RJ 1487 (#22786)
December 30, 1974
Computer Science

ON THE NOTIONS OF CONSISTENCY AND PREDICATE LOCKS IN A DATA BASE SYSTEM

K. P. Eswaran
J. N. Gray
R. A. Lorie
I. L. Traiger

IBM Research Laboratory
San Jose, California 95193

ABSTRACT: Locking protocols in a shared data base are substantially different from those common to operating systems. In data base systems, users access shared data under the assumption that the data satisfies certain consistency constraints. This paper formally defines the concepts of transaction, consistency and schedule and shows that consistency implies that a transaction cannot request new locks after releasing a lock. Then it is argued that a transaction needs to lock a logical rather than a physical subset of the data base. These subsets are specified by predicates. An implementation of predicate locks which satisfies the consistency condition is suggested. This paper is not concerned with resource scheduling, preemption, backup or deadlock (although these topics are mentioned).

1. Introduction

Much work has been done on the problem of sharing resources in an operating system. However, there are basic differences between the requirements of an integrated data base system and those of an operating system. The underlying mechanism, locking, is the same so the issues of scheduling, preemption, and deadlock persist; but the unit of locking is fundamentally different. While fairly static locking schemes are acceptable in a conventional operating system, a particular data base transaction may lock an arbitrary logical subset of the data base.

Transactions lock such subsets in order to obtain a consistent view of the system state. Associated with the data base are a large number of semantic constraints. Transactions expect the state to satisfy these constraints and in turn are not allowed to leave the data base in an inconsistent state. The notion of consistency is a novel aspect of the transaction model presented here. Yet the whole purpose of locking is to insure that each transaction sees a consistent image of the state and so the formulation of this notion is essential to a realistic model of the locking problem.

The paper begins in general terms introducing the concepts of entity, action, transaction, schedule, consistency, locking, deadlock, and preemption. The discussion of this section is applicable to data base systems and to more conventional environments such as operating systems. The principle result is that consistency implies that a transaction must

be constructed to have a growing and a shrinking phase. During the growing phase it can request new locks. However, once a lock has been released, the transaction cannot request a new one.

After this general discussion, a second section considers the peculiarities of locking in a data base system. A phenomenon called phantoms seems to imply that one must lock logical subsets of the data base rather than locking individual records present in the data base. An implementation of logical locks satisfying the requirements of consistency is then proposed. For definiteness, this section is couched in terms of a relational model of data.

2. General Properties of Locking

For simplicity we first consider a system with a fixed set of named resources called entities. Each entity has a name and a value. A novel aspect of the model is that we recognize that there is a set of assertions about the system state. Examples of such assertions are:

- "A" is equal to "P"
- "C" is the count of the free cells in "D"
- "E" is an index for "F"

Most such assertions are never explicitly stated in designing or using a system and yet all programs and users depend on the correctness of these assertions whenever they deal with the system state.

The assertions above are quite simple; however, in practice assertions become extremely complex. A complete set of assertions about a system

would no doubt be as large as the system itself. In practice there is little reason for explicitly enumerating all such assertions but for the purposes of this discussion we presume that a set of assertions, hereafter called consistency constraints, is explicitly defined and we say that the state is consistent if the contents of the entities of the state satisfy all the consistency constraints.

The system state is not static. It is continually undergoing changes due to actions performed by processes on the entities. These modifications usually break neatly into independent sequences of actions called transactions. In this paper it is assumed that all transactions, when executed alone, transform the system state from a consistent state to a new consistent state; that is transactions preserve consistency. One might think that consistency could be enforced at each action but this is not true. Transactions may need to temporarily violate the consistency of the system state while modifying it. For example, in moving money from one bank account to another there will be an instant during which one account has been debited and the second not yet credited. This violates a consistency constraint that the number of dollars in the system is constant. To take a more abstract and complete example consider the two transactions T1 and T2 of Figure 1:

```
T1:  A ← A + 100          T2:  A ← A * 2
     B ← B + 100               B ← B * 2
```

Figure 1. Two Transactions.

Suppose that the only assertion about the system state is that A = B. Although when considered alone, both T1 and T2 conserve consistency they have the following properties:

(1.a) temporary inconsistency: After the first step of T1 or T2, A ≠ B and so the state is inconsistent.

(1.b) conflict: If transaction T2 is scheduled to run between the first and second steps of T1 then the end result is A ≠ B which is an inconsistent state.

The fact that T1 run after T2 may not produce the same result as T2 run after T1 is not an issue of consistency. Transactions are not commutative. We do not require determinism (i.e. all schedules produce the same state); we require only that all schedules preserve consistency. This is a major departure from most previous work on concurrency.

The problem of temporary inconsistency is inherent and implies that enforcement of some consistency assertions cannot be done before the end of a transaction. Conflict on the other hand is not inherent and is undesirable. Yet another desirable property is reproducibility. Even if there are no consistency constraints it is desirable in explaining the operation of the system to be able to say that transactions always appear to be run in some sequential order.

If transactions are run one after another with no concurrency then the problem of conflict never arises and reproducibility is guaranteed. Each transaction starts in a consistent state and, since transactions

preserve consistency, each transaction ends in a consistent state. Any inconsistencies seen by an in-progress transaction are due to changes it has made to the state. If transactions were instantaneous, there would be no penalty for a serial schedule for transactions. However, transactions are not instantaneous and substantial performance gains can be attained by running several transactions in parallel.

In most cases a particular transaction depends only on a small part of the system state. Therefore, one technique for assuring consistency is to partition entities into disjoint classes, such that each consistency constraint falls within a single class. One can then schedule transactions concurrently only if they use distinct classes of entities. Transactions using common parts of the state can still be scheduled serially. If such a policy is adopted then each transaction will see a consistent version of the state. Unfortunately, it is usually impossible to examine a transaction and decide exactly which subset of the state it will use. For this reason the "partition" scheme described above is abandoned in favor of a more flexible scheme where individual entities are locked dynamically. In this system, transactions lock entities for two reasons: they want to prevent conflict with other transactions (i.e. lock out changes made by other transactions) and they may want to temporarily suspend consistency assertions on the locked entities.

