Modelling in Data Base Management Systems, G.M. Nijssen, (ed.)
North Holland Publishing Company, 1976

Granularity of Locks and Degrees of Consistency in a Shared Data Base

J.N. Gray, R.A. Lorie, G.R. Putzolu, I.L. Traiger

IBM Research Laboratory
San Jose, California

The problem of choosing the appropriate granularity (size) of lockable objects is introduced and the tradeoff between concurrency and overhead is discussed. A locking protocol which allows simultaneous locking at various granularities by different transactions is presented. It is based on the introduction of additional lock modes besides the conventional share mode and exclusive mode. A proof is given of the equivalence of this protocol to a conventional one.

Next the issue of consistency in a shared environment is analyzed. This discussion is motivated by the realization that some existing data base systems use automatic lock protocols which insure protection only from certain types of inconsistencies (for instance those arising from transaction backup), thereby automatically providing a limited degree of consistency. Four degrees of consistency are introduced. They can be roughly characterized as follows: degree 0 protects others from your updates, degree 1 additionally provides protection from losing updates, degree 2 additionally provides protection from reading incorrect data items, and degree 3 additionally provides protection from reading incorrect relationships among data items (i.e. total protection). A discussion follows on the relationships of the four degrees to locking protocols, concurrency, overhead, recovery and transaction structure.

Lastly, these ideas are compared with existing data management systems.

I. GRANULARITY OF LOCKS:

An important issue which arises in the design of a data base management system is the choice of lockable units, i.e. the data aggregates which are atomically locked to insure consistency. Examples of lockable units are areas, files, individual records, field values, and intervals of field values.

The choice of lockable units presents a tradeoff between concurrency and overhead, which is related to the size or granularity of the units themselves. On the one hand, concurrency is increased if a fine lockable unit (for example a record or field) is chosen. Such unit is appropriate for a "simple" transaction which accesses few records. On the other hand a fine unit of locking would be costly for a "complex" transaction which accesses a large number of records. Such a transaction would have to set and reset a large number of locks, incurring the computational overhead of many invocations of the lock subsystem, and the storage overhead of representing many locks. A coarse lockable unit (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 it would be desirable to have lockable units of different granularities coexisting in the same system.

This paper presents a lock protocol satisfying these requirements and discusses the related implementation issues of scheduling, granting and converting lock requests.

Hierarchical locks:

We will first assume that the set of resources to be locked is organized in a hierarchy. Note that this hierarchy is used in the context of a collection of resources and has nothing to do with the data model used in a data base system. 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.

```
DATA BASE
    |
    |
  AREAS
    |
    |
  FILES
    |
    |
 RECORDS
```

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. 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 thereby prevents 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 first lock all ancestors of R in intention mode and then to lock node R in exclusive or share mode. For example, using Figure 1, to lock a particular file one should 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 share) mode.

Access modes and compatibility:

We say that two lock requests for the same node by two different transactions are compatible if they can be granted concurrently. 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 but 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 simultaneously be granted the data base and some area and some file in intention mode. In this case their explicit locks on particular 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: the intention share mode only requests share or intention share locks at the lower nodes of the tree (i.e. never requests an exclusive lock below the intention share node), hence IS is compatible with S mode. 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. Rather to get the combined rights of share mode and intention exclusive mode one must obtain an X or SIX mode lock. (This issue is discussed in the section on rerequests 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, where for completeness we have also introduced the null mode (NL) which represents the absence of requests of a resource by a transaction.

	NL	IS	IX	S	SIX	X
NL	YES	YES	YES	YES	YES	YES
IS	YES	YES	YES	YES	YES	NO
IX	YES	YES	YES	NO	NO	NO
S	YES	YES	NO	YES	NO	NO
SIX	YES	YES	NO	NO	NO	NO
X	YES	NO	NO	NO	NO	NO

Table 1. Compatibilities among access modes.

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

NL: Gives no access to a node, i.e. represents the absence of a request of a resource.

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.)

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.)

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.)

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.)

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. Locking lower nodes in S or IS mode would give no increased access.)

IS mode is the weakest non-null 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.

```
         X
         |
         |
        SIX
         |
     ____|____
    |         |
    |         |
    S         IX
    |         |
    |_________|
         |
         |
         IS
         |
         |
         NL
```

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 in SIX or IX mode by the requestor.

(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 end of transaction, one should not hold a lock after releasing its ancestors.

