STOC(Milwaukee 1981), 363-367.

ISSUES OF CORRECTNESS IN DATABASE CONCURRENCY CONTROL BY LOCKING

Mihalis Yannakakis

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

In a database system several users may read and update information concurrently. If the operations of the various user-transactions are not interleaved in a correct fashion, several undesirable situations may arise such as creation of inconsistent data or users receiving inconsistent information. The concurrency control problem is to coordinate the concurrent accesses of the database by the various transactions so that the effect is the same as if the transactions ran one-at-a-time.

Most systems use *locking* as a mechanism for controlling concurrency. That is, each transaction locks groups of data according to some *locking policy*; then the *scheduler* (the part of the database manager system that controls access to the database, i.e. grants, rejects or suspends requests to read or write) just keeps track of the locks and makes sure that no two transactions lock the same data in conflicting modes [*E*,*G*,*LW*,*SK*,*YPK*]. Presumably the locking policy is designed so that only correct schedules will be produced; such a locking policy is said to be *safe*.

The design of the concurrency controller involves the modelling and understanding of when a schedule is correct. Of course a *serial* schedule, one in which the transactions ran one-at-a-time, is correct. The paper by Eswaran et. al. [*E*] was the first to formalize mathematically the concurrency control problem. In [*E*], correctness of a schedule S (called *serializability* there) was defined as follows. Two steps of two transactions are said to *conflict* if they involve the same data and at least one of the two steps updates this data. A schedule S is defined in [*E*] to be correct if there is some serial schedule S' in which pairs of conflicting steps are executed in the same order as in S. [E] proves a very simple condition for a schedule S to satisfy this criterion: From S a directed graph $D(S)$ is constructed so that S satisfies the correctness criterion if and only if $D(S)$ is acyclic.

Most existing or proposed solutions to the concurrency control problem enforce Eswaran's criterion [*KS*, *G*, *LW*, *S*, *SK*, *Y*1]. This criterion, although defined syntactically, seems quite reasonable: It seems at first sight that reversing the order of two conflicting steps will alter the result. Surprisingly, [*P*1] shows that this is not true: Correctness of a schedule is a much more intricate property than this criterion suggests - in fact it is NP-complete to test whether the effect of the schedule on the database is the same as that of some serial schedule. Thus, Eswaran's criterion (termed *D-serializability* in [*P*1]) is a much stronger requirement than correctness.

Our aim in this paper is to show that there is a mathematically inherent reason why existing systems enforce D-serializability (rather than just because of its simplicity): it is because they are based on locking. Our main result is a characterization of the power of locking which states that if a locking policy is safe then it must allow only D-serializable schedules. Furthermore any such schedule can be produced by some safe locking policy.

The rest of the paper is organized as follows. In Section 2 we formalize our concepts and describe the model. In Section 3 we characterize D-serializability in semantic terms. In Section 4 we examine when a set of transactions can be let to run safely by themselves without locking or any intervention from the scheduler. Section 5 is concerned with locking policies and in Section 6 we discuss some implications of our results.

2. THE MODEL

A database is partitioned into *entities* (e.g. files, records, etc.). An entity is a group of data that is considered as an atomic unit as far as reading and updating is concerned; i.e. no two user transactions access (parts of) an entity simultaneously. A transaction is a finite sequence of *steps*. Each step reads some entity x (denoted rx for short) or writes x (wx) *. The value written by a transaction in a write step is an uninterpreted function of the values read thus far. We assume for simplicity that a transaction does not write an entity more than once and does not read an entity after it has written it. A *transaction system* τ is a set $\{T_1,\ldots,T_n\}$ of transactions. A *schedule* S of τ is an execution of the transactions of τ in a (possibly) interleaved fashion. Note that each transaction must execute its steps in S in the prescribed order. A schedule is *serial* if there is no interleaving, i.e. once a transaction starts executing it finishes without any other transaction executing some step in-between. A *state* of the database is an assignment of values to the entities. The *view* of a transaction T in a schedule S is the set of values read by T in its read steps.

