Management Applications H. Morgan Editor

The Notions of Consistency and Predicate Locks in a Database System

K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger
IBM Research Laboratory
San Jose, California

In database systems, users access shared data under the assumption that the data satisfies certain consistency constraints. This paper defines the concepts of transaction, consistency and schedule and shows that consistency requires 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 database. These subsets may be specified by predicates. An implementation of predicate locks which satisfies the consistency condition is suggested.

Key Words and Phrases: consistency, lock, database, concurrency, transaction

CR Categories: 4.32, 4.33

1. Introduction

In database systems, users access shared data under the assumption that the data satisfies certain consistency assertions. For simplicity consider a system with a fixed set of named resources called *entities*. Each entity has a name and a value. Examples of such assertions are

"A" is equal to "B",
"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. Read and write are examples of such actions. We assume that actions are atomic; that is, if two processes concurrently perform actions, the effect will be as though one of the actions were performed before the other.

One might think that consistency constraints could be enforced at each action but this is not true. One 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 other not yet credited. This violates a constraint that the number of dollars in the system is constant. For this reason, the actions of a process are grouped into sequences called *transactions* which are units of consistency. In general, consistency assertions cannot be enforced before the end of a transaction. In this paper it is assumed that each transaction, when executed alone, transforms a consistent state into a new consistent state; that is, transactions preserve consistency.

Having grouped actions into transactions, we are interested in the problem of running transactions with maximal concurrency by interleaving actions from several transactions while continuing to give each transaction a consistent view of the system state. In such an environment, each transaction must employ a locking protocol to insure that it and others do not access data which is temporarily inconsistent. This lock protocol results in an additional set of actions called lock and unlock. A particular sequencing of the actions of a set of transactions is called a *schedule*. A schedule which gives each transaction a consistent view of the state is called a *consistent schedule*.

Not all consistent schedules for a set of transactions give exactly the same state (i.e. consistency is a weaker property than determinacy). For example, in an airlines reservation system if a set of transactions each requests a seat on a particular flight, then each consistent schedule will have the property that no seat is sold twice and no request is denied if there is a free seat, but two distinct consistent schedules may differ in the details of the seat assignment.

In the next section, we consider the problems of

Authors' address: IBM Corp., Monterey and Cottle Roads, San Jose, CA 95193.

locking and consistency in more detail. The discussion is applicable to database systems and to more conventional environments such as operating systems. The principal result is that consistency requires 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 database system. A phenomenon called *phantoms* seems to imply that one must lock logical subsets of the database rather than locking individual records present in the database. 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

To see the problems associated with running transactions concurrently consider the two transactions $\mathbf{T}_1$ and $\mathbf{T}_2$ of Figure 1 (below):

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

Suppose that the only assertion about the system state is that $A = B$. Although when considered alone both $\mathbf{T}_1$ and $\mathbf{T}_2$ conserve consistency, they have the following properties:

temporary inconsistency—after the first step of $\mathbf{T}_1$ or $\mathbf{T}_2$, $A \neq B$ and so the state is inconsistent. (1a)

conflict—if transaction $\mathbf{T}_2$ is scheduled to run between the first and second steps of $\mathbf{T}_1$, then the end result is $A \neq B$, which is an inconsistent state. (1b)

The problem of temporary inconsistency is inherent. Conflict on the other hand is not inherent and is undesirable.

If transactions are run one after another with no concurrency then conflict never arises. 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 may be obtained 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 avoiding conflict is to partition entities into disjoint classes. One can then schedule transactions concurrently only if they use distinct classes of entities. Transactions using common parts of the state must 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 several reasons. In terms of the above discussion, 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. Still another motive for locking is reproducibility of reads. Unless a transaction locks an entity, successive reads of the entity may yield distinct values reflecting updates by concurrent transactions. This has little to do with consistency constraints; rather it rests on the notion that entities hold their values until updated.

