RJ 1606 (#23896)
June 30, 1975
Computer Science

GRANULARITY OF LOCKS IN A LARGE SHARED DATA BASE

J. N. Gray, R. A. Lorie and G. R. Putzolu

IBM Research Laboratory
San Jose, California

ABSTRACT: This paper proposes a locking protocol which associates locks with sets of resources. This protocol allows simultaneous locking at various granularities by different transactions. It is based on the introduction of additional lock modes besides the conventional share mode and exclusive mode. The protocol is generalized from simple hierarchies to directed acyclic graphs and to dynamic graphs. The issues of scheduling and granting conflicting requests for the same resource are then discussed. Lastly, these ideas are compared with the lock mechanisms provided by existing data management systems.

I. INTRODUCTION:

We assume the data base consists of a collection of records and constraints defined on these records. There are physical constraints (ex: in a list of records, if a record A points to record B then record B must exist) as well as logical constraints (ex: conservation of money in a bank checking account application). When all such constraints are satisfied the data base is said to be consistent.

A transaction is a series of accesses (for read or write operations) to the data base which, applied to a consistent data base, will produce a consistent data base. During the execution of a transaction, the data base may be temporarily inconsistent. The programs used to perform the transactions assume that they "see" a consistent data base. So if several transactions are run concurrently, a locking mechanism must be used to insure that one transaction does not see temporarily inconsistent data caused by another transaction. Also, even if there are no consistency constraints, locks must be used so that the updates of one transaction are not made available to others before the transaction completes. Otherwise, transaction backup might cascade to other transactions which read or updated the "backed up" updates.

Section II of the paper gives a brief description of a possible locking mechanism. It introduces the notion of granularity of lockable objects. Very informally, granularity refers to the size of a lockable object. The following and main Section III outlines a mechanism which simultaneously supports fine and gross lockable objects organized into a hierarchy. The generalization of these notions to directed acyclic graphs is then presented. Section IV discusses the problems of scheduling lock requests. Section V relates these ideas to existing data base systems and briefly describes our use of this protocol in an experimental data base system.

II.GRANULARITY OF LOCKS:

The description of a lock mechanism must cover the definition of the lockable objects, the operations which can be performed on the objects, how and when objects are allocated to particular transactions, and the duration of an allocation. A previous paper [1] discussed in some detail a lock protocol which would insure that the data base state obtained after running the same transactions concurrently is equivalent to the state obtained by running the same transactions sequentially in some arbitrary order determined by the system. This defines the consistency provided by the system.

Suppose the lockable object is a simple record in a file. The proposed protocol can be stated as:

(a) Recognize the classical notion of shared locks for read operations and exclusive locks for write operations on objects.

(b) Any record must be appropriately locked before being accessed.

(c) Any lock is kept to the end of transaction.

Note that in order to work correctly the notion of locking a record must be extended to include the ability to lock the non-existence of a record (i.e. to prevent the insertion of "phantom" records by other transactions). This has been extensively discussed in [1]. A forthcomming paper genaralizes this notion of consistency to include protocols which release locks before the end of the transaction. The present paper also applies those more general protocols.

It is a general problem in large integrated data bases that a transaction does not know which records it will access. So to avoid locking entire files or areas in advance, locks are requested dynamically. This creates a scheduling problem and the chosen scheduler must be prepared to handle deadlock situations. We return to this problem in section IV.

The choice of lock granularity presents a tradeoff between concurrency and overhead. On the one hand, concurrency is increased if a fine unit of locking (for example a record or field) is chosen. Such a choice is appropriate to "simple" transactions which access a few records. If a transaction accesses many records there are many locks. Each such access incurs the computational overhead of setting and perhaps waiting for a lock and the storage overhead of representing the lock until the end of transaction. Using a coarse unit of locking (for example a file) is probably convenient for a transaction which accesses many records. However, such a coarse unit discriminates against transactions which only want to lock one member of the file. From this discussion it follows that one needs a multiplicity of granularities of lockable objects.

III. HIERARCHICAL LOCKS:

In [1] it was proposed that one locks sets of records chosen from a file by specifying a predicate expression which "selects" the set of records to be locked. For example, using the obvious syntax, we could lock all employees in the EMPLOYEE file who work in the accounting department by:

```
LOCK EMPLOYEE WHERE (DEPARTMENT = 'ACCOUNTING').
```

(DEPARTMENT = 'ACCOUNTING' is the predicate.) Similarly,

```
LOCK EMPLOYEE WHERE (TRUE)
```

would lock the entire employee file.

Two locks conflict if their predicates are mutually satisfiable and one of them is exclusive. If two lock requests conflict, one must wait. Predicate locks have the virtue of generality but have the flaw that to grant a new lock it must be compared against the predicate of every outstanding lock. Predicate comparison is a computationally difficult problem. So we have come to feel that predicate locking is an excellent paradigm for the more efficient scheme described below which partitions the data base into a hierarchy; each subtree of the hierarchy corresponds to a very simple predicate and a protocol is adopted which makes conflict testing very easy.