A schedule is correct if it behaves like a serial schedule. We will deal with two correctness criteria depending on how "behaves" is understood. We say that a schedule S of τ *presents serial views* (PSV for short) if there is a serial schedule S' of τ such that for every initial state of the database, and every interpretation of the various functions computed by the transactions, each transaction of τ receives the same view in S and S', and execution of S or S' will leave the database in the same final state. We say that S and S' are *view equivalent*.

Thus, presentation of serial views means that as far as the final database and the transactions in τ are concerned, S behaves like the serial schedule S'.

A schedule S of τ has a *serial final state* (SFS for short) if

* Some other transaction models have been also studied; e.g. the *action model* in which a *step* of a transaction reads and then writes an entity x [*E*,*YPK*], and the *2-step model* in which a transaction has 2 steps, a first one in which it reads a set R of entities and a second one in which it writes a set W of entities [*P*1,*PBR*]. Our results are valid for these restricted models too.

there is a serial schedule S' of τ which, for every initial state of the database and interpretation of the functions computed by the transactions, produces the same final state as S. (This is the *serializability* criterion of [*P1*, *PBR*, *BSW*].) We say that S and S' are *final state equivalent*.

Some other (weaker) criteria have been proposed based on the integrity constraints of the database, such as preserving the consistency of the database or presenting consistent views to the transactions. However, schedules can be unsatisfactory even if they satisfy these consistency-based criteria. For example, in an airline reservation system it is possible that a traveler's reservation is lost - overwritten by another person - while the consistency of the database is being preserved. [*Y2*] presents a detailed comparison of the various correctness criteria in the different transaction models.

It is easy to test whether a schedule is view equivalent or final state equivalent to a given serial schedule. Let us add two transactions, an initial transaction T_{in} that writes all entities at the beginning, and a final transaction T_f that reads all entities at the end. A read-step rx of a transaction T_k *reads* x *from* transaction T_l in a schedule S, if the last wx step *before* the rx step of T_k belongs to T_l. We define two binary relations on the steps of a schedule S: "immediately-affects" and "affects". A step s_1 *immediately-affects* step s_2 if either s_2 reads some entity from s_1, or s_1 is a read-step of a transaction T preceding s_2, a write-step of the same transaction T. "*Affects*" is the transitive closure of "immediately-affects". A step is *live* if it affects a step of the final transaction T_f; otherwise it is *dead*. We can define now two relations from a schedule S: READ-FROM and LIVE-READ-FROM.

READ−FROM $(S) = \{(T_i, x, T_j)|$ a step rx of T_j reads x from $T_i\}$.
LIVE−READ−FROM $(S) = \{(T_i, x, T_j)|$ a
live step rx of T_j reads x from $T_i\}$.

We view these relations also as directed graphs with the transactions as nodes and the arcs labelled with entities. It is easy to see that a schedule S is view-equivalent to the serial schedule S' if S and S' have identical READ-FROM relations, and S is final-state-equivalent to S' if they have identical LIVE-READ-FROM relations ([*PBR*]).

Two steps of two (different) transactions *conflict* if they involve the same entity and at least one of them is a write-step. From a schedule S we construct a relation (labelled directed graph) $D(S) = \{(T_i, x, T_j)|$ a step of T_i involving x precedes in S a conflicting step of $T_j\}$. Schedule S is *D-serializable* (DSR for short) if there is a serial schedule S' with the same D relation. Clearly, DSR is a sufficient condition for presentation of serial views, but not a necessary one. (See, for example, [*BCG*, *P1*] for several counterexamples to a theorem of [*S*] claiming the contrary).