Recovery and transaction backup provide an additional motive for locking. Database systems usually maintain a log of all changes made by each transaction. This log forms an audit trail. It may also be used for backup. Backup arises not only from deadlock-preemption but also from protection violations, hardware errors, and human errors. One backup procedure for a transaction $\mathbf{T}$ is to undo all of its updates as recorded in the log. Then all entities locked by $\mathbf{T}$ may be unlocked and $\mathbf{T}$ may be reset to its initial state. As Davies and Bjork [1, 2] point out, this procedure may not work correctly after $\mathbf{T}$ has unlocked (committed) any entities which it has modified. This implies that (update) locks should be held to the end of a transaction.

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 and is mentioned parenthetically as the section develops.

If transaction $\mathbf{T}_1$ attempts to lock entity e_1 which is already locked by transaction $\mathbf{T}_2$ then either $\mathbf{T}_1$ must wait for $\mathbf{T}_2$ to unlock e_1 or $\mathbf{T}_1$ must preempt e_1 from $\mathbf{T}_2$. If $\mathbf{T}_1$ waits and then $\mathbf{T}_2$ attempts to lock an entity e_2 locked by $\mathbf{T}_1$ then $\mathbf{T}_2$ must wait or preempt. If both $\mathbf{T}_1$ and $\mathbf{T}_2$ 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 [3] presents a scheme for deciding which transaction to preempt. When a resource is preempted, the preempted transaction must be backed up.

To insure that each transaction sees a consistent state, a transaction must not request a new lock after releasing some lock. To state and prove this result we must proceed more formally. However, for the sake of simplicity, we assume in the sequel that *all* transactions have the property that they do not relock an entity at step i which is already locked at step i, that they do not unlock an entity at step i which is not locked through step i, and that they end with no locks set.

A *transaction* is a sequence[1]: $\mathbf{T} = ((T, a_i, e_i))_{i=1}^{n}$ 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

for some $j \leq i$, a_j = lock and $e_j = e$, and (2a)
there is no $k, j < k < i$, such that
a_k = unlock and $e_k = e$. (2b)

A transaction $\mathbf{T}$ is *well-formed* if

for each step $i = 1, \ldots, n$, (3a)
if a_i = lock then e_i is not locked by $\mathbf{T}$ through step $i - 1$,
if $a_i \neq$ lock then e_i is locked by $\mathbf{T}$ through step i,

and

at step n, only e_n is still locked by $\mathbf{T}$ and a_n = unlock. (3b)

Figure 2 shows two well-formed versions of transaction $\mathbf{T}_1$ from Figure 1.

Any sequence obtained by collating the actions of transactions $\mathbf{T}_1, \ldots, \mathbf{T}_n$ is called a *schedule* for $\mathbf{T}_1, \ldots, \mathbf{T}_n$. If the schedule takes actions from one transaction at a time it is called a *serial* schedule. More formally, a *schedule* for a set of transactions $\mathbf{T}_1, \ldots, \mathbf{T}_n$ is any sequence $\mathbf{S} = ((T_i, a_i, e_i))_{i=1}^{m}$ such that

for each $j = 1, \ldots, n$, (4a)
$\mathbf{T}_j = ((T_i, a_i, e_i) \in \mathbf{S} \mid T_i = T_j)_{i=1}^{m}$

and

The length of $\mathbf{S}$, m, is the sum of the lengths of the transactions $T_1, \ldots, T_n$ (i.e. $\mathbf{S}$ contains only elements of $T_1, \ldots, T_n$). (4b)

Note that m is the number of steps in *all* transactions.

A schedule $\mathbf{S}$ is *serial* if for some permutation π, $\mathbf{S} = \mathbf{T}_{\pi(1)}\mathbf{T}_{\pi(2)} \ldots \mathbf{T}_{\pi(n)}$ (i.e. $\mathbf{S}$ is the concatenation of the transactions). Figure 3 gives three examples of schedules for a set of three transactions.