For simplicity, this section ignores the distinction between shared and exclusive access to an entity. It assumes that each action (other

than lock and unlock) modifies the entity. The generalization of this section to the case of shared access is straightforward.

If transaction T_1 attempts to lock entity e_1 which is already locked by transaction T_2 then either T_1 must wait for T_2 to unlock e_1 or T_1 must preempt e_1 from T_2. If T_1 waits and then T_2 attempts to lock an entity e_2 locked by T_1 then T_2 must wait or preempt. If both T_1 and T_2 wait, then deadlock arises. The question of when to wait and when to preempt is not the subject of this paper. The paper by Chamberlin, Boyce and Traiger [1] presents a scheme for deciding which transaction to preempt. When a resource is preempted, the preempted transaction must be backed up.

Unlike operating systems where (task) backup is quite uncommon, data base systems usually maintain a log of all changes made by each transaction. This log forms an audit trail as well as being used for backup. Backup arises not only from deadlock - preemption but also from protection violations, hardware errors or human error. One backup procedure for a transaction T is to undo all of its updates as recorded in the log. Then all entities locked by T may be unlocked and T may be reset to its initial state. As Davies and Bjork [2,3] point out, this procedure may not work correctly after T has unlocked (committed) any entities which it has modified. This implies that (update) locks should be held to the end of a transaction.

There is a second reason for wanting transactions to unlock entities as late as possible, namely consistency. As pointed out earlier, each transaction wants to see a consistent view of the system state. In order for locks to assure this, a transaction must not request a new lock after releasing some lock. To state and prove this result we must proceed much more formally.

Two actions lock and unlock are introduced. A transaction is said to be well formed if

(2.a) It locks an entity before otherwise acting on it.

and (2.b) It ends with no entities locked.

Note that a transaction may lock and unlock the same entity several times.

More formally, a transaction is a sequence (see footnote): $T = ((T, a_i, e_i))_{i=1}^{n}$ of n steps where T is the transaction name, a_i is the action at step i and e_i is the entity acted upon at step i. A transaction has locked entity e through step i if

(3.a) for some $j \leq i$, a_j = lock and $e_j = e$,

and (3.b) there is no k, $j < k < i$, such that a_k = unlock and $e_k = e$.

A transaction T is well formed if

(2.a') at each step i = 1, ..., n, T has locked e_i through step i,

and (2.b') at step n, only e_n is still locked by T and a_n = unlock.

Figure 2 shows two well formed versions of transaction T_1 from Figure 1.

T11:

T11	LOCK		A
T11	A + 100	→	A
T11	UNLOCK		A
T11	LOCK		B
T11	B + 100	→	B
T11	UNLOCK		B

T12:

T12	LOCK		A
T12	A + 100	→	A
T12	LOCK		B
T12	UNLOCK		A
T12	B + 100	→	B
T12	UNLOCK		B

Figure 2. Two well formed versions of transaction T1 of Figure 1.

Any sequence obtained by collating the actions of transactions $T_1,\cdots,T_n$ is called a schedule for $T_1,\cdots,T_n$. If the schedule takes actions from one transaction at a time it is called a serial schedule. A schedule for a set of transactions $T_1,\cdots,T_n$ is any sequence $S = ((T_i, a_i, e_i))_{i=1}^{m}$ such that for each $j=1,\cdots,n$

$$T_j = ((T_i, a_i, e_i) \in S | T_i = T_j)_{i=1}^{m}.$$

That is, S contains T_j and preserves its sequence of actions. Also, the length of S is the sum of the lengths of the transactions $T_1,\cdots,T_n$ (i.e. S contains only elements of $T_1,\cdots,T_n$). A schedule S is <u>serial</u> if for some permutation π, $S = T_{\pi(1)}\ T_{\pi(2)}\ \cdots\ T_{\pi(n)}$ (i.e. S is the concatenation of the transactions). Figure 3 gives three examples of schedules for a set of three transactions.

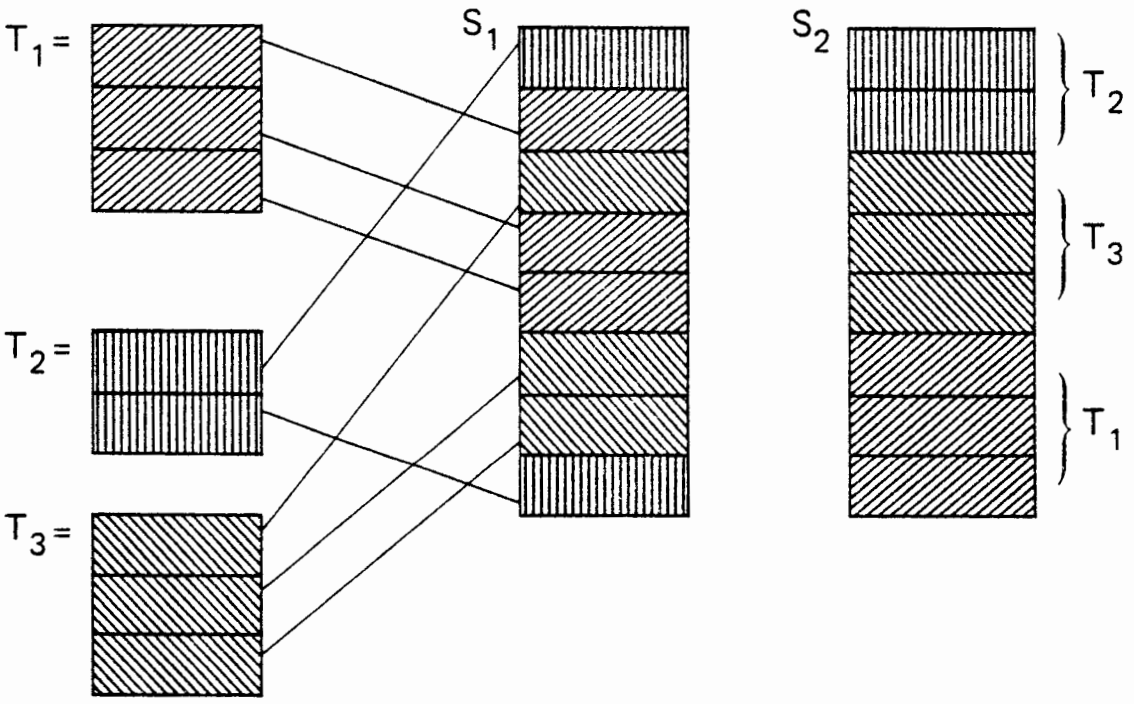

Figure 3. Schedules for three transactions T_1, T_2, T_3. S_2 is a serial schedule. Each small rectangle represents a transaction step.