Proposition 1 [*E*]. A schedule S is D-serializable if and only if $D(S)$ is acyclic. □

We can define a similar sufficient condition for serial-final-state correctness. Let $LD(S) = D(\bar{S})$ where $\bar{S}$ is the schedule obtained from S by deleting dead read steps. We say that S is *Live D-serializable* if there is a serial schedule with the same LD relation. It is again easy to see that Live DSR is a sufficient condition for SFS, and that:

Proposition 2. A schedule S is Live D-serializable if and only if $LD(S)$ is acyclic. □

Note that from the definitions of $D(S)$ and $LD(S)$, all arcs leaving the initial transactions T_{in} and all arcs entering the final transaction T_f can be inferred from the rest of the graphs. For this reason we will drop these two dummy transactions from the D and LD graphs.

3. A CHARACTERIZATION OF DSR AND LIVE DSR

The key notion for our characterizations is that of a *subschedule*. Let S be a schedule of a transaction system τ and $\tau' \subseteq \tau$. The *subschedule* $S(\tau')$ of S *generated* by τ' is the subsequence of S formed by the steps of the transactions in τ'.

Theorem 1. A schedule is D-serializable if and only if all its subschedules present serial views.

Proof. *(only if).* Suppose that S is D-serializable, and let S' be a subschedule of S. From the definition of the digraph D, $D(S')$ is a subgraph of $D(S)$ and therefore is acyclic. Thus (Proposition 1), S' presents serial views.
(if).
Claim 1 Let S_1 be a schedule of a set of transactions τ_1. Suppose that $D(S_1)$ contains an arc $T_i \rightarrow T_j$ labelled x and suppose that no other transaction of τ_1 (besides T_i,T_j) writes x. Then in any serial schedule view equivalent to S_1, transaction T_i must precede transaction T_j.
Proof of Claim 1 Let us call the arc $T_i \rightarrow T_j$ a w-w or r-w or w-r arc depending on the type of the conflicting steps involving x of the two transactions. Suppose that the arc is a w-w arc. Then, since no other transaction writes x, the final value of x in S_1 is written by T_j and therefore the same must be true in any serial schedule S_1' view equivalent to S_1. If the arc is a w-r arc then T_j reads x from T_i and the same must hold in any S_1' view equivalent to S_1. If it is a r-w arc, and in some serial S_1', T_j precedes T_i, then T_i will read x in S_1' from T_j since no other transaction writes x. □

Suppose now that $D(S)$ has a cycle and pick a shortest cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_k \rightarrow T_1$. Let $\tau_1 = \{T_1,\ldots,T_k\}$ and S_1 be the subschedule of S generated by τ_1.

Case 1 $k=2$
Since $|\tau| = 2$ we can apply Claim 1 and deduce that if an arc $T_i \rightarrow T_j$ is in D $(i,j=1,2)$ then T_i must precede T_j in a serial schedule view equivalent to S_1. Thus, if $T_1 \rightarrow T_2 \rightarrow T_1$ there is no such schedule
Case 2 $k=3$

Let $T_1 \rightarrow T_2$ be an arc labelled x. We will show at first that T_3 does not write x. If T_3 had a wx step, then because all arcs between T_2 and T_3 are directed from T_2 to T_3 this step should occur after all actions of T_2 on x; since all arcs between T_3 and T_1 are directed from T_3 to T_1, the wx step of T_3 should occur before all actions of T_1. But then all actions of T_1 on x must follow all actions of T_2 on x contradicting the fact that there is an edge $T_1 \rightarrow T_2$ labelled x.

Therefore, Claim 1 is applicable and we conclude that there is no serial schedule view equivalent to S_1
Case 3 $k>3$

Let $T_i \rightarrow T_{i+1}$ be an arc labelled x. If another transaction T_j had a wx step, then T_j would have an arc connecting it to T_i and T_{i+1} and therefore we should have $k=3$ since the cycle chosen was minimal. Thus, again Claim 1 can be applied.□

The direct analogue of Theorem 1 for Live DSR and SFS does not hold; that is, a schedule might be Live DSR and still have some subschedule which does not have a serial final state. The correct analogue is:

Theorem 2. All subschedules of a schedule are Live D-serializable if and only if they all have serial final states.