Nonserial 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 $\mathbf{S}$, $DEP(\mathbf{S})$, is a ternary relation on $T \times E \times T$ (where T is the set of all transaction names in $\mathbf{S}$ and E is the set of all entities) defined by $(T_1, e, T_2) \in DEP(\mathbf{S})$ iff for some $i < j$

$\mathbf{S} = (\ldots, (T_1, a_i, e), \ldots, (T_2, a_j, e), \ldots)$, and (5a)
there is no k such that $i < k < j$ and $e_k = e$. (5b)

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

Informally, if (T_1, e, T_2) is in $DEP(\mathbf{S})$ then entity e is an output of $\mathbf{T}_1$ and an input of $\mathbf{T}_2$ and $\mathbf{T}_1$ gives e to $\mathbf{T}_2$. 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 (5a) and adjoin the clause "and a_k is an update action" to (5b)).

Two schedules, $\mathbf{S}_1$ and $\mathbf{S}_2$ are *equivalent* if $DEP(\mathbf{S}_1) = DEP(\mathbf{S}_2)$ and a schedule $\mathbf{S}_1$ is *consistent* if it has an equivalent serial schedule. Figure 4 illustrates these definitions. It shows three schedules, where $\mathbf{S}_1$ is consistent, $\mathbf{S}_2$ is not consistent and $\mathbf{S}_3$ is serial (therefore consistent). Since a serial schedule starts with a consistent state and since each transaction (when run alone) transforms a consistent state into a new consistent state, a serial schedule gives each transaction a consistent set of inputs. If a set of transactions is consistently scheduled, then each transaction sees the same state it would see in the corresponding serial schedule (i.e. a consistent state). These observations justify the dual use of the term consistency to describe states and schedules.

It is very easy to explain 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). This observation yields the following important properties of serial schedules:

If $\mathbf{T}_1$ and $\mathbf{T}_2$ are any two transactions and e_1 and e_2 are any entities, then $(T_1, e_1, T_2) \in DEP(\mathbf{S})$ implies $(T_2, e_2, T_1) \notin DEP(\mathbf{S})$. (6a)

More generally,

The binary relation $<$ on the set of transactions is defined by: $T_1 < T_2$ if and only if $(T_1, e, T_2) \in DEP(\mathbf{S})$ for some entity e. Then $<$ is an acyclic relation which may be extended to a total order of the transactions. (6b)

Any consistent schedule also has these properties because it has the same dependency set as some serial schedule. Conversely, it will later be shown that any schedule with property (6b) is consistent.

We would like to further characterize those nonserial 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 *locked by transaction* $\mathbf{T}$ *through step k of schedule* $\mathbf{S}$ if

there is a $j \leq k$ such that $\mathbf{S}(j) = (T, \text{lock}, e)$ and (7a)
there is no j', $j < j' < k$ such that $\mathbf{S}(j') = (T, \text{unlock}, e)$. (7b)

Fig. 2. Two well-formed versions of transaction $\mathbf{T}_1$ of Fig. 1.

T_{11}:

T_{11}	LOCK		A
T_{11}	A + 100	→	A
T_{11}	UNLOCK		A
T_{11}	LOCK		B
T_{11}	B + 100	→	B
T_{11}	UNLOCK		B

T_{12}:

T_{12}	LOCK		A
T_{12}	A + 100	→	A
T_{12}	LOCK		B
T_{12}	UNLOCK		A
T_{12}	B + 100	→	B
T_{12}	UNLOCK		B

Fig. 3. Schedules for three transactions $\mathbf{T}_1$, $\mathbf{T}_2$, $\mathbf{T}_3$. $\mathbf{S}_2$ is a serial schedule. Each small rectangle represents a transaction step.

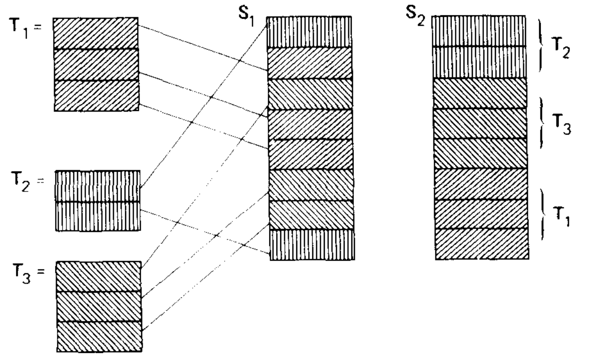


Fig. 4. Three schedules for $\mathbf{T}_1$, $\mathbf{T}_2$ of Fig. 1. $\mathbf{S}_1$ is equivalent to serial schedule $\mathbf{S}_3$ and hence is consistent. $\mathbf{S}_2$ is inconsistent.