Non-serial schedules run the risk of giving a transaction an inconsistent view of the state. So we are particularly interested in those schedules which are "equivalent" to serial schedules. The equivalence between schedules hinges on the dependency relation of a schedule.

The dependency relation induced by schedule S is a ternary relation on T×E×T (where T is the set of all transaction names in S and E is the set of all entities) defined by $(T_1,e,T_2) \in DEP(S)$ iff for some i<j:

(4.a) $S = (\cdots,(T_1,a_i,e),\cdots,(T_2,a_j,e),\cdots)$

(4.b) There is no k such that $i<k<j$ and $e_k = e$.

Informally, if (T_1, e, T_2) is in DEP(S) then entity e is an output of T_1 and an input of T_2 and T_1 gives e to T_2. Again, we are assuming that each action on an entity modifies the entity. If one distinguishes "read-share" actions, then the dependency relation must be modified so that entities which are only read by a transaction are not recorded as outputs of the transaction (i.e. adjoin the clause "and a_i or a_j is an update action" to (4.a) and adjoin the clause "and a_k is an update action" to (4.b)).

Two schedules, S_1 and S_2 are equivalent if $DEP(S_1) = DEP(S_2)$ and a schedule S_1 is consistent if it has an equivalent serial schedule. Figure 4 illustrates these definitions. It shows three schedules, where S_1 is consistent, S_2 is not consistent and S_3 is serial (therefore consistent).

S_1:				$DEP(S_1)$ =
	T_1	A + 100 →	A	$\{(T_1, A, T_2), (T_1, B, T_2)\}$
	T_2	A * 2 →	A	
	T_1	B + 100 →	B	
	T_2	B * 2 →	B	
S_2	T_1	A + 100 →	A	$DEP(S_2)$ =
	T_2	A * 2 →	A	$\{(T_1, A, T_2), (T_2, B, T_1)\}$
	T_2	B * 2 →	B	
	T_1	B + 100 →	B	
S_3	T_1	A + 100 →	A	$DEP(S_3) = DEP(S_1)$
	T_1	B + 100 →	B	
	T_2	A * 2 →	A	
	T_2	B * 2 →	B	

Figure 4. Three schedules for T_1, T_2 of Figure 1. S_1 is equivalent to serial schedule S_3 and hence is consistent. S_2 is inconsistent.

It is very easy to explain and to reproduce the effect of a serial schedule. The user thinks of a complete transaction as being an "atomic" transformation of the state just as the scheduler thinks each action is an atomic transformation of the state. He sees all the changes made by transactions "before" his transaction starts and none of the changes of transactions "after" his transaction completes (i.e. he sees a consistent

state). Any non-serial consistent schedule also has these properties. This discussion yields the following important properties of serial schedules:

(5.a) If T_1 and T_2 are any two transactions and e_1 and e_2 are any entities then $(T_1,e_1,T_2) \in DEP(S)$ implies $(T_2,e_2,T_1) \notin DEP(S)$.

More generally:

(5.b) The binary relation $<$ on the set of transactions is defined by: $T_1<T_2$ if and only if $(T_1,e,T_2) \in DEP(S)$ for some entity e. Then $<$ is an acyclic relation which may be extended to a total order of the transactions.

Clearly any consistent schedule also has these properties. Conversely, any schedule with property (5.b) is consistent.

We would like to further characterize those non-serial schedules which are consistent. To do this it is necessary to consider the lock and unlock actions of each step. Entity e is said to be <u>locked</u> by transaction T through step k of schedule S if:

(6.a) There is a $j \leq k$ such that $S(j) = (T, lock, e)$.

and (6.b) There is no j', $j<j'<k$ such that $S(j') = (T, unlock, e)$.

Schedule S is <u>legal</u> if for all k, if $S(k) = (T, a, e)$ and e is locked by T through step k, then

(7) e is not locked by any other transaction through step k.

Legal schedules observe the lock protocol that a transaction attempting to lock an already-locked entity must wait. A schedule gives a history of how transactions were processed. As the processing is being done, we

imagine a scheduler choosing a transaction step at each instant. This scheduler allows lock actions on free entities but never chooses a lock action on an already-locked entity. Such a scheduler only produces legal schedules since it never chooses to run a lock step on an already-locked entity.

The example schedule of Figure 5 shows that not every legal schedule is consistent. It is very important to know how transactions must be constructed so that any legal schedule is consistent.

Clearly, if legality is to insure consistency then it is necessary that each transaction lock each entity before otherwise acting on it and that the transaction ultimately unlock each such locked entity. More formally, using the definition of well formed transactions (2.a'), (2.b'):

(8.a) Consistency requires that transactions be well formed. To prove this consider any transaction $T_1 = (T_1, a_i, e_i)_{i=1}^{n}$ which is not well formed. Then for some step k, T_1 does not have e_k locked through step k. Consider the (well formed) transaction $T_2 = (T_2, \text{lock}, e_k), (T_2, \text{unlock}, e_k)$, and the schedule $S = (T_1(i) \mid i=1,\cdots,k-1), T_2(1), T_1(k), T_2(2), (T_1(i) \mid i=k+1,\cdots,n)$. Since (T_1, e_k, T_2) and (T_2, e_k, T_1) are both in DEP(S), S is not equivalent to any serial schedule (by property 5.a). So S is not consistent.

A less obvious fact is that consistency requires that a transaction be divided into a growing and a shrinking phase. During the growing phase the transaction is allowed to request locks. The beginning of the

shrinking phase is signaled by the first unlock action. After the first unlock, a transaction cannot issue a lock action on any entity. More formally, transaction $T = ((T, a_i, e_i))_{i=1}^{n}$ is <u>two phase</u> if for some $j \leq n$,

$$i < j \quad \text{implies} \quad a_i \neq \text{unlock},$$

$$i = j \quad \text{implies} \quad a_i = \text{unlock},$$

$$i > j \quad \text{implies} \quad a_i \neq \text{lock}.$$

Steps $1, \cdots, j-1$ are called the <u>growing phase</u> and steps $j, \cdots, n$ are the <u>shrinking phase</u> of T.