In order to achieve efficient locking at several granularities, the set of resources and their corresponding locks are organized into a hierarchy. The hierarchy of Figure 1 may be suggestive. We adopt the notation that each level of the hierarchy is given a node type which is a generic name for all the node instances of that type. For example, the data base has nodes of type area as its immediate descendants, each area in turn has nodes of type file as its immediate descendants and each file has nodes of type record as its immediate descendants in the hierarchy. Since it is a hierarchy each node has a unique parent.

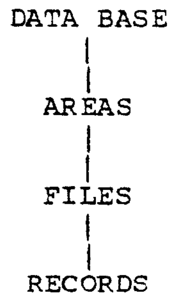

Figure 1. A sample lock hierarchy.

Each node of the hierarchy can be locked. If one requests _exclusive_ access (X) to a particular node, then when the request is granted, the requestor has exclusive access to that node _and implicitly to each of its descendants_. If one requests _shared_ access (S) to a particular node, then when the request is granted, the requestor has shared access to that node _and implicitly to each descendant of that node_. These two access modes lock an _entire subtree_ rooted at the requested node.

Our goal is to find some technique for _implicitly_ locking an entire subtree. The way to _explicitly_ lock an entire subtree is to lock each node of the subtree in leaf-to-root order. This approach has the problem that setting a file lock requires locking each record in the file. The scheme has the virtue that it is cheap to lock individual leaves of the tree (i.e. sets only one lock per leaf); whereas locking the root of the tree is expensive if the tree is large (i.e. requires locking _all_ nodes of the tree). It does no implicit locking, all locks are explicit. The following approach makes locking the root cheap but makes it somewhat more expensive to lock individual leaves of the tree.

In order to lock a subtree rooted at node R in share or exclusive mode it is important to _prevent_ share or exclusive locks on the ancestors of R which would implicitly lock R and its descendants. Hence a new access mode, _intention mode_ (I), is introduced. Intention mode is used to "tag" (lock) all ancestors of a node to be locked in share or exclusive mode. These tags signal the fact that locking is being done at a "finer" level and prevent implicit or explicit exclusive or share locks on the ancestors.

The protocol to lock a subtree rooted at node R in exclusive or share mode is to lock all ancestors of R in intention mode and to lock node R in exclusive or share mode. So for example using Figure 1, to lock a particular file obtain intention access to the data base, to the area containing the file and then request exclusive (or share) access to the file itself. This implicitly locks all records of the file in exclusive (or shared) mode.

Access modes and compatibility:

We say two lock requests for the same node are compatible if they can be granted concurrently without sacrificing consistency. The mode of the request determines its compatibility with requests made by other transactions. The three modes: X, S, and I are incompatible with one another but distinct S requests may be granted together and distinct I requests may be granted together.

The compatibilities among modes derive from their semantics. Share mode allows reading but not modification of the corresponding resource by the requestor and by other transactions. The semantics of exclusive mode is that the grantee may read and modify the resource and no other transaction may read or modify the resource while the exclusive lock is set. The reason for dichotomizing share and exclusive access is that several share requests can be granted concurrently (are compatible) whereas an exclusive request is not compatible with any other request. Intention mode was introduced to be incompatible with share and exclusive mode (to prevent share and exclusive locks). However, intention mode is compatible with itself since two transactions having intention access to a node will explicitly lock descendants of the node in X, S, or I mode and thereby will either be compatible with one another or will be scheduled on the basis of their requests at the finer level. For example, two transactions can be concurrently granted the data base and some area and some file in intention mode. In this case their explicit locks on records in the file will resolve any conflicts among them.

The notion of intention mode is refined to intention share mode (IS) and intention exclusive mode (IX) for two reasons: First, intention access is compatible with share (S) access if the intention only requests intention or share locks at the lower nodes of the tree (i.e. never requests an exclusive lock below the intention share node). Since read-only is a common form of access it will be profitable to distinguish this for greater concurrency. Secondly, if a transaction has an intention share lock on a node it can convert this to a share lock at a later time, but one cannot convert an intention exclusive lock to a share lock on a node (see below).