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

Schedule $\mathbf{S}$ is *legal* if for all k, if $\mathbf{S}(k) = (T, a, e)$ and e is locked by $\mathbf{T}$, through step k, then 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 at each instant choosing a particular transaction step from the set of all next steps of all incomplete transactions. 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 in all contexts, 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 (3a), (3b):

Consistency requires that transactions be well-formed. Unless all transactions are well-formed, it is possible to construct a legal but inconsistent schedule. (8a)

To prove this, consider any transaction $\mathbf{T}_1 = (T_1, a_i, e_i)_{i=1}^{n}$ which is not well-formed. Then for some step k, $\mathbf{T}_1$ does not have e_k locked through step k. Consider the (two-phase well-formed) transaction $\mathbf{T}_2 = ((T_2, \text{lock}, e_k), (T_2, \text{read}, e_k), (T_2, \text{write}, e_k), (T_2, \text{unlock}, e_k))$. The schedule $\mathbf{S} = (\mathbf{T}_1(i))_{i=1}^{k-1}\mathbf{T}_2(1), \mathbf{T}_2(2), \mathbf{T}_1(k), \mathbf{T}_2(3), \mathbf{T}_2(4), (\mathbf{T}_1(k))_{i=k+1}^{n}$ is legal. Since (T_1, e_k, T_2) and (T_2, e_k, T_1) are both in $DEP(\mathbf{S})$, $\mathbf{S}$ is not equivalent to any serial schedule (by property (6a)). So $\mathbf{S}$ is not a consistent schedule and (8a) is established. Intuitively, T_1 could change e_k after T_2 read it but before T_2 wrote it. This would not be possible in a serial (i.e. consistent schedule).

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 $\mathbf{T} = ((T, a_i, e_i))_{i=1}^{n}$ is *two-phase* if for some $j < n$,

$i < j$ implies $a_i \neq$ unlock,
$i = j$ implies $a_i =$ unlock,
$i > j$ implies $a_i \neq$ lock.

Steps $1, \ldots, j - 1$ are called the *growing phase* and steps $j, \ldots, n$ are the *shrinking phase* of T.

Transaction $\mathbf{T}_{11}$ of Figure 2 is not two-phase since it locks B after releasing A. Transaction $\mathbf{T}_{12}$ of Figure 2 is well-formed and two-phase. To see that $\mathbf{T}_{11}$ may see an inconsistent state, consider the legal schedule S shown in Figure 5. In the schedule $\mathbf{S}$, $\mathbf{T}_{12}$ sees A from $\mathbf{T}_{11}$ and $\mathbf{T}_{11}$ sees B from $\mathbf{T}_{12}$. So $\mathbf{S}$ is not equivalent to any serial schedule and hence $\mathbf{S}$ is inconsistent. This construction can be generalized to prove:

Consistency requires that transactions be two-phase. That is, unless all transactions are two-phase, it is possible to construct a legal but inconsistent schedule. (8b)

Conversely,

If each transaction in the set of transactions $T = \{\mathbf{T}_1, \ldots, \mathbf{T}_n\}$ is well-formed and two-phase then any legal schedule for T is consistent. (8c)

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

First define the integer $SHRINK(\mathbf{T}_i)$ for each transaction $\mathbf{T}_i$ to be the least integer j such that $\mathbf{T}_i$ unlocks some entity at step j of $\mathbf{S}$:

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

If each transaction $\mathbf{T}_i$ is non-null then $SHRINK(\mathbf{T}_i)$ is well-defined because each $\mathbf{T}_i$ is well-formed.