Transaction T11 of Figure 2 is not two phase since it locks B after releasing A. Transaction T12 of Figure 2 is two phase. To see that T11 may see an inconsistent state consider the legal schedule S shown in Figure 5. In the schedule S, T12 sees A from T11 and T11 sees B from T12. So S is not equivalent to any serial schedule and hence S is inconsistent. In general

(8.b) Consistency requires that transactions be two phase.

S

T11	LOCK	A	
T11	UPDATE	A	
T11	UNLOCK	A	
T12	LOCK	A	T11 gives A to T12
T12	LOCK	B	
T12	UPDATE	A	
T12	UPDATE	B	
T12	UNLOCK	B	
T12	UNLOCK	A	T12 gives B to T11
T11	LOCK	B	
T11	UPDATE	B	
T11	UNLOCK	B	

DEP(S) = {(T11, A, T12),(T12, B, T11)}

Figure 5. A schedule for transactions T11 and T12 which is legal but not consistent because T11 is not two phase.

Conversely,

(8.c) If each transaction in the set of transactions $T = \{T_1,\cdots,T_n\}$ is well-formed and two-phase then any legal schedule for T is consistent.

A sketch of the proof for this is fairly simple. Let S be any schedule for T. Define the binary relation '<' on T by $T_i < T_j$ iff $(T_i, e, T_j) \in DEP(S)$ for some entity e. One can prove a lemma that < may be extended to a total order << on T as follows.

First define the integer $SHRINK(T_i)$ for each transaction T_i to be the least integer j such that T_i unlocks some entity at step j of S:

$$SHRINK(T_i) = \min \{j \mid S(j) = (T_i, \text{unlock}, e) \text{ for some entity } e\}.$$

If each transaction T_i is non-null then $SHRINK(T_i)$ is well defined because each T_i is well formed.

Now observe that for any transactions T_1 and T_2 and entity e, if $(T_1, e, T_2) \in DEP(S)$ then $SHRINK(T_1)$ is less than $SHRINK(T_2)$. For if $(T_1, e, T_2) \in DEP(S)$ then by definition of DEP(S) there are integers i and j such that

$$S = (\cdots, (T_1, a_i, e), \cdots, (T_2, a_j, e), \cdots)$$

and so that for any integer k between i and j $e_k \neq e$. Since S is legal, e must be locked only by T_1 through step i of S and e must be locked only by T_2 through step j of S. So a_i = unlock and a_j = lock. This immediately implies that $SHRINK(T_1)$ is less than or equal to i. Since T_2 is two phase, then no unlock by T_2 preceeds step j of S so $SHRINK(T_2)$ is greater than j.

Thus we have shown that if $T_1 < T_2$ then SHRINK(T_1) is less than SHRINK(T_2). This implies property (5.b) and hence < can be extended to a total order << on T.

Assume without loss of generality that $T_1 << T_2 << \cdots << T_n$. Induct on n to show that S is equivalent to the serial schedule $T_1 \cdots, T_n$. If n=1 the result is trivial. The induction step follows in two steps.

First show that S is equivalent to the schedule

$$S' = T_1 \ ((T_i, a_i, e_i) \in S | T_i \neq T_1)_{i=1}^{m}.$$

Then note that by hypothesis

$$((T_i, a_i, e_i) \in S | T_i \neq T_1)_{i=1}^{m} \text{ is equivalent to } T_2, \cdots, T_n.$$

So S' is equivalent to $T_1, T_2, \cdots, T_n$. But $T_1, \cdots, T_n$ is a serial schedule so S is equivalent to a serial schedule and is consistent. Figure 6 gives a graphic illustration of the construction of a serial schedule from S.

To summarize then

(8.d) A necessary and sufficient condition for all legal schedules to be consistent is that each transaction be well-formed and two-phase.