We recognize one further refinement of modes, namely share and intention exclusive mode (SIX). Suppose one transaction wants to read an entire subtree and to update particular nodes of that subtree. Using the modes provided so far it would have the options of: (a) requesting exclusive access to the root of the subtree and doing no further locking or (b) requesting intention exclusive access to the root of the subtree and explicitly locking the lower nodes in intention, share or exclusive mode. Alternative (a) has low concurrency. If only a small fraction of the read nodes are updated then alternative (b) has high locking overhead. The correct access mode would be share access to the subtree thereby allowing the transaction to read all nodes of the subtree without further locking and intention exclusive access to the subtree thereby allowing the transaction to set exclusive locks on those nodes in the subtree which are to be updated and IX or SIX locks on the intervening nodes. Since this is such a common case, SIX mode is introduced for this purpose. It is compatible with IS mode since other transactions requesting IS mode will explicitly lock lower nodes in IS or S mode thereby avoiding any updates (IX or X mode) produced by the SIX mode transaction. However SIX mode is not compatible with IX, S, SIX or X mode requests. Table 1 gives the compatibility of the request modes. (Note that compatibility is not transitive.)

Table 1. Compatibilities among access modes.

COMPATIBILITY					
	IS	IX	S	SIX	X
IS	YES	YES	YES	YES	NO
IX	YES	YES	NO	NO	NO
S	YES	NO	YES	NO	NO
SIX	YES	NO	NO	NO	NO
X	NO	NO	NO	NO	NO

To summarize, we recognize five modes of access to a resource:

X: Gives exclusive access to the requested node and to all descendants of the requested node without setting further locks. (It implicitly sets X locks on all descendants.)

S: Gives share access to the requested node and to all descendants of the requested node without setting further locks. (It implicitly sets S locks on all descendants of the requested node.)

IX: Gives intention exclusive access to the requested node and allows the requestor to explicitly lock descendants in X, S, SIX, IX or IS mode. (It does no implicit locking.)

IS: Gives intention share access to the requested node and allows the requestor to lock descendant nodes in S or IS mode. (It does no implicit locking.)

SIX: Gives share and intention exclusive access to the requested node. In particular it implicitly locks all descendants of the node in share mode and allows the requestor to explicitly lock descendant nodes in X, SIX, or IX mode. (Locking lower nodes in S or IS mode would give no increased access.)

IS mode is the weakest form of access to a resource. It carries fewer privileges than IX or S modes. IX mode allows IS, IX, S, SIX, and X mode locks to be set on descendant nodes while S mode allows read only access to all descendants of the node without further locking. SIX mode carries the privileges of S and of IX mode (hence the name SIX). X mode is the most privileged form of access and allows reading and writing of all descendants of a node without further locking. Hence the modes can be ranked in the partial order (lattice) of privileges shown in Figure 2. Note that it is not a total order since IX and S are incomparable. Given two modes, the least node greater than or equal to them in Figure 2 is called their supremum (see Table 3).

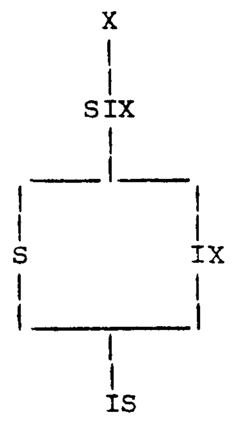

Figure 2. The partial ordering of modes by their privileges.

Rules for requesting nodes:

The implicit locking of nodes will not work if transactions are allowed to leap into the middle of the tree and begin locking nodes at random. The implicit locking implied by the S and X modes depends on all transactions obeying the following protocol:

(a) Before requesting an S or IS lock on a node, all ancestor nodes of the requested node must be held in IX or IS mode by the requestor.

(b) Before requesting an X, SIX, or IX lock on a node, all ancestor nodes of the requested node must be held by this requestor in SIX or IX mode.

(c) Locks should be released either at the end of the transaction (in any order) or in leaf to root order. In particular, if locks are not held to the end of transaction, one should not hold a lower lock after releasing its ancestor.

To paraphrase this, locks are requested root to leaf, and released leaf to root. Notice that leaf nodes are never requested in intention mode since they have no descendants.

Several examples:

It may be instructive to give a few examples of hierarchical request sequences:

To lock record R for read:

```
 lock data-base           with mode = IS
 lock area containing R   with mode = IS
 lock file containing R   with mode = IS
 lock record R            with mode = S
```

Don't panic, the transaction probably already has the data base, area and file lock.

To lock record R for write-exclusive access:

```
 lock data-base            with mode = IX
 lock area containing R    with mode = IX
 lock file containing R    with mode = IX
 lock record R             with mode = X
```

Note that if the records of this and the previous example are distinct, each request can be granted simultaneously to different transactions even though both refer to the same file.

To lock a file F for read and write access:

```
 lock data-base            with mode = IX
 lock area containing F    with mode = IX
 lock file F               with mode = X
```

Since this reserves exclusive access to the file, if this request uses the same file as the previous two examples it or the other transactions will have to wait.

To lock a file F for complete scan and occasional update:

```
 lock data-base            with mode = IX
 lock area containing F    with mode = IX
 lock file F               with mode = SIX
```

Thereafter, particular records in F can be locked for update by locking records in X mode. Notice that (unlike the previous example) this transaction is compatible with the first example. This is the reason for introducing SIX mode.