We now argue that for any transactions $\mathbf{T}_1$ and $\mathbf{T}_2$ and entity e, if $(T_1, e, T_2) \in DEP(\mathbf{S})$ then $SHRINK(\mathbf{T}_1)$ is less than $SHRINK(\mathbf{T}_2)$. For if $(T_1, e, T_2) \in DEP(\mathbf{S})$ then by definition of $DEP(S)$ there are integers i and j such that $\mathbf{S} = (\ldots, (T_1, a_i, e), \ldots, (T_2, a_j, e), \ldots)$ and so that for any integer k between i and j, $e_k \neq e$ by Definition (5). Since $\mathbf{S}$ is legal, e must be locked only by $\mathbf{T}_1$ through step i of $\mathbf{S}$ and since $\mathbf{T}_2$ is well formed e must be locked only by $\mathbf{T}_2$ through step j of $\mathbf{S}$. So a_i = unlock and a_j = lock. This immediately implies that $SHRINK(\mathbf{T}_1)$ is less than or equal to i. Since $\mathbf{T}_2$ is two-phase, no unlock by $\mathbf{T}_2$ precedes step j of $\mathbf{S}$ so $SHRINK(\mathbf{T}_2)$ is greater than j.

Thus we have shown that if $\mathbf{T}_1 < \mathbf{T}_2$ then $SHRINK(\mathbf{T}_1)$ is less than $SHRINK(\mathbf{T}_2)$. This implies property (6b) and hence $<$ can be extended to a total order $\ll$ on T.

Assume without loss of generality that $\mathbf{T}_1 \ll \mathbf{T}_2 \ll \ldots \ll \mathbf{T}_n$. Induce on n to show that $\mathbf{S}$ is equivalent to the serial schedule $\mathbf{T}_1, \ldots, \mathbf{T}_n$. If $n = 1$ the result is trivial. The induction step follows in two steps.

First show that $\mathbf{S}$ is equivalent to the schedule

$$\mathbf{S}' = \mathbf{T}_1((T_i, a_i, e_i) \in \mathbf{S} \mid T_i \neq T_1)_{i=1}^{m}.$$

Then note that by hypothesis

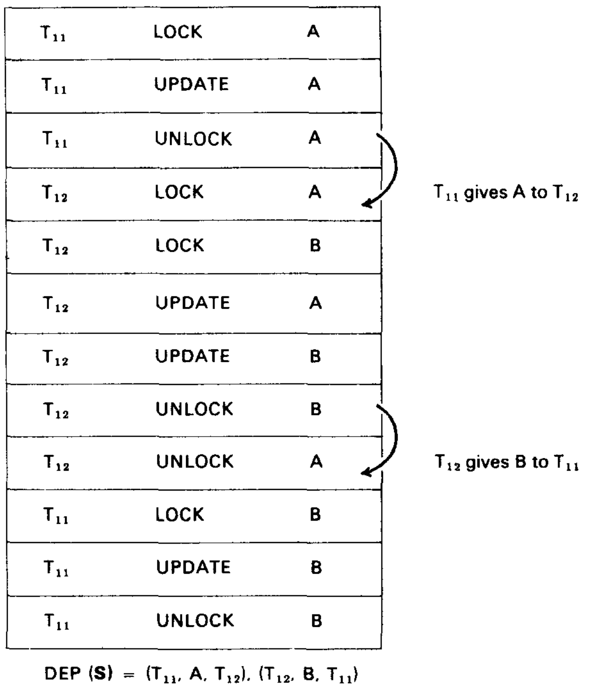
$((T_i, a_i, e_i) \in \mathbf{S} \mid T_i \neq T_1)_{i=1}^{m}$ is equivalent to $\mathbf{T}_2, \ldots, \mathbf{T}_n$.