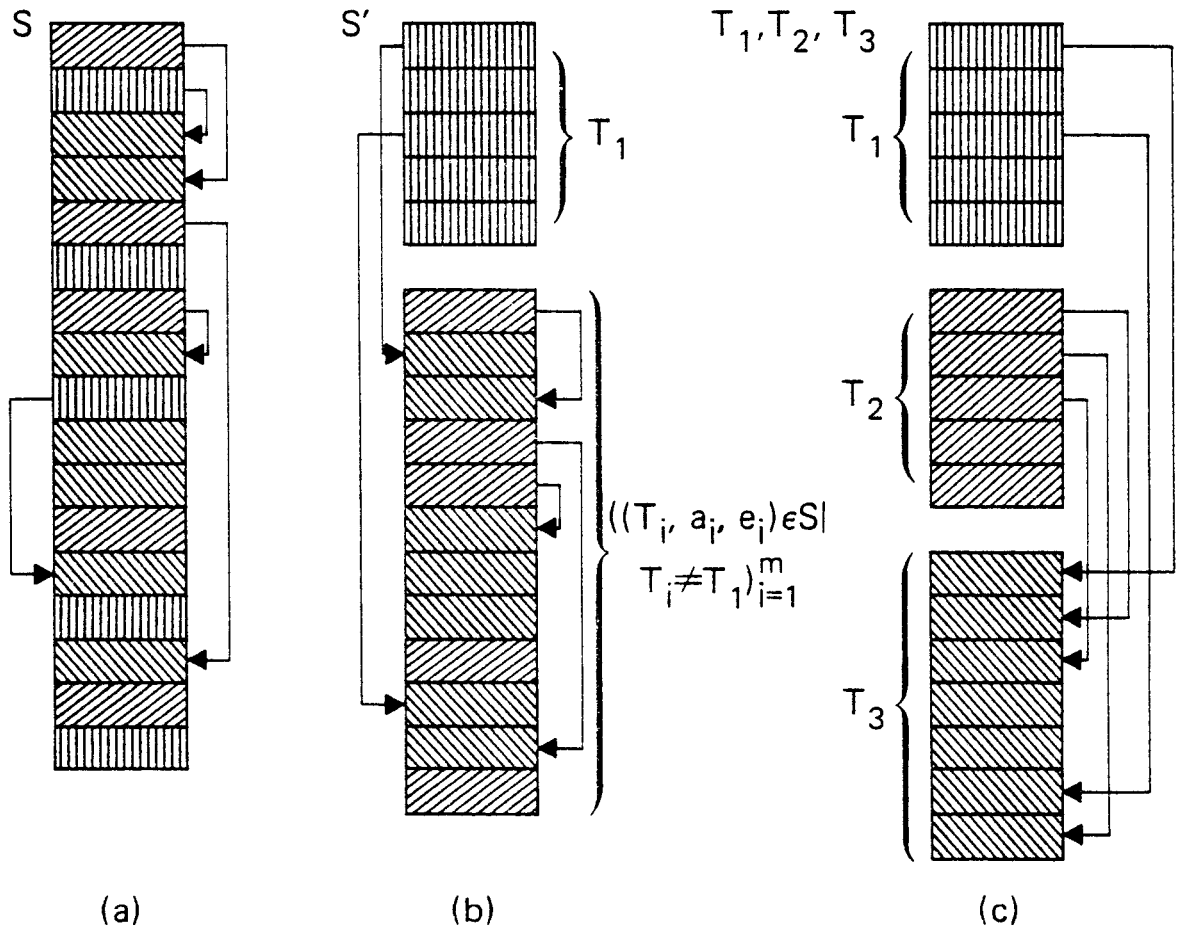

Figure 6. A graphic illustration of the construction of a serial schedule. The arrows show the dependencies of S. $T_1 << T_2 << T_3$ and so S' has the same dependencies as S. The induction hypothesis applies to S' to give T_1, T_2, T_3.

3. Predicate Locks

Section 2 introduced the notions of consistency and of locking and it explored the locking protocols required by consistency. The discussion was quite general and applies to any system which supports the concepts of transaction and shared entity. Next we consider locking in a data base environment. Aside from the problem of scale (billions of entities rather than hundreds or thousands), there are substantial differences in the unit of locking. These differences stem from associative addressing of entities by transactions in a data base environment. It is not uncommon for a transaction to want to lock the set of all entities with a certain value (i.e. "key" addressing). Updating a seemingly unrelated entity may add it to such a set, creating the problem of "phantom" records. This section explains this problem and proposes a spectrum of solutions.

For definiteness we adopt the relational model of data (Codd [4]). The data base consists of a collection of relations, $R_1, R_2, \cdots, R_n$. Each relation can be thought of as a table or flat file. It is a homogeneous set of distinct tuples (records). Each tuple consists of a fixed number of fields. The columns of the relation are called domains. Each domain has a name. Figure 7 shows an example of such a data base.

ACCOUNTS		
Location	Number	Balance
NAPA	32123	1050
ST HELENA	36592	506
NAPA	5320	287

ASSETS	
Location	Total
NAPA	1337
ST HELENA	506

Assertions:
1) Account numbers are unique.
2) The sum of balances of accounts at a location is equal to the total assets at that location.

Figure 7. The sample data base.

One approach would be to lock whole relations or domains whenever any member of the relation or domain is referenced. However, since there are many more tuples than relations or domains this will not produce much concurrency. For example, two transactions making deposits in different accounts could not run concurrently if they were required to lock whole relations.

This suggests that locks should apply to as small a unit as possible so that transactions do not lock information they do not need. Therefore the natural unit of locking is the field or tuple of a relation. However, a tuple is not an entity in the sense of section 2, since it has no name

which is separate from its value. This may seem odd at first, but it stems from the fact that tuples are referenced by value rather than by the address that their storage occupies.

To illustrate this point consider the example of a transaction T_1, on the data base of Figure 7. The transaction checks the assertion that the sum of Napa account balances is equal to the sum of Napa assets by:

(9.1) Associatively addressing the ACCOUNTS relation, locking any accounts located in Napa.

(9.2) Summing the balances in the locked accounts.

(9.3) Locking the Napa tuple in ASSETS and comparing its value with the computed sum.

(9.4) Releasing all locks.

If a second transaction T_2 inserts a new tuple in ACCOUNTS with Location = Napa and adds its balance to the Napa assets and if T_2 is scheduled between steps (9.2) and (9.3) of T_1 then T_1 will see an inconsistent state: T_1 will see the balance of the new account reflected in the ASSETS but will not have seen the account in the ACCOUNTS relation. A similar problem arises if T_2 merely transferred an account from St. Helena to Napa.

A still more elementary example is the test for the existence of a tuple in a relation. If the tuple exists it is to be locked to insure that no other transaction will delete it before the first transaction terminates. If the tuple does not exist, "it" should be locked to insure that no other transaction will create such a tuple before the first transaction terminates. In this case the "non-existence" of the tuple is

being locked. Such non-existent tuples are called phantoms. Inspection of the earlier example shows that transaction T_1 should lock not only all existing Napa accounts but also all phantom Napa accounts.

As argued in the previous section, consistency requires that a transaction lock all tuples examined, both real and phantom (i.e. it be well formed). The set of all possible Napa accounts is the Cartesian product: {Napa} × INTEGERS × INTEGERS. This set is infinite so there is little hope of locking each individual phantom. Rather it seems natural to lock the set of tuples and phantoms satisfying the predicate: Location = Napa. More generally, if P is a predicate on tuples t of relation R then P defines the set S where $t \in S$ iff P(t). Transactions will be allowed to lock any subset of a relation specified by such a predicate. We only require that the truth or falsity of P depend only on t.

If such predicates are used as the unit of locking then a list of locks becomes a (much smaller) list of sets identified by their predicates. Locking the entire relation is achieved by using the predicate 'TRUE' while locking the tuple (NAPA, 32123, 1050) is achieved by the predicate t = (NAPA, 32123, 1050). However, one cannot directly apply the formulation of locking and consistency in the previous section, because entities were assumed to be uniquely named objects. In this section we extend the results on scheduling and consistency to apply to locks on possibly overlapping sets of tuples.