To quiesce the data base:

```
     lock data base   with mode = X
```

Note that this locks everyone else out.

Directed acyclic graphs of locks:

The notions so far introduced can be generalized to work for directed acyclic graphs (DAG) of resources rather than simply hierarchies of resources. A tree is a simple DAG. The key observation is that: to implicitly or explicitly lock a node, lock all the parents of the node in the DAG and so by induction lock all ancestors of the node. In particular, to lock a subgraph one must implicitly or explicitly lock all ancestors of the subgraph in the appropriate mode (for a tree there is only one parent). To give an example of a non-hierarchical structure, imagine the locks are organized as in Figure 3.

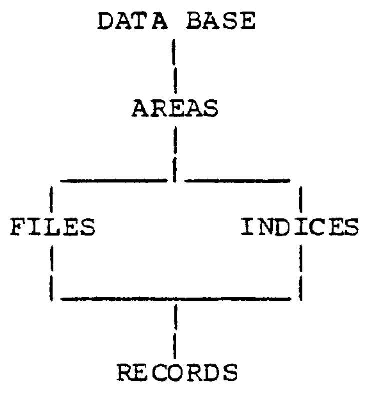

Figure 3. A non-hierarchical lock graph.

We postulate that areas are "physical" notions and that files, indices, and records are logical notions. The data base is a collection of areas. Each area is a collection of files and indices. Each file has a corresponding index in the same area. Each record belongs to some file and to its corresponding index. A record is comprised of field values and some field is indexed by the index associated with the file containing the record. The file gives a sequential access path to the records and the index gives an associative access path to the records based on field values. Since individual fields are never locked, they do not appear in the lock graph.

```
To write a record R in file F with index I:
 lock data base           with mode = IX
 lock area containing F   with mode = IX
 lock file F              with mode = IX
 lock index I             with mode = IX
 lock record R            with mode = X
```

Note that all paths to record R are locked. Alternaltively, one could lock F and I in exclusive mode thereby implicitly locking R in exclusive mode.

To give a more complete explanation we observe that a node can be locked _explicitly_ (by requesting it) or _implicitly_ (by appropriate explicit locks on the ancestors of the node) in one of five modes: IS, IX, S, SIX, X. However, the definition of implicit locks and the protocols for setting explicit locks have to be extended as follows:

A node is _implicitly granted in S_ mode to a transaction if _at least one_ of its parents is (implicitly or explicitly) granted to the transaction in S, SIX, or X mode. By induction that means that at least one of the node's ancestors must be explicitly granted in S, SIX, or X mode to the transaction.

A node is _implicitly granted in X_ mode if _all_ of its parents are (implicitly or explicitly) granted to the transaction in X mode. By induction, this is equivalent to the condition that all nodes in some cut set of the collection of all paths leading from the node to the roots of the graph are explicitly granted to the transaction in X mode and all ancestors of nodes in the cut set are explicitly granted in IX or SIX mode.

From Figure 2, a node is implicitly granted in IS mode if it is implicitly granted in S mode, and a node is implicitly granted in IS, IX, S, and SIX mode if it is implicitly granted in X mode.

The protocol for explicitly requesting locks on a DAG:

(a) Before requesting an S or IS lock on a node, request at least one parent (and by induction a path to a root) in IS (or greater) mode. As a consequence none of the ancestors along this path can be granted to another transaction in a mode incompatible with IS.

(b) Before requesting IX, SIX, or X mode access to a node, request all parents of the node in IX (or greater) mode. As a consequence all ancestors will be held in IX (or greater mode) and cannot be held by other transactions in a mode incompatible with IX (i.e. S, SIX, X).

(c) Locks should be released either at the end of the transaction (in any order) or in leaf to root order. In particular, if locks are not held to the end of transaction, one should not hold a lower lock after releasing its ancestors.

Given this protocol, one can prove the invariant condition:

If a node is granted implicitly or explicitly to a transaction then it is not granted to another transaction (implicitly or explicitly) in an incompatible mode.

Clearly if _all_ ancestors were locked by both the share mode lock protocol and by the exclusive mode lock protocol then the protocol would work. However, in the above protocol, only exclusive locks need lock all ancestors. The share lock protocol need lock only a single path to a single root. This is because the exclusive access protocol will get a cut set of all paths to all roots explicitly granted in X mode and all ancestors of the cut set are explicitly granted to the transaction in IX mode. Such a cut set will prevent _any_ paths from the node to the root from containing a node implicitly or explicitly granted to any other transaction in X or S mode.