So $\mathbf{S}'$ is equivalent to $\mathbf{T}_1, \mathbf{T}_2, \ldots, \mathbf{T}_n$. But $\mathbf{T}_1, \ldots, \mathbf{T}_n$ is a serial schedule so $\mathbf{S}$ is equivalent to a serial schedule and is consistent. Figure 6 gives a graphic illustration of the construction of a serial schedule from $\mathbf{S}$. To summarize then,

If the transactions $\mathbf{T}_1, \ldots, \mathbf{T}_n$ are each well-formed and two-phase then any legal schedule is consistent. (8d)

Unless transaction $\mathbf{T}$ is well-formed and two-phase there is a transaction $\mathbf{T}'$, which is well-formed and two-phase, such that $\mathbf{T}$ and $\mathbf{T}'$ have a legal but inconsistent schedule. (8e)

Fig. 5. A schedule for transactions $\mathbf{T}_{11}$ and $\mathbf{T}_{12}$ which is legal but not consistent because $\mathbf{T}_{11}$ is not two-phase.



T_{11}	LOCK	A
T_{11}	UPDATE	A
T_{11}	UNLOCK	A
T_{12}	LOCK	A
T_{12}	LOCK	B
T_{12}	UPDATE	A
T_{12}	UPDATE	B
T_{12}	UNLOCK	B
T_{12}	UNLOCK	A
T_{11}	LOCK	B
T_{11}	UPDATE	B
T_{11}	UNLOCK	B

T_{11} gives A to T_{12}
T_{12} gives B to T_{11}

DEP (S) = $(T_{11}, A, T_{12}), (T_{12}, B, T_{11})$

Clearly a transaction run alone is consistent. Further, any set of transactions which do not interact (i.e. $DEP(\mathbf{S}) = \varnothing$) can be consistently scheduled in any order without locking. Even if the transactions interact, the two-phase restriction may be too strong. If, for example, transaction $\mathbf{T}_{12}$ of Figure 2 had updated entity C rather than entity B, then any legal schedule for $\mathbf{T}_{11}$ and $\mathbf{T}_{12}$ would be consistent even though neither transaction is two-phase. However, if one added a transaction $\mathbf{T}_{13}$ which accesses entities A, B, and C, then the new transaction set would have legal but inconsistent schedules. It therefore seems difficult to give nontrivial necessary conditions for all legal schedules for a set of transactions to be consistent ((8d) is sufficient). We can make the following assertion: if one intends to run a transaction concurrently with an *unknown* set of other transactions then, to guarantee that all legal schedules be consistent, all transactions must be well-formed and two-phase.

3. Predicate Locks

Section 2 introduced the notions of consistency and locking; 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 database environment. Aside from the problem of scale (millions 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 database 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 solution.

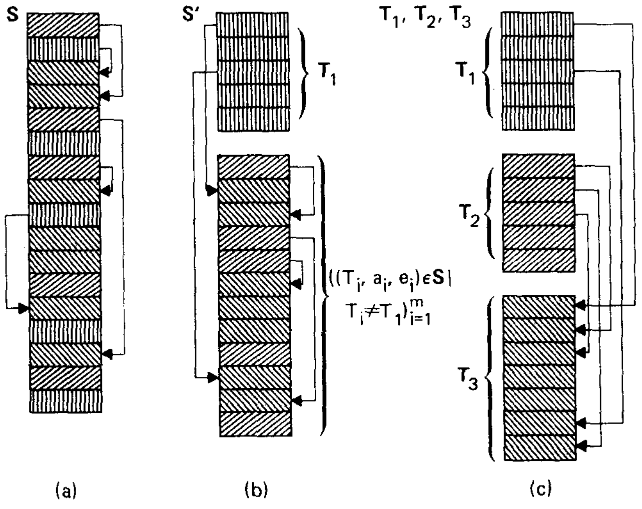
For definiteness we adopt the relational model of data (Codd [4]). The database consists of a collection of relations, $R_1, R_2, \ldots, R_n$. Each relation can be thought of as a table or flat file. Each column of the relation is called a *domain* and each element of the relation (row) is called a *tuple* (record). Each tuple consists of a fixed number of *fields*. Each domain has a name. Figure 7 shows an example of such a database.

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 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 of the storage they occupy.