First of all, if predicates are arbitrarily complex there is little hope of deciding whether two distinct predicates define overlapping sets of tuples (and hence whether they conflict as locks). In fact the problem is recursively unsolvable (Kleene [5]), so it is not clear how to make predicate locks "work." A method for scheduling of predicate locks is introduced first by example and then more abstractly.

In the sample data base of Figure 7 suppose that transaction T_1 is interested in all tuples in ACCOUNTS for which Location = Napa. A transaction T_2 starts during the processing of T_1. T_2 is interested in all tuples in ACCOUNTS with Location = Sonoma. When T_1 declares its intent to access Napa accounts by executing the action:

T_1 LOCK ACCOUNTS: Location = Napa,

this predicate lock is associated with T_1 and with the ACCOUNTS relation. Later when T_2 declares its intent to access Sonoma accounts by executing the action:

T_2 LOCK ACCOUNTS : Location = Sonoma

this predicate lock is also associated with the ACCOUNTS relation. Before T_2 can be granted access to the Sonoma accounts, the lock controller must check that T_2's lock does not conflict with locks held by other transactions. In the case above, the controller must decide that the predicates Location = Napa and Location = Sonoma are mutually exclusive. In general, the controller must compare the requested predicate lock against the outstanding predicate locks of other transactions on this relation. If two such predicates are mutually satisfiable (i.e. have an

existing or phantom tuple in common) then there is conflict and the request must wait or preempt.

That is more or less how predicate locks work. It does not explain how sharing works and finesses the fact that predicate satisfiability is recursively unsolvable. In order to give a more complete explanation of how predicate locks "work", it is necessary to define how an action is allowed or prohibited by a lock and how two locks may conflict. A particular action on a single tuple may be denoted by:

$$(R, t, \{(f_i , a_i)\}_{i=1}^{n})$$

meaning that field f_i of tuple t of relation R is accessed in mode a_i. Two modes are distinguished here:

a_i = read allows sharing with other readers,

while a_i = write requires an exclusive lock on f_i (update, insert, and delete are all examples of write access).

The action reads those fields f_i of tuple t such that a_i=read and it writes those fields f_i of tuple t such that a_i=write. Fields which are not mentioned are not acted upon.

Reading the balance of account number 32123 would be an action:

(ACCOUNTS, (Napa,32123,1050), {(Number,read),(Balance,read)})

Note that this action does not read the location field. An update of the balance by 50 dollars would be one action but involves two tuples, first

(ACCOUNTS, (Napa,32123,1050), {(Number,read),(Balance, write)})

and also

(ACCOUNTS, (Napa,32123,1100), {(Number,read),(Balance,write)})

because both tuples are written by the atomic update operation (one is "deleted" and the other "inserted"). Further, consistency requires that the Napa ASSETS tuple be updated by 50 dollars.

In the model of actions described above, the action specifies a tuple by providing the values of all fields of the tuple. Although this is formally correct, the examples above show that it is inappropriate for the context at hand. The first example wants to read the balance of account number 32123 and cares nothing about the location of the account. Yet the model requires that the action specify both the balance and location of the account as well as the account number. Similarly the second transaction wants to read the balance and location of account number 32123 and then add 50 dollars to the balance of the account and to the assets of the account's location.

If one considers the problem of reading the Napa tuple of ASSETS without a-priori knowing its current balance the problem and its solution

becomes quite clear. The concept of action must be generalized to the concept of access which acts on all tuples satisfying a given predicate. This notion is consistent with the idea of associative addressing which returns the set of all tuples with designated values in given fields. To access account number 32123 reading the balance, one specifies the access:

(ACCOUNTS, Number = 32123, {(Number, read), (Balance, write)})

which returns either a single tuple or no tuples since account numbers are unique. An access which reads the location of and updates the balance of account number 32123 would be denoted by:

(ACCOUNTS, Number = 32123, {(Location, read),
(Number, read),
(Balance, write)}).

Consistency requires that such an access be followed by an access

(ASSETS, Location = 'Napa', {(Location, read),
(Balance, write)})

since we require that the assets be the sum of the balances at each location.

An access to find the numbers of all Napa accounts would return a set of tuples and would be denoted by:

(ACCOUNTS, Location = 'Napa', {(Location, read),
(Number, read)}).

To proceed more formally we need the following definitions. If the relation R is drawn from the Cartesian product of sets $S_1, S_2, \cdots S_n$, ($R \subseteq \underset{i=1}{\overset{n}{\times}} S_i$) then any predicate P defined on all tuples $(s_1, \cdots, s_n) \in \underset{i=1}{\overset{n}{\times}} S_i$ is an admissible predicate for R. We ask that P be an effective test: given a tuple t, P(t) = TRUE or P(t) = FALSE.

A particular access on relation R is denoted by:

$$(R, P, \{(f_i, a_i)\}_{i=1}^{n})$$

where P is an admissible predicate. Such an access is equivalent to the (possibly infinite) set of actions

$(R, t, \{(f_i, a_i)\}_{i=1}^{n})$ where P(t) = TRUE, and where t ranges over the Cartesian product underlying R.

In particular it reads all tuple-field pairs (t, f_i) read by such actions and writes all tuple-field pairs written by such actions. A predicate lock on relation R is denoted by:

$$(R, P, \{(f_i, a_i)\}_{i=1}^{n}),$$

where P is an admissible predicate for R and each f_i is a field locked for access mode a_i. It is further required that if the value of P depends upon the value of field f then $f = f_i$ for some $i = 1,\cdots,n$ (since the predicate "reads" these fields).

An action $(R, t, \{(f_i, a_i)\}_{i=1}^{n})$ is said to <u>satisfy</u> predicate lock $(R', P', \{(f_i', a_i')\}_{i=1}^{m})$ if

(10.a) $R = R'$

(10.b) $P'(t) = TRUE$

(10.c) for each $i = 1,\cdots,n$, there is a $j : (f_i, a_i) = (f_j', a_j')$ or ($f_i = f_j'$ and a_i = read and a_j' = write).

In the second clause of (10.c) we are assuming that write access implies read and write access.

The action $(R, t, \{(f_i, a_i)\}_{i=1}^{n})$ <u>conflicts</u> with predicate lock $(R', P', \{(f_i', a_i')\}_{i=1}^{m})$ if

(11.a) $R = R'$

(11.b) $P'(t) = TRUE$

(11.c) for some i, j:

$f_i = f_j'$ and either a_i = write or a_j' = write.