To give an example using Figure 3, a sequential scan of all records in file F need not use an index so one can get an implicit share lock on each record in the file by:

```
 lock data base              with mode = IS
 lock area containing F      with mode = IS
 lock file F                 with mode = S
```

This gives implicit S mode access to all records in F. Conversely, to read a record in a file via the index I for file F, one need not get an implicit or explicit lock on file F:

```
 lock data base              with mode = IS
 lock area containing R      with mode = IS
 lock index I                with mode = S
```

This again gives implicit S mode access to all records in index I (in file F). In both these cases, _only one path was locked for reading_.

But to insert, delete, or update a record R in file F with index I one must get an implicit or explicit lock on all ancestors of R.

The first example of this section showed how an explicit X lock on a record is obtained. To get an implicit X lock on all records in a file simply lock the index and file in X mode, or lock the area in X mode. The latter examples allow bulk load or update of a file without further locking since all records in the file are implicitly granted in X mode.

Dynamic lock graphs:

Thus far we have pretended that the lock graph is static. However, examination of Figure 3 suggests otherwise. Areas, files, and indices are dynamically created and destroyed, and of course records are continually inserted, updated, and deleted. (If the data base is only read, then there is no need for locking at all.)

The lock protocol for such operations is nicely demonstrated by the implementation of index interval locks. Rather than being forced to lock entire indices or individual records, we would like to be able to lock all records with a certain index value; for example, lock all records in the bank account file with the location field equal to Napa. Therefore, the index is partitioned into lockable key value intervals. Each indexed record "belongs" to a particular index interval and all records in a file with the same field value on an indexed field will belong to the same key value interval (i.e. all Napa accounts will belong to the same interval). This new structure is depicted in figure 4.

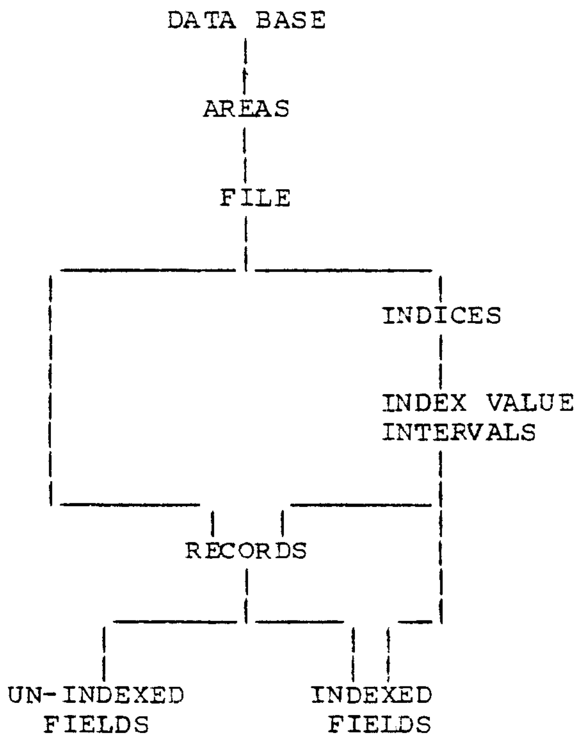

Figure 4. The lock graph with key interval locks.

The only subtle aspect of Figure 4 is the dichotomy between indexed and un-indexed fields and the fact that a key value interval is the parent of both the record and its indexed fields. Since the field value and record identifier (data base key) appear in the index, one can read the field directly (i.e. without touching the record.) Hence a key value interval is a parent of the corresponding field values. On the other hand, the index "points" via record identifiers to all records with that value and so is a parent of all records with that field value.

Since Figure 4 defines a DAG, the protocol of the previous section can be used to lock the nodes of the graph. However, it should be extended as follows. When an indexed field is updated, it and its parent record move from one index interval to another. So for example when a Napa account is moved to the St. Helena branch, the account record and its location field "leave" the Napa interval of the location index and "join" the St. Helena index interval. When a new record is inserted it "joins" the interval containing the new field value and also it "joins" the file. Deletion removes the record from the index interval and from the file.

The lock protocol for changing the parents of a node is:

(d) Before moving a node in the lock graph, the node must be implicitly or explicitly granted in X mode in both its old and its new position in the graph. Further, the node must not be moved in such a way as to create a cycle in the graph.

So to carry out the example of this section, to move a Napa bank account to the St. Helena branch one would:

```
 lock data base                 in mode = IX
 lock area containg accounts    in mode = IX
 lock accounts file             in mode = IX
 lock location index            in mode = IX
 lock Napa interval             in mode = IX
 lock St. Helena interval       in mode = IX
 lock record                    in mode = IX
 lock field                     in mode = X.
```

Alternatively, one could get an implicit lock on the field by requesting explicit X mode locks on the record and index intervals.

IV. SCHEDULING AND GRANTING REQUESTS:

Thus far we have described the semantics of the various request modes and have described the protocol that the requestors must follow. To complete the discussion we must explain how requests are scheduled and granted.