The reason for this difference is the fact that the digraph $LD(S')$ of a subschedule S' of schedule S is not necessarily a subgraph of $LD(S)$; some steps might be live in S but not S' and vice-versa. It is this fact that causes the main complications in the proof of Theorem 2 (which is in fact stronger than Theorem 1).

Example. Consider the following schedule S of the set

$\tau = \{T_1, \ldots, T_5\}$ (horizontal axis is time).

T_1: rx wy
T_2: ry wz
T_3: ru wy
T_4: wu wx
T_5: wy

The rx step of T_1 is dead. Figure 1(a) shows the graph $LD(S)$.

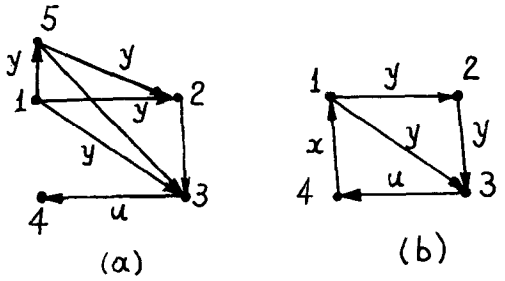

Figure 1

$LD(S)$ is acyclic. Thus, S is Live D-serializable, and therefore has a serial final state. Figure 1(b) shows the graph $LD(S_1)$ of the subschedule S_1 generated by the set $\{T_1, T_2, T_3, T_4\}$. All read steps are live in S_1, which is neither Live DSR nor SFS. However, the subschedule generated by the minimal cycle $\{T_1,T_3,T_4\}$ of $LD(S_1)$ is Live DSR : the rx step of T_1 becomes again dead. □

To prove Theorem 2 we use a necessary (but not sufficient) condition for SFS. We define a subgraph $H(S)$ of $LD(S)$ which consists of those arcs in $LD(S)$ that are either (i) LIVE READ-FROM arcs, or (ii) write-write arcs with the second write step being the last one for the corresponding entity in the schedule, or (iii) read-write arcs with the read step reading from the input database. Clearly (as in Claim 1 of Theorem 1), none of these arcs can be reversed in a serial schedule equivalent to S. Therefore, if S has a serial final state then $H(S)$ must be acyclic. Theorem 2 is based on

Lemma 1. $H(S')$ is acyclic for every subschedule S' of S if and only if $LD(S')$ is acyclic for every subschedule S' of S.

To prove Lemma 1, we must find, given a schedule S with a cyclic $LD(S)$, a subschedule S' with a cyclic $H(S')$. We will just sketch here the main points of the proof.

Selection 1. Choose τ_1 to be a minimal set of transactions such that $LD_1 = LD(S_1)$ has a cycle, where $S_1 = S(\tau_1)$.
Fact 1: LD_1 is strongly connected.

Selection 2. Choose C to be a cycle in LD_1 with the minimum number k of edges that are not LIVE-READ-FROM.

If $k = 0$, then $H(S)$ has a cycle. Thus, let's assume $k \geq 1$. The transactions in C are divided into *groups* with each group being a path of READ-FROM arcs. Let R be the set of transactions that can reach in $LRF_1 = LIVE-READ-FROM\ (S_1)$ a node in C.
Fact 2. R is partitioned into sets corresponding to the groups of C, such that each transaction of R can reach (and be reached) in LRF_1 only (by) nodes in the corresponding group.

Thus, for each group g of C we get an *expanded group* $\bar{g}$ that includes also the transactions in R that correspond to g.

Fact 3. (1) The subgraph of LRF_1 induced by $\bar{g}$ has a unique source - the first node v_1 of g - and a unique sink - the last node v_l of g. (2) Moreover, it has a Hamilton path from v_1 to v_l.

Therefore, we can expand C to a cycle C' with every transaction of C' reading only from the input or preceding transactions in its (expanded) group.

Selection 3. Among the cycles C of Selection 2 choose the one with the shortest C'.