To give an example, the predicate lock:

L = (ACCOUNTS, Location = Napa, {(Location, read), (Balance, read)})

is satisfied by the action

(ACCOUNTS, (Napa, 3213, 1050), {(Location, read), (Balance, read)})

and is satisfied and conflicts with the action:

(ACCOUNTS, (Napa, 3213, 1050), {(Location, write)})

Satisfiability and conflict are defined analogously for accesses. Access A = (R, P, $\{(f_i, a_i)\}_{i=1}^{n}$) satisfies predicate lock L if and only if for each tuple t in the Cartesian product underlying R, if P(t) is true then action (R, t, $\{(f_i, a_i)\}_{i=1}^{n}$) satisfies L. Access A conflicts with L if for some tuple t in the Cartesian product underlying R, P(t) is true and action (R, t, $\{(f_i, a_i)\}_{i=1}^{n}$) conflicts with L.

To give an example, the access which moves account #23175 from Napa to Sonoma would be denoted:

(ACCOUNTS, (Location = 'Napa' $\vee$ Location = 'Sonoma')
$\wedge$ Number = 23175,
{(Location, write), (Number, read)}).

This access would require that the transaction have a lock on the ACCOUNTS relation of the form:

(ACCOUNTS, P, $\{(f_i, a_i)\}_{i=1}^{n}$),

where Location and Number are included among the f_i, write access is allowed to Location and read access is allowed to Number. Further the predicate P must be satisfied by the tuples:

(Napa, 23175,*)

and

(Sonoma, 23175, *).

That is, the lock predicate P must cover both the old and new values.

Note that we require an access to be covered by a single predicate lock. If one holds two locks, one for Napa and another for Sonoma, then the access would not satisfy either one and so would not be allowed. It is possible to relax this restriction so that an access is allowed if it satisfies the union of the locks held by a transaction.

Two predicate locks are said to conflict if there is some action which satisfies one of them and conflicts with the other. That is, if one lock allows an access which is prohibited by the other lock.

Given these definitions, the notions of the previous section generalize as follows. A transaction is a sequence of (transaction name, access) pairs. A transaction is well formed if each access it makes satisfies some predicate lock it holds through that step. A transaction is two phase if it does not request predicate locks after releasing a predicate lock.

A schedule for a set of transactions is any merging of the composite sequences. The dependency relation is defined by choosing (field, tuple, relation) triples as the entities. Let E be the set of all such entities. The notion of an access reading or writing such entities has already been introduced. If S is a schedule for the set of transactions T, then the dependency set of S is defined to be the set of triples:

$$(T_1, e, T_2) \in T \times E \times T$$

such that for some integers $i<j$:

(12.1) $S(i) = (T_1, A_1)$ and A_1 reads or writes entity e,

(12.2) $S(j) = (T_2, A_2)$ and A_2 reads or writes entity e and not both A_1 and A_2 simply reads e,

(12.3) for any k between i and j, if $S(k) = (T_3, A_3)$ then A_3 does <u>not</u> write entity e.

Since infinite sets of tuples are involved, satisfiability and conflict for accesses and predicate locks may not be decidable. The introduction of simple predicates later in this section will give a decidable subset of possible access predicates and lock predicates.

To implement arbitrary predicate locks, associate with the data base a table called LOCK which is a binary relation between transactions and predicate locks (see Figure 8).

LOCK	
Transaction	Predicate Lock
T_1	(ACCOUNTS, Location=Napa {(Location, read) (Balance, write)})
T_2	(ACCOUNTS, Balance < 500, {(Number, read), (Balance, read)})

Figure 8. An example of the LOCK table.

The legal lock scheduler functions as follows. Transactions are presumed to be two phase and well formed; the scheduler enforces this rule. Any growing transaction may request any predicate lock. When this happens, the scheduler tries to enter the transaction name and predicate lock into the LOCK table. If the predicate lock does not conflict with any other predicate lock in the table, it may be entered and granted immediately. If the predicate lock does conflict with one or more locks held by other transactions then the requestor must wait for the other locks to be released or he must preempt the locks (or be preempted). As commented earlier, this is a scheduling decision and not the proper topic of this paper. Any transaction may release any predicate lock belonging to it. This deletes the lock from LOCK and marks the transaction as shrinking. If other transactions are waiting for tuples released by this lock then they may be started. Each time a transaction T* makes an action or access A the LOCK table is examined to find the set:

$$YES = \{(T, L) \in LOCK \mid A \text{ satisfies } L \text{ and } T \neq T^*\}$$

YES is a list of all the reasons T* should be allowed to make the access. If YES is empty then T* is not well formed and it should be given an error.

It is clear that the scheduler described above has the properties:

(13.1) All transactions are well formed and two phase.

(13.2) If transaction T locks predicate P on relation R, then for any tuple t in the Cartesian product underlying R such that P(t)=TRUE, no other transaction may insert, delete or modify the locked fields of t until T releases the predicate lock. That is, predicate locks solve the problem of phantoms, thereby providing consistency.

So the scheduler described produces legal schedules and by the results of the previous section, gives each transaction a consistent view of the state of the system.

Thus far we have ignored the details of how the scheduler decides whether two locks conflict. In general this is a recursively unsolvable problem (even if predicates are restricted to using the arithmetic operators +, *, -, ÷ as shown by Presburger [5]). The problem then is to find an interesting class of predicates for which it is easily decidable whether two predicates "overlap". We propose the following simple class of predicates.

A simple predicate is any Boolean combination of atomic predicates. Atomic predicates have the form:

$$\text{<field name>} \left\{ \begin{array}{c} \text{'<'} \\ \text{'='} \\ \text{'}\neq\text{'} \\ \text{'>'} \end{array} \right\} \text{<constant>}$$

where constant is a string or number and field name is the name of some field of the relation. For example

$$((\text{Location} = \text{'Napa'} \vee \text{Location} = \text{'Santa Rosa'}) \wedge$$
$$((\text{Balance} < 200) \wedge (\text{Balance} > 10))$$

is a simple predicate with four atomic predicates.