The set of all requests for a particular resource are kept in a queue sorted by some fair scheduler. By "fair" we mean that no particular transaction will be delayed indefinitely. First-in first-out is the simplest fair scheduler and we adopt such a scheduler for this discussion modulo deadlock preemption decisions.

The group of mutually compatible requests for a resource appearing at the head of the queue is called the granted group. All these requests can be granted concurrently. Assuming that each transaction has at most one request in the queue then the compatibility of two requests by different transactions depends only on the modes of the requests and may be computed using Table 1. Associated with the granted group is a group mode which is the supremum mode of the members of the group which is computed using Figure 2 or Table 3. Table 2 gives a list of the possible types of requests that can coexist in a group and the corresponding mode of the group.

Table 2. Possible request groups and their group mode. Set brackets indicate that several such requests may be present.

MODES OF REQUESTS	MODE OF GROUP
X	X
SIX, {IS}	SIX
S, {S}, {IS}	S
IX, {IX}, {IS}	IX
IS, {IS}	IS

Figure 5 depicts the queue for a particular resource, showing the requests and their modes. The granted group consists of six requests and has group mode IX. The next request in the queue is for S mode which is incompatible with the group mode IX and hence must wait.

```
**********************************
* GRANTED GROUP: GROUPMODE = IX *
* |IS|--|IX|--|IS|--|IS|--|IS|--*-|S|-|IS|-|X|-|IS|-|IX|
**********************************
```

Figure 5. The queue of requests for a resource.

When a new request for a resource arrives, the scheduler appends it to the end of the queue. There are two cases to consider, either someone is already waiting or all outstanding requests for this resource are granted (i.e. no one is waiting). If no one is waiting and the new request is compatible with the granted group mode then the new request can be granted immediately. Otherwise the new request must wait its turn in the queue and in the case of deadlock it may preempt some incompatible requests in the queue. (In Figure 5 all the requests decided to wait.) When a particular request leaves the granted group the group mode of the group may change. If the mode of the next request in the queue is compatible with the new mode of the granted group, then the waiting request is granted. In Figure 5, if the IX request leaves the group, then the group mode becomes IS which is compatible with S and so the S may be granted. The new group mode will be S and since this is compatible with IS mode the IS request following the S request may also join the granted group. This produces the situation depicted in Figure 6:

```
*****************************************
* GRANTED GROUP GROUPMODE = S           *
* |IS|--|IS|--|IS|--|IS|--|S|--|IS|--*-|X|-|IS|-|IX|
*****************************************
```

Figure 6. The queue after the IX request is released.

The X request of Figure 6 will not be granted until all the requests leave the granted group since it is not compatible with any mode.

Conversion:

A transaction might re-request the same object for several reasons: Perhaps it has forgotten that it already has access to the record, after all if it is setting many locks it may be simpler to just always request access to the record rather than first asking itself "have I seen this record before". The lock subsystem has all the information to answer this question and it seems wasteful to duplicate. Alternatively, the transaction may know it has access to the record, but want to increase its access mode (for example from S to X mode if it is in a read, test, and sometimes update scan of a file). So the lock subsystem must be prepared for re-requests by a transaction for a lock. We call such re-requests conversions.

When a request is found to be a conversion, the old (granted) mode of the requestor to the resource and the newly requested mode are compared using Table 3 to compute the new mode which is the supremum of the old and the requested mode (ref. Figure 2).

Table 3. The new mode given the requested and old mode.

	NEW MODE				
	IS	IX	S	SIX	X
IS	IS	IX	S	SIX	X
IX	IX	IX	SIX	SIX	X
S	S	SIX	S	SIX	X
SIX	SIX	SIX	SIX	SIX	X
X	X	X	X	X	X

So for example, if one has IX mode and requests S mode then the new mode is SIX.

If the new mode is equal to the old mode (note it is never "less" than the old mode) then the request can be granted immediately and the granted mode is unchanged. If the new mode is compatible with the group mode of the other members of the granted group (a requestor is always compatible with himself) then again the request can be granted immediately. The granted mode is the new mode and the group mode is recomputed using Table 2. In all other cases, the requested conversion must wait until the group mode of the other granted requests is compatible with the new mode. Note that this immediate granting of conversions over waiting requests is a minor violation of fair scheduling.

If two conversions are waiting each of which is incompatible with an already granted request of the other transaction then a deadlock exists and the already granted access of one must be preempted. Otherwise there is a consistent way of scheduling the waiting conversions: namely, grant a conversion when it is compatible with all other granted modes in the granted group. (Since there is no deadlock cycle this is always possible.)

The following example may help to clarify these points. Suppose the queue is:

```
******************************
* GROUPMODE = IS             *
*  |IS|---|IS|------------------------------
******************************
```

Figure 7. A simple queue.