Fact 4. All arcs of C' (and C) are also in $H(S_1)$. □

4. SAFE TRANSACTION SYSTEMS

A transaction system τ is *safe with respect to criterion A* (A is PSV, SFS, DSR, Live DSR) or *A-safe* if every schedule S of τ satisfies the criterion A.

Safety is a hereditary property; i.e.

Proposition 3. If τ is safe with respect to A (A = PSV, SFS, DSR, Live-DSR) then every subset of τ is also safe with respect to A . □

An immediate consequence of Theorems 1 and 2 now is:
Corollary 1. (1) A system τ is PSV-safe iff it is D-safe.
(2) A system τ is SFS-safe iff it is Live D-safe. □

[PBR] showed that it is NP-hard to test if a transaction system is not SFS-safe*, and conjectured that the problem is in NP. From the Corollary it follows that this is true: we just have to guess a schedule S of τ and verify that $LD(S)$ contains a cycle.

In the case of the PSV criterion, however, things are much simpler: [PBR] shows how to test efficiently a transaction system for D-safety in the 2-step model, and [Y1] in the action model. The method for the multistep model is similar. We construct a labelled multigraph $G(\tau)$ with the transactions as nodes and an edge (T_i, T_j) labelled x if T_i and T_j have two conflicting steps involving x.

Theorem 3. A set τ of transactions is D-safe if and only if (1) no entity x that appears in two steps of a transaction T is written by another transaction, and (2) $G(\tau)$ does not contain a cycle with two edges having different labels. (Two parallel edges are considered to form a cycle). □

We can easily check part (2) of Theorem 3 by looking at the labels on the edges of each biconnected component of $G(\tau)$.

5. LOCKING POLICIES

In general, a locking policy uses a set LV of *locking variables* (not necessarily related to the entities) which can be locked in certain *modes* (e.g. shared, exclusive etc.) There is a *compatibility relation* among the various modes which describes whether two different transactions can hold simultaneously locks on a common variable. For example, "shared" is compatible with "shared" but not with "exclusive" mode. A *locked transaction* is a sequence of *action steps* (read and write), *locking steps* (Lock in mode m variable x - Lmx) and *unlocking steps* (Unlock in mode m variable x - Umx), put together in a well-formed manner; i.e. no variable is unlocked in mode m unless it is actually locked in this mode, every variable is eventually unlocked in every mode it is locked, etc. A *legal schedule* S of a set of locked transactions is a schedule that observes the compatibility relation.

A *locking policy* P is a mapping from the set of all transactions on the set of entities E to the power-set of locked transactions such that for each transaction T and each $\bar{T} \in P\ (T)$, T and $\bar{T}$ have the same action steps in the same order. Thus, for example, the *two-phase policy* of $[E]$ has the set E of entities as locking variables, uses shared and exclusive modes of locking, and maps a transaction T to the set $P(T)$ of locked transactions $\bar{T}$ such that (1) when $\bar{T}$ reads an entity x it has locked it in shared or exclusive mode, and when it writes x it has locked x in exclusive mode, and (2) no unlocking step

[PBR] shows this for the 2-step model; essentially the same proof goes through also for our multistep model.

of $\bar{T}$ precedes a locking step.

Note that no assumption is made on the complexity of computing P (say, testing $\bar{T} \in P(T)$). Also, the possibility that $P(T) = \emptyset$ allows the locking policy to reject unacceptable transactions, and thus have complete control (and knowledge) over which transactions can run in the system.

A locking policy P *ensures criterion* A, (or is *A-safe*) if for every set $\{T_1,\ldots,T_k\}$ of transactions on E with $P(T_i) \neq \emptyset$, and every $\bar{T}_i \in P(T_i)$, any legal schedule of $\{\bar{T}_1, \ldots, \bar{T}_k\}$ satisfies A. Thus, for example a transaction system τ is A-safe if the locking policy P, defined by $P(T) = \{T\}$ if $T \in \tau$, and $\emptyset$ otherwise, is A-safe.