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:

```
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.

<u>Directed acyclic graphs of locks</u>:

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, one should lock <u>all</u> 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.

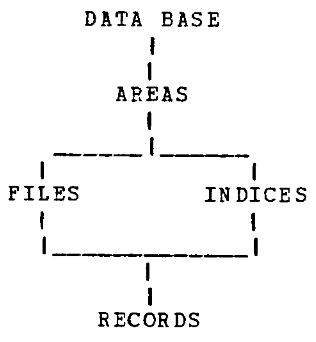

Figure 3. A non-hierarchical lock graph.

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 for DAG's 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, one should 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, one should 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.

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, <u>only one path was locked for reading</u>.

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 one can simply lock the index <u>and</u> 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.

<u>Proof of equivalence of the lock protocol</u>.

We will now prove that the described lock protocol is equivalent to a conventional one which uses only two modes (S and X), and which explicitly locks atomic resources (the leaves of a tree or sinks of a DAG).

Let G = (N,A) be a finite (directed acyclic) <u>graph</u> where N is the set of nodes and A is the set of arcs. G is assumed to be without circuits (i.e. there is no non-null path leading from a node n to itself). A node p is a <u>parent</u> of a node n and n is a <u>child</u> of p if there is an arc from p to n. A node n is a <u>source</u> (<u>sink</u>) if n has no parents (no children). Let SI be the set of sinks of G. An <u>ancestor</u> of node n is any node (including n) in a path from a source to n. A <u>node-slice</u> of a sink n is a collection of nodes such that each path from a source to n contains at least one node of the slice.

We also introduce the set of lock modes M = {NL,IS,IX,S,SIX,X} and the compatibility matrix C : MxM->{YES,NO} described in Table 1. Let c : mxm->{YES,NO} be the restriction of C to m = {NL,S,X}.

A lock-graph is a mapping L : N->M such that:

(a) if L(n) ∈ {IS,S} then either n is a source or there exists a parent p of n such that L(p) ∈ {IS,IX,S,SIX,X}. By induction there exists a path from a source to n such that L takes only values in {IS,IX,S,SIX,X} on it. Equivalently L is not equal to NL on the path.

(b) if L(n) ∈ {IX,SIX,X} then either n is a root or for all parents p1...pk of n we have L(pi) ∈ {IX,SIX,X} (i=1...k). By induction L takes only values in {IX,SIX,X} on all the ancestors of n.

The interpretation of a lock-graph is that it gives a map of the explicit locks held by a particular transaction observing the six state lock protocol described above. The notion of projection of a lock-graph is now introduced to model the set of implicit locks on atomic resources acquired by a transaction.

The projection of a lock-graph L is the mapping l: SI->m constructed as follows:

(a) l(n)=X if there exists a node-slice {n1...ns} of n such that L(ni)=X for each node in the slice.

(b) l(n)=S if (a) is not satisfied and there exists an ancestor a of n such that L(a) ∈ {S,SIX,X}.

(c) l(n)=NL if (a) and (b) are not satisfied.

Two lock-graphs L1 and L2 are said to be compatible if C(L1(n),L2(n))=YES for all n ∈ N. Similarly two projections l1 and l2 are compatible if c(l1(n),l2(n))=YES for all n ∈ SI.

Theorem:

If two lock-graphs L1 and L2 are compatible then their projections l1 and l2 are compatible. In other words if the explicit locks set by two transactions do not conflict then also the three-state locks implicitly acquired do not conflict.

Proof: Assume that l1 and l2 are incompatible. We want to prove that L1 and L2 are incompatible. By definition of compatibility there must exist a sink n such that l1(n)=X and l2(n) ∈ {S,X} (or vice versa). By definition of projection there must exist a node-slice {n1...ns} of n such that L1(n1)=...=L1(ns)=X. Also there must exist an ancestor n0 of n such that L2(n0) ∈ {S,SIX,X}. From the definition of lock-graph there is a path P1 from a source to n0 on which L2 does not take the value NL.

If P1 intersects the node-slice at ni then L1 and L2 are incompatible since L1(ni)=X which is incompatible with the non-null value of L2(ni). Hence the theorem is proved.

Alternatively there is a path P2 from n0 to the sink n which intersects the node-slice at ni. From the definition of lock-graph L1 takes a value in {IX,SIX,X} on all ancestors of ni. In particular L1(n0) ∈ {IX,SIX,X}. Since L2(n0) ∈ {S,SIX,X} we have C(L1(n0),L2(n0))=NO. Q.E.D.

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.)

We introduce the lock protocol for dynamic DAG's by example. Consider 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 contiguous range of index values; 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. In [1] such locks were called predicate locks and and an alternate (more general but less efficient) implementation was proposed.

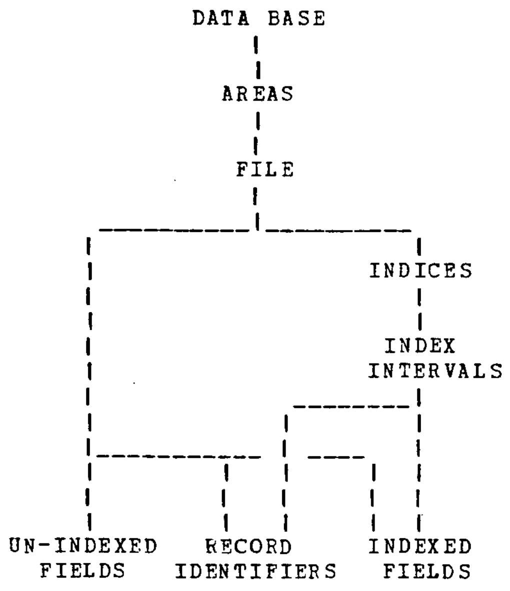

Figure 4. The lock graph with index interval locks.

The only subtle aspect of Figure 4 is the dichotomy between indexed and un-indexed fields. Since the indexed field value and record identifier (logical address) appear in the index, one can read the indexed field directly (i.e. without "touching" the record). Hence an index interval is a parent of the corresponding field values. Further, the index "points" via record identifiers to all records with that value and so is a parent of all such record identifiers. On the other hand, one can read and update un-indexed fields of the record without affecting the index and so the file is the only parent of such fields.

When an indexed field is updated, it and its record identifier move from one index interval to another. 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. index is not a lock ancestor of such fields.

Since Figure 4 defines a DAG, albeit a dynamic DAG, the protocol of the previous section can be used to lock the nodes of the DAG. However, the protocol should be extended as follows to handle dynamic changes to the lock graph:

(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.

Carrying out the example of this section, to move a Napa bank account to the St. Helena branch:

```
 lock data base                in mode = IX
 lock area containing 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.

Scheduling and granting requests:

Thus far we have described the semantics of the various request modes and have described the protocol which requestors must follow. To complete the discussion we discuss 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 five 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. (Alternatively the new request could be canceled. 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 first waiting 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.

Conversions:

A transaction might re-request the same resource 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 <u>other</u> 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 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 for a particular resource 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 <u>granted</u> (IS) mode of the first transaction. When the second transaction eventually leaves the queue, the first conversion can be made:

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

Figure 9. One 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 by transaction T. There are two cases: If R is a conversion, T is WAITING_FOR all transactions granted incompatible requests to the queue. If R is not a conversion, T is WAITING_FOR all transactions ahead of it in the queue granted or waiting for incompatible requests. Given this WAITING_FOR relation computed for all waiting transactions, 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 may 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 (synchronous calls to lock system), only waits require recomputation of the WAITING_FOR relation. If deadlock is improbable, deadlock testing can be done periodically rather than on each wait, 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 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.

II. DEGREES OF CONSISTENCY:

We now focus on how locks can be used to construct transactions out of atomic actions. The data base consists of entities which are related in certain ways. These relationships are best thought of as assertions about the data. Examples of such assertions are:
'Names is an index for Telephone_numbers.'
'The value of Count_of_x gives the number of employees in department x.'

The data base is said to be consistent if it satisfies all its assertions [1]. In some cases, the data base must become temporarily inconsistent in order to transform it to a new consistent state. For example, adding a new employee involves several atomic actions and the updating of several fields. The data base may be inconsistent until all these updates have been completed.

To cope with these temporary inconsistencies, sequences of atomic actions are grouped to form transactions. Transactions are the units of consistency. They are larger atomic actions on the data base which transform it from one consistent state to a new consistent state. Transactions preserve consistency. If some action of a transaction fails then the entire transaction is 'undone' thereby returning the data base to a consistent state. Thus transactions are also the units of recovery. Hardware failure, system error, deadlock, protection violations and program error are each a source of such failure.

If transactions are run one at a time then each transaction will see the consistent state left behind by its predecessor. But if several transactions are scheduled concurrently then locking is required to insure that the inputs to each transaction are consistent.

Responsibility for requesting and releasing locks can either be assumed by the user or be delegated to the system. User controlled locking results in potentially fewer locks due to the user's knowledge of the semantics of the data. On the other hand, user controlled locking requires difficult and potentially unreliable application programming. Hence the approach taken by some data base systems is to use automatic lock protocols which insure protection from general types of inconsistency, while still relying on the user to protect himself against other sources of inconsistencies. For example, a system may automatically lock updated records but not records which are read. Such a system prevents lost updates arising from transaction backup. Still, the user should explicitly lock records in a read-update sequence to insure that the read value does not change before the actual update. In other words, a user is guaranteed a limited automatic degree of consistency. This degree of consistency may be system wide or the system may provide options to select it (for instance a lock protocol may be associated with a transaction or with an entity).