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

Fig. 6. A graphic illustration of the construction of a serial schedule from a consistent schedule. The arrows show the dependencies of **S**. $\mathbf{T}_1 \ll \mathbf{T}_2 \ll \mathbf{T}_3$ and so **S′** has the same dependencies as **S**. The induction hypothesis applies to **S′** to give $\mathbf{T}_1, \mathbf{T}_2, \mathbf{T}_3$.



Associately addressing the *ACCOUNTS* relation, locking any accounts located in Napa. (9a)

Summing the balances in the locked accounts. (9b)

Locking the Napa tuple in *ASSETS* and comparing its value with the computed sum. (9c)

Releasing all locks. (9d)

If a second transaction T_2 inserts a new tuple in *ACCOUNTS* with Location = Napa and adds the deposit to the Napa assets and if T_2 is scheduled between steps (9b) and (9c) 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 "nonexistence" of the tuple is being locked. Such nonexistent tuples are called *phantoms*. Inspection of the earlier example shows that T_1 should lock not only all existing Napa accounts but also all phantom ones.

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: $\{\text{Napa}\} \times \mathit{INTEGERS} \times \mathit{INTEGERS}$. This set is infinite so there is little hope of locking each individual tuple of the set. 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 by specifying 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 $P(t) \triangleq t = (\text{NAPA}, 32123, 1050)$. 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

Fig. 7. The sample database.

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.

recursively unsolvable (Kleene [5]), so it is not clear how to make predicate locks "work." A method for scheduling predicate locks is introduced first by example and then more abstractly.

In the sample database 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; otherwise, the request can be granted immediately.

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. First we need to decide on the lockable entities. In [8] a field was chosen as the basic unit of locking. This choice gives maximal concurrency but presents many notational complexities. For the sake of simplicity, the formal development of predicate locks here is done for tuple-level locking rather than field-level locking. After a formal development of tuple-level predicate locks the generalization to field-level predicate locks is informally discussed.

A particular *action* on a single tuple may be denoted by (R, t, a), meaning that tuple t of relation R is accessed in mode a. Two modes are distinguished here:

a = read allows sharing with other readers,

while

a = write requires an exclusive lock on tuple t (update, insert, delete are all examples of write access).

The action *reads* tuple t if a = read and it *writes* tuple t if a = write.

Reading the balance of account number 32123 would be an action

(*ACCOUNTS*, (Napa, 32123, 1050), read)

An update of the balance by $50 would involve two actions and *two* tuples, first

(*ACCOUNTS*, (Napa, 32123, 1050), write)

and also

(*ACCOUNTS*, (Napa, 32123, 1100), 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.

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 3123 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 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 become 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, one specifies the access:

(*ACCOUNTS*, Number = 32123, read)

which refers to either a single tuple or no tuples, since account numbers are unique. An access which updates the balance of account number 32123 would be denoted by

(*ACCOUNTS*, Number = 32123, write).

Consistency assertions require that such an access be followed by an access

$(ASSETS$, Location = 'Napa', 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', read).

To proceed more formally we need the following definitions. If the relation R is drawn from the Cartesian product of sets $S_1, S_2, \ldots S_n$ $(R \subseteq \times_{i=1}^{n} S_i)$, then any predicate P defined on all tuples $(s_1, \ldots, s_n) \in \times_{i=1}^{n} S_i$ 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, a) where P is an admissible predicate. Such an access is equivalent to the (possibly infinite) set of actions (R, t, a) where $P(t) = TRUE$, and where t ranges over the Cartesian product underlying R. In particular, it *reads* all such tuples if a = read and *writes* all such tuples if a = write. A *predicate lock* on relation R is denoted by (R, P, a) where P is an admissible predicate for R and a is an access mode.

An action (R, t, a) is said to *satisfy* predicate lock (R', P', a') if

$R = R'$ and (10a)
$P'(t) = TRUE$ and (10b)
$a = a'$ or $a' =$ write. (10c)

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

The action *conflicts* with the predicate lock if

$R = R'$ and (11a)
$P'(t) = TRUE$ and (11b)
$a =$ write or $a' =$ write. (11c)

To give an example, the predicate lock

$L = (ACCOUNTS$, Location = Napa, read)

is satisfied by the action

$(ACCOUNTS$, (Napa, 3213, 1050), read)

and conflicts with the action

$(ACCOUNTS$, (Napa, 3213, 1050), write).