Any subschedule of a legal schedule of a locked transaction system is obviously also legal. Thus, from Theorems 1 and 2 we have.

Corollary 2. A locking policy ensures Presentation of Serial Views (resp. Serial Final State) if and only if it ensures D-serializability (resp. Live D-serializability). □

Thus, Corollary 2 explains (and justifies) why the various systems enforce DSR: it is necessary if one wants to ensure presentation of serial views when locking is used to control concurrency. A similar argument can be made also for schedulers as in Protocol P3 of SDD-1 [*BS*, *BSW*, *PSR*] that are not based on locking. This protocol works with the 2-step model. Briefly speaking it works as follows. From the set of possible transactions a graph (similar to the one in Theorem 3) is constructed; then, from this graph a binary relation "*guardian-of*" is defined among the transactions. The protocol P3 requires that if T_i is a guardian of T_j then T_j cannot execute its write-step between the read- and the write-step of T_i. This protocol can be implemented using locks as follows. For each pair (T_i, T_j) in the guardian-of relation we have a locking variable z_{ij}. The locking policy P encloses transaction T_i and the write-step of T_j into a $LX\ z_{ij} - UX\ z_{ij}$ pair (X stands for exclusive mode). Protocol P3 then is equivalent to observing these locks. Since P3 can be thought of as a locking policy, Corollary 2 is applicable. (Strictly speaking, the criterion in P3 is SFS; however if no transactions are really dead - e.g. they print out something [*BS*] - then SFS and PSV are equivalent.)

Let us now see why the notion of subschedules and their correctness was the right way for characterizing the schedules of safe locking policies.

Theorem 4. Let S be any schedule. There is a locking policy P whose legal schedules are exactly all serial executions of subschedules of S. □

Corollary 3. (1) Every D-serializable schedule can be produced by some PSV-safe locking policy. (2) Every schedule all of whose subschedules are Live D-serializable can be produced by some SFS-safe locking policy. □

In part (2) of Corollary 3 we have to require Live D-serializability from all subschedules because, as we mentioned in Section 3, a schedule might be Live-DSR and still have a subschedule that does not satisfy the serial final state criterion. Unfortunately, it is not possible to get rid of this quantification while keeping the criterion simple, since:

Theorem 5. It is NP-complete to test whether a schedule has a subschedule that is not Live-DSR. □

Corollary 4. It is NP-complete to test whether a schedule cannot be produced by any SFS-safe locking policy. □

Papadimitriou [*P2*] examines the schedules produced by locking policies in a different way: He proves a necessary and sufficient condition for a set of schedules of a transaction system to be realizable by some *single* (not necessarily safe) locking policy. Here we are interested rather on describing the set of schedules that can be produced by *all safe* locking policies. The definition of a locking policy is also slightly different there in that $|P(T)| \leq 1$ for each T. Using the criterion of [*P2*], it is easy to construct an example of two small transactions (3 steps each) such that no locking policy (in the sense of [*P2*]) can realize all their DSR schedules.

This is not the case if we allow $|P(T)| > 1$. Corollary 3 can be strengthened to:

Theorem 6. For every set τ of transactions, there is a locking policy (using only exclusive locks) whose legal schedules are all D-serializable schedules of τ. □

A similar result holds for the set of schedules of a transaction system with Live-DSR subschedules. Unfortunately, the locking policy of Theorem 6 is not practical: If we want to get all DSR schedules it is much simpler to have the scheduler keep track of the digraph D of the schedule and make sure that no cycle is created, rather than use locking.

6. CONCLUSIONS

The result of this paper can be seen both in a negative and positive light. On the negative side we showed that the class of schedules between PSV-correct and DSR is forbidden territory to PSV-safe locking policies.

