduced scheduler. The reason is that when a member of M executes its remaining steps it has no active predecessors, and therefore by the rules no cycle will be formed. All arcs that are added to the (original or reduced) graph during these steps have nodes of M as their tails. Therefore, the succesors of T_j do not change.

Let y be an entity which witnesses the fact that T_j does not satisfy clause (2). The final two steps of r belong to a new transaction T_n and access x and y in the weakest mode that conflicts with T_i and the future step of T_j respectively. That is, if T_i has written x then T_n reads x, otherwise T_n writes x. Similarly with y; if T_j will write y then T_n reads y, otherwise it writes y. These two steps will add edges $T_i \to T_n$, $T_n \to T_j$ which together with the path from T_j to T_i create a cycle in the original graph.

We claim that no cycle will be created in the reduced graph. Such a cycle must necessarily involve T_n. When T_n starts and declares its steps, we will add arcs in the reduced graph from all nodes which have already performed conflicting steps on x and y. By (1) of C4, no successor of T_j in the reduced graph has performed a conflicting step on x. By (2) the same is true of y. Therefore, all arcs into T_n come from nonsuccessors of T_j. On the other hand, the last two steps of T_n cause arcs from T_n only to active transactions; these are all successors of T_j. For a cycle to form there must be an arc from a successor of T_j to a nonsuccessor, which is impossible. If the remaining active transactions complete their steps now serially in a topological order of the graph $R(p, T_i, r)$, the reduced scheduler has accepted a non-CSR schedule. ∎

6. Conclusions

We have studied the problem of when it is safe to remove a transaction that has completed without jeopardizing correctness. We proved necessary and sufficient conditions in several versions of conflict-graph-based schedulers and analyzed their complexity.

We also formulated the corresponding dynamic problem of repeatedly deleting transactions as they become irrelevant and solved it by reducing it to the static problem of a single deletion. We believe the same techniques may be applicable in other similar situations, where we have an algorithm which operates continuously taking decisions depending on the past history, and we want to remove information as it becomes redundant.

REFERENCES

[BHR] R. BAYER, H. HELLER, AND A. REISER, Parallelism and recovery in database systems, *ACM Trans. Database Systems* **5** (1980), 139–156.

[BEG] P. A. BERNSTEIN AND N. GOODMAN, A sophisticate's introduction to Distributed database concurrency control, *in* "Proceedings, 8th Internat. Conf. on VLDB, 1982," pp. 62–76.

[BOG] H. BORAL AND I. GOLD, Towards a self-adapting centralized concurrency control algorithm, *in* "Proceedings, ACM-SIGMOD '84," pp. 18–32.

[GJ] M. R. GAREY AND D. S. JOHNSON, "Computers and Intractability: A Guide to the Theory of NP-Completeness," Freeman, San Francisco, 1978.

[EGLT] K. P. ESWARAN, J. N. GRAY, R. A. LORIE, AND I. L. TRAIGER, The notions of consistency and predicate locks in a database system, *Comm. ACM* **19** (1976), 624–633.

[Ha] V. HADZILACOS, A theory of reliability for database systems, to appear.

[HP] T. HADZILACOS AND C. H. PAPADIMITRIOU, Some algorithmic aspects of multiversion concurrency control, *in* "Proceedings, 4th ACM Symp. on PODS, 1985," pp. 96–104.

[Pa] C. H. PAPADIMITRIOU, "The Theory of Database Concurrency Control," Computer Sc., Rockville, MD, 1986.

[S] G. SCHLAGETER, Process synchronization in database systems, *ACM Trans. Database Systems* **3** (1978), 248–271.

[SK] A. SILBERSCHATZ AND Z. KEDEM, Consistency in hierarchical database systems, *J. Assoc. Comput. Mach.* **27** (1980), 72–80.

[SR] R. E. STEARNS AND D. J. ROSENKRANTZ, Distributed database concurrency controls using before-values, *in* "Proceedings, ACM-SIGMOD '81," pp. 74–83.

[Y] M. YANNAKAKIS, A theory of safe locking policies in database systems, *J. Assoc. Comput. Mach.* **29** (1982), 718–740.