Now suppose the first transaction wants to convert to X mode. It must wait for the second (already granted) request to leave the queue. If it decides to wait then the situation becomes:

```
******************************
*  GROUPMODE = IS            *
*   |IS->X|---|IS|-------------------
******************************
```

Figure 8. A conversion to X mode waits.

No new request may enter the granted group since there is now a conversion request waiting. In general, conversions are scheduled before new requests. If the second transaction now converts to IX, SIX, or S mode it may be granted immediately since this does not conflict with the granted (IS) mode of the first transaction. When the second transaction eventually leaves the queue, the first conversion can be made:

```
******************************
* GROUPMODE = X              *
*  |X|------------------------------------
******************************
```

Figure 9. The second transaction leaves and the conversion is granted.

However, if the second transaction tries to convert to exclusive mode one obtains the queue:

```
******************************
*   GROUPMODE = IS           *
*  |IS->X|---|IS->X|----------------------
******************************
```

Figure 10. Two conflicting conversions are waiting.

Since X is incompatible with IS (see Table 1), this situation implies that each transaction is waiting for the other to leave the queue (i.e. deadlock) and so one transaction must be preempted. In all other cases (i.e. when no cycle exists) there is a way to schedule the conversions so that no already granted access is violated.

Deadlock and lock thrashing:

Whenever a transaction waits for a request to be granted, it runs the risk of waiting forever in a deadlock cycle. For the purposes of deadlock detection it is important to know who is waiting for whom. The request queues give this information: Consider any waiting request R. there are two cases: If R is a waiting conversion then it is WAITING_FOR incompatible (with the new mode of R) granted requests in the granted group. If R is not a conversion it is WAITING_FOR all incompatible requests ahead of it in the queue. Given this WAITING_FOR relation computed for all waiting requests, there is no deadlock if and only if WAITING_FOR is acyclic.

The WAITING_FOR relation may change whenever a request or release occurs and when a conversion is granted. If a transaction can wait for at most one request at a time then the deadlock state can only change when some process decides to wait. In this special case, only waits require recomputation of the WAITING_FOR relation. If deadlock is improbable, deadlock testing can be done periodically rather than on each wait, thereby further reducing computational overhead.

One new request may form many cycles and each such cycle must be broken. When a cycle is detected, to break the cycle some granted or waiting request must be preempted. The lock scheduler should choose a minimal cost set of victims to preempt, so that all cycles are broken, undo all the changes to the data base made by the victims since the preempted resources were granted, and then preempt the resource and signal the victims that they have been backed up.

The issues discussed so far--lock scheduling, detecting and breaking deadlocks--are very low level scheduling decisions. They must be connected with a high level transaction scheduler which regulates the load on the system and regulates the entry and progress of transactions to prevent long waits, high probability of waiting (lock thrashing), and deadlock. By analogy, a page management system with only a low level page frame scheduler, which allocates and preempts page frames in a fairly naive way, is likely to produce page thrashing unless it is coupled with a working set scheduler which regulates the number and character of processes competing for page frames.

V. LOCK HIERARCHIES IN EXISTING SYSTEMS:

Some of the major enhancements of IMS/VS [2] are in the areas of locking for enhanced concurrency. With the program isolation feature, IMS/VS has a two level lock hierarchy: segment types (sets of records), and segment instances (records) within a segment type. Segment types may be locked in EXCLUSIVE (E) mode (which corresponds to our exclusive (X) mode) or in EXPRESS READ (R), RETRIEVE (G), or UPDATE (U) (each of which correspond to our notion of intention (I) mode). Segment instances can be locked in share or exclusive mode. Segment type requests are all made at transaction initiation, usually in intention mode. Segment instance locks are dynamically set as the transaction proceeds. In addition IMS/VS has user controlled share locks on segment instances (the *Q option) which allow other read requests but not other *Q or exclusive requests. IMS/VS has no notion of S or SIX locks on segment types (which would allow a scan of all members of a segment type concurrent with other readers but without the overhead of locking each segment instance). Since IMS/VS does not support S mode on segment types one need not distinguish the two intention modes IS and IX (see the section introducing IS and IX modes).

The ideas presented here were developed in the process of designing and implementing an experimental data base system at the IBM San Jose Research Laboratory. (We wish to emphasize that this system is a vehicle for research in data base architecture, and does not indicate plans for future IBM products.) A subsystem which provides the modes of locks herein described, plus the necessary logic to schedule requests and conversions, and to detect and resolve deadlocks has been implemented as one component of the data manager. The lock subsystem is in turn used by the data manager to automatically lock the nodes of its lock graph (see Figure 11). Users can be unaware of these lock protocols beyond the verbs "begin transaction" and "end transaction".