Satisfiability and conflict are defined analogously for accesses. Access $A = (R, P, a)$ *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, a) 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, a) conflicts with L.

As an example, the access which moves account 23175 from Napa to Sonoma would be denoted

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

This access would require that the transaction have a lock on the $ACCOUNTS$ relation of the form $(ACCOUNTS, P$, write), where 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 collating (merging) of the transaction sequences. The dependency relation is defined by choosing (tuple, relation) pairs as the entities (for all tuples in the Cartesian products underlying the relations). 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 ot **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 $i < j$:

S$(i) = (T_1, A_1)$ and A_1 reads or writes entity e, (12a)
S$(j) = (T_2, A_2)$ and A_2 reads or writes entity e (12b)
and not both A_1 and A_2 simply reads e,
for any k between i and j, if **S**$(k) = (T_3, A_3)$ (12c)
then A_3 does *not* write entity e.

The generalization of tuple-level predicate locks to field-level predicate locks can be done as follows (see [8] for a formal development of this notion): A particular field-level access to a relation reads, writes, or ignores each of the fields of the relation specified by the access predicate. A field-level predicate lock locks particular fields of the tuples covered by the predicate. Fields are either ignored by the lock or are locked in read or write mode. Two predicate locks conflict if their predicates are mutually satisfiable and one demands write access to a field locked in read or write mode by the other. A field-level access satisfies a predicate lock if it only accesses tuples covered by the predicate lock and it only reads fields locked in read mode

Fig. 8. An example of the *LOCK* table.

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

by the lock and only writes fields locked in write mode by the predicate lock. Similarly, an access conflicts with a predicate lock if the two predicates are mutually satisfiable and the access reads a field of a tuple locked by the lock in write mode or it writes a field locked by the predicate lock in read or write mode. Given these definitions of access, satisfiability, and conflict, the development of this section generalizes to field-level predicate locks.

To give concrete examples, the reading of an account balance is denoted by the access

$(ACCOUNT, \text{Number} = 32123, \{(\text{Number, read}), (\text{Balance, write})\})$

which ignores the Location field, reads the Number field, and updates the Balance field. This access satisfies the predicate lock

$(ACCOUNT, \text{Number} = 32123, \{(\text{Number, write}), (\text{Balance, write})\})$

and this predicate lock conflicts with the predicate lock

$(ACCOUNT, \text{Number} = 32123, \{(\text{Number, read})\})$.

The access does not satisfy the latter predicate above.

We now return to the simpler model where locks apply to whole tuples. To implement arbitrary predicate locks, associate with the database a table called *LOCK* which is a binary relation between transactions and predicate locks (see Figure 8).

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

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

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 checks the following properties:

All transactions are well-formed and two-phase. (13a)

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 t until T releases the predicate lock. *That is, predicate locks solve the problem of phantoms.* (13b)

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 or not two predicate locks conflict. In general this is a recursively unsolvable problem (even if predicates are restricted to using the arithmetic operators $+, *, -, \div$ as shown by Gödel (see Kleene [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

$$\langle\text{field name}\rangle \left\{ \begin{matrix} `<' \\ `=' \\ `\neq' \\ `>' \end{matrix} \right\} \langle\text{constant}\rangle$$

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 (see Kleene [5]) showed a procedure to decide if two predicates overlap for a class of predicates slightly more general than simple predicates (he allowed $+, -, <, =, \neq, >$, 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 simple predicate locks L and L' conflict is a fairly straightforward matter. Suppose $L = (R, P, a)$ and $L' = (R', P', a')$ are two predicate locks. Then

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

If neither a = write nor a' = write then there is no conflict. (14b)

Otherwise, there will be no conflict only if there is no tuple t such that $P \wedge P'(t)$ is $TRUE$. (14c)