On the positive side, we showed that it is easy to test whether a set of transactions can be left to run alone without any locking or intervention from the scheduler while ensuring PSV correctness. Also we saw that if concurrency is controlled by locking, then a complex (NP-complete) criterion (PSV or SFS) can be replaced by a simple (polynomial) criterion (DSR or Live-DSR). This fact is important in designing safe locking policies and in proving their correctness.

Several particular safe locking policies have been proposed in the literature [*E*, *LW*, *SK*, *KS*, *YPK*]; in [*YPK*, *Y1*] a more general approach is taken showing that all safe policies in a natural class of policies - one that includes these particular policies - are instances of a single simple structured policy for different choices of the underlying structure of the database. Although the safety of a locking policy is an NP-complete property [*YPK*] even in a very simple model, it becomes polynomial for the natural class of policies that we mentioned. An important factor there is that only exclusive (read-write) and shared (read-only) locks on entities are considered, a model in which DSR and PSV become identical [*SLR*, *P1*, *U*]. Several other modes of locking are in use or have been proposed, e.g. *update* (write-only), *intention* [*G*, *U*]. Our main result implies that for these modes too (as well as any other) PSV can be replaced by DSR. We hope that this will allow us to treat policies with such modes in a more systematic way, similar to the one in [*YPK*, *Y1*].

REFERENCES

[*BCG*] P. A. Bernstein, M. Casanova, and N. Goodman, "Comments on 'Process Synchronization in Database Systems' ", ACM Trans. on Database Sys., 4(4), (1979).

[*BS*] P. A. Bernstein and D. W. Shipman, "The Correctness of Concurrency Control Mechanisms in a System for Distributed Databases (SDD-1)", ACM Trans. Database System, 5(1), 52-68, (1980).

[*BSW*] P. A. Bernstein, D. W. Shipman, and S. W. Wong, "Formal Aspects of Serializability in Database Concurrency Control", IEEE Trans. Soft. Eng., SE-5(3), 203-216, (1979).

[E] 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. Ass. Comp. Mach., 19(11), 624-633, (1976).

[G] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Database", IBM Res. Rep. RJL654, (1975).

[KS] Z. Kedem, and A. Silberschatz, "Controlling Concurrency using Locking Protocols", 20th Annual Symp. on Found. Comp. Sci. , 274-285, (1979).

[LW] Y. E. Lien and P. J. Weinberger, "Consistency, Concurrency and Crash Recovery", Proc. ACM-SIGMOD, 9-14, (1978).

[P1] C. H. Papadimitriou, "The Serializability of Concurrent Database Updates", J. Ass. Comp. Mach., 26(4), 631-653, (1979).

[P2] C. H. Papadimitriou, "On the Power of Locking", to appear in Proc. ACM-SIGMOD, (1981).

[PBR] C. H. Papadimitriou, P. A. Bernstein, and J. R. Rothnie, "Some Computational Problems Related to Database Concurrency Control", Proc. Conf. Theor. Comp. Sci., Waterloo, 275-282, (1977).

[SK] A. Silberschatz, and Z. Kedem, "Consistency in Hierarchical Database Systems", J. Ass. Comp. Mach., 27(1), 72-80, (1980).

[S] G. Schlageter, "Process Synchronization in Database Systems", ACM Trans. Database Sys., 3(3), 248-271, (1978).

[SLR] R. E. Stearns, P. M. Lewis, and D. J. Rosenkrantz, "Concurrency Control for Database Systems", Proc. 17th Annual Symp. on Found. Comp. Sci., 19-32, (1976).

[U] J. D. Ullman, Principles of Database Systems, Computer Science Press, 1979.

[Y1] M. Yannakakis, "A Theory of Safe Locking Policies in Database Systems", submitted, (1980).

[Y2] M. Yannakakis, "The Various Notions of Correctness in Database Concurrency Control", in preparation.

[YPK] M. Yannakakis, C. H. Papadimitriou, and H. T. Kung, "Locking Policies: Safety and Freedom from Deadlock", Proc. 20th Annual Symp. on Found. Comp. Sci., 283-287, (1979).