The data base is broken into several storage areas. Each area contains a set of relations (files), their indices, and their tuples(records) along with a catalog of the area. Each tuple has a unique tuple identifier (data base key) which can be used to quickly (directly) address the tuple. Each tuple identifier maps to a set of field values. All tuples are stored together in an area-wide heap to allow physical clustering of tuples from different relations. The unused slots in this heap are represented by an area-wide pool of free tuple identifiers (i.e. identifiers not allocated to any relation). Each tuple "belongs" to a unique relation, and all tuples in a relation have the same number and type of fields. One may construct an index on any subset of the fields of a relation. Tuple identifiers give fast direct access to tuples, while indices give fast associative access to field values and to their corresponding tuples. Each key value in an index is made a lockable object in order to solve the problem of "phantoms" [1] without locking the entire index. We do not explicitly lock individual fields or whole indices so those nodes appear in Figure 11 only for pedagogical reasons. Figure 11 gives only the "logical" lock graph, there is also a graph for physical page locks and for other low level resources.

As can be seen, Figure 11 is not a tree. Heavy use is made of the techniques mentioned in the section on locking DAG's. For example, one can read via tuple identifier without setting any index locks but to lock a field for update its tuple identifier and the old and new index key values covering the updated field must be locked in X mode. Further, the tree is not static, since data base keys are dynamically allocated to relations; field values dynamically enter, move around in, and leave index value intervals when records are inserted, updated, and deleted; relations and indices are dynamically created and destroyed within areas; and areas are dynamically allocated. The implementation of such operations observes the lock protocol presented in the section on dynamic graphs: When a node changes parents, all old and new parents must be held (explicitly or implicitly) in intention exclusive mode and the node to be moved must be held in exclusive mode.

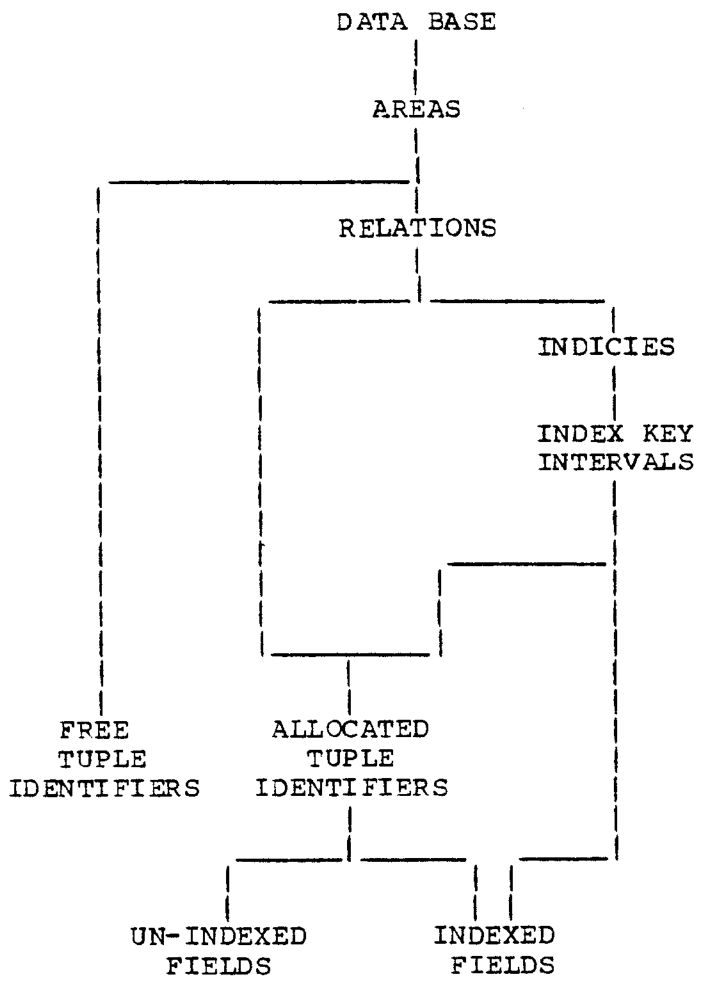

Figure 11. A lock graph.

VI. ACKNOWLEDGMENTS:

We gratefully acknowledge many helpful discussions with Jim Mehl and Brad Wade on how locking works in existing systems and how these results might be better presented. We are especially indebted to Paul McJones and Irving Traiger in this regard.

VII. REFERENCES:

[1] K.E. Eswaran, J.N. Gray, R.A. Lorie, I.L. Traiger, "On the notions of consistency and predicate locks," Technical Report RJ.1487, IBM Research Laboratory, San Jose, Ca., Nov. 1974.

[2] "Information management system virtual storage (IMS/VS). System application design guide," form no. SH20-9025, pp. 3.19-3.25, IBM Corp. 1974.