Again, Presburger showed a decision procedure for a class of predicates slightly more general than simple predicates (he allowed +, -, <, =, ≠, >, mod and allowed any boolean combination of these operators and operands on integers.) However his decision procedure is much more complicated than the procedure for this simple set of predicates.

To decide whether two predicate locks L and L' conflict is a fairly simple matter. Suppose $L = (R, P, \{(f_i, a_i)\}_{i=1}^{n})$ and that $L' = (R', P', \{(f_i', a_i')\}_{i=1}^{m})$ are two predicate locks. Then

(14.a) if $R \neq R'$ there is no conflict as the locks apply to different relations.

(14.b) if there is no field f such that f_i=f and f'_j=f and either a_i=write or a'_j=write then there is no conflict.

(14.c) otherwise there will be no conflict only if there is no tuple t such that P∧P'(t) is TRUE.

Similarly, to decide whether access $A = (R', P', \{(f'_i, a'_i)\}_{L=1}^{m})$ conflicts with lock L above consists of testing (14.a), (14.b) and (14.c) above for access A. A will satisfy L if it passes the tests

(15.a) $R = R'$, and

(15.b) for each $i = 1,\ldots, m$ there is a j such that $f'_i = f_j$ and a'_i = read or a_j = write, and

(15.c) For any tuple t, if P'(t) is TRUE then P(t) is TRUE (i.e. P' => P or equivalently P' ∧ ~ P is not satisfiable).

Thus the conflict - satisfiability questions for both accesses and locks have been reduced to the question of deciding whether a particular simple predicate is satisfiable. But simple predicates are defined to have an easy decision procedure.

The procedure is to organize P∧P' of case (c) into disjunctive normal form (Kleene [5]) and then for each disjunct see whether it is satisfiable or not. Each such disjunct will be a conjunct of atomic predicates and so this is trivial. Consider the example:

P = (Location = 'Napa' ∨ Location = 'Santa Rosa') ∧
(Balance < 500 ∧ Balance > 10)

P' = Location = 'Napa' ∧ Balance = 700.

Then the disjunctive normal form of P ∧ P' is

Location = 'Napa' ∧ Balance < 500 ∧ Balance > 10 ∧ Balance = 700
∨ Location = 'Santa Rosa' ∧ Location = 'Napa' ∧ Balance < 500
∧ Balance > 10 ∧ Balance = 700

The first disjunct is not satisfied because Balance = 700 contradicts Balance < 500 while the second has the added contradiction that Location = 'Napa' and Location = 'Santa Rosa'. So P ∧ P' is not satisfiable and there is no conflict.

To give an example of conflict, suppose

P = (Location = 'Napa')

and P' = (Balance > 500)

Then P ∧ P' is satisfiable by the tuple (Napa, 0, 501) and so the predicates "overlap" and allow conflict.

In summary then, if only simple predicates are allowed in accesses and predicate locks then predicate locks can be scheduled in the same way ordinary locks are scheduled.

As mentioned before, predicate locks solve the problem of phantom records. When coupled with the results on consistency, predicate locks

can be used to construct consistent legal schedulers. The degenerate form of predicates, locking entire relations with the predicate which is always TRUE or locking a particular tuple by the predicate which is only TRUE for that tuple gives the more conventional forms of locking. If the desired set is not describable by a simple predicate then any 'larger' simple predicate (i.e. a simple predicate which is implied by the desired predicate) will be a suitable predicate for the lock. If only simple predicates are used then predicate locks can be legally scheduled.

There are simple analogs to predicate locks in existing data-base systems. For example in hierarchial systems such as IMS (IBM [7]) it is common to lock a subtree of the hierarchy. This subtree is a logical set of records (i.e. those with a given parent). Similarly, in a network model it is desirable to lock all members of a "set" in the DBTG [6] sense although DBTG lacks such a facility.

Lastly we observe that locking is a very dynamic form of authorization. All the techniques we have described (predicate locks, simple predicates, the YES set,...) apply to the problem of doing value dependent authorization of access to data base records at the granularity of a field.

4. Summary

Section 2 introduced a very simple data model and discussed the notions of transaction, consistency and locking. It argued that consistency requires that transactions be two phase and well formed and conversely

that if all transactions are well formed and two phase then any legal schedule is consistent.

Section 3 was couched in terms of the relational model of data. It described the problems that associative addressing introduces: namely phantom records entering and leaving the set of records locked by a transaction. Predicate locks are proposed as a solution to this problem. To schedule and enforce these locks, predicates are restricted to the class of simple predicates. It is possible to schedule simple predicate locks in the same way "ordinary" locks are scheduled.

6. Acknowledgements

The concept of consistency presented here grew out of discussions with Ray Boyce, Don Chamberlin and Frank King. An earlier draft of the paper was polished by helpful comments from Rudolph Bayer, Paul McJones and Gianfranco Putzolu. We would like to thank Elizabeth Hoover for preparing this manuscript.

7. References

[1] Chamberlin, D. D., Boyce, R. F., Traiger, I. L.; A deadlock-force scheme for resource locking in a data-base environment, Proc. IFIP '74 Congress, North Holland, pp. 340-343 (1974).

[2] Davies, C. T., Recovery semantics for a DB/OC System, NCC, pp. 136-141 (May 1973).

[3] Bjork, L. A., Recovery scenario for a DB/OC System, NCC, pp. 142-146 (May 1973).

[4] Codd, E. F., A relational model for large shared data banks, CACM, Vol. 13, No. 6, pp. 377-387 (June 1970).

[5] Kleene, S. C., Introduction to Metamathematics, Van Nostrand Company, p. 204 (1952).

[6] IBM Information Management System for Virtual Storage (IMS/VS), Conversion and Planning Guide, pp. 38-44, Form No. SH20-9034, IBM, Armonk, New York.

[7] CODASYL, Data base task group report, ACM New York (1971).

FOOTNOTE

The sequence $S=s_1,\cdots,s_n$ is denoted $(s_i)_{i=1}^n$. The subsequence of elements satisfying condition C is denoted $(s_i \varepsilon S | C(s_i))_{i=1}^n$ by analogy with the notation for sets. The ith element of S is denoted by S(i).