We now present several _equivalent_ definitions of four consistency degrees. The first definition is an operational and intuitive one useful in describing the system behavior to users. The second definition is a procedural one in terms of lock protocols, it is useful in explaining the system implementation. The third definition is in terms of a trace of the system actions, it is useful in formally stating and proving properties of the various consistency degrees.

Informal definition of consistency:

An output (write) of a transaction is _committed_ when the transaction abdicates the right to 'undo' the write thereby making the new value available to all other transactions. Outputs are said to be _uncommitted or dirty_ if they are not yet committed by the writer. Concurrent execution raises the problem that reading or writing other transactions' dirty data may yield inconsistent data.

Using this notion of dirty data, the degrees of consistency may be defined as:

Definition 1:

Degree 3: Transaction T _sees degree 3 consistency_ if:
 (a) T does not overwrite dirty data of other transactions.
 (b) T does not commit any writes until it completes all its writes (i.e. until the end of transaction (EOT)).
 (c) T does not read dirty data from other transactions.
 (d) Other transactions do not dirty any data read by T before T completes.

Degree 2: Transaction T _sees degree 2 consistency_ if:
 (a) T does not overwrite dirty data of other transactions.
 (b) T does not commit any writes before EOT.
 (c) T does not read dirty data of other transactions.

Degree 1: Transaction T _sees degree 1 consistency_ if:
 (a) T does not overwrite dirty data of other transactions.
 (b) T does not commit any writes before EOT.

Degree 0: Transaction T _sees degree 0 consistency_ if:
 (a) T does not overwrite dirty data of other transactions.

Note that if a transaction sees a high degree of consistency then it also sees all the lower degrees.

Degree 0 consistent transactions commit writes before the end of transaction. Hence backing up a degree 0 consistent transaction may require undoing an update to an entity locked by another transaction. In this sense, degree 0 transactions are unrecoverable.

Degree 1 transactions do not committ writes until the end of the transaction. Hence one may undo (back up) an in-progress degree 1 transaction without setting additional locks. This means that transaction backup does not erase other transactions' updates. This is the principal reason one data management system automatically provides degree 1 consistency to all transactions.

Degree 2 consistency isolates a transaction from the uncommitted data of other transactions. With degree 1 consistency a transaction might read uncommitted values which are subsequently updated or are undone. In degree 2 no dirty data <u>values</u> are read.

Degree 3 consistency isolates the transaction from dirty relationships among values. Reads are <u>repeatable</u>. For example, a degree 2 consistent transaction may read two different (committed) values if it reads the same entity twice. This is because a transaction which updates the entity could begin, update and end in the interval of time between the two reads. More elaborate kinds of anomalies due to concurrency are possible if one updates an entity after reading it or if more than one entity is involved (see example below). Degree 3 consistency completely isolates the transaction from inconsistencies due to concurrency [1].

Each transaction can elect the degree of consistency appropriate to its function. When the third definition is given we will be able to state the consistency and recovery properties of such a system more formally.
Briefly:

> If one elects degree i consistency then one sees a degree i consistent state (so long as all other transactions run at least degree 0 consistent)
>
> If all transactions run at least degree 1 consistent, system backup (undoing all in-progress transactions) loses no updates of completed transactions.
>
> If all transactions run at least degree 2 consistent, transaction backup (undoing any in-progress transaction) produces a consistent state.

To give an example which demonstrates the application of these several degrees of consistency, imagine a process control system in which some transaction is dedicated to reading a gauge and periodically writing batches of values into a list. Each gauge reading is an individual entity. For performance reasons, this transaction sees degree 0 consistency, committing all gauge readings as soon as they enter the data base. This transaction is not recoverable (can't be undone). A second transaction is run periodically which reads all the recent gauge readings, computes a mean and variance and writes these computed values as entities in the data base. Since we want these two values to be consistent with one another, they must be committed together (i.e. one cannot commit the first before the second is written). This allows transaction undo in the case that it aborts after writing only one of the two values. Hence this statistical summary transaction should see degree 1. A third transaction which reads the mean and writes it on a display sees degree 2 consistency. It will not read a mean which might be 'undone' by a backup. Another transaction which reads both the mean and the variance must see degree 3 consistency to insure that the mean and variance derive from the same computation (i.e. the same run which wrote the mean also wrote the variance).

<u>Lock protocol definition of consistency</u>:

Whether an instantiation of a transaction sees degree 0, 1, 2 or 3 consistency depends on the actions of other concurrent transactions. Lock protocols are used by a transaction to guarantee itself a certain degree of consistency independent of the behavior of other transactions (so long as all transactions at least observe the degree 0 protocol).

The degrees of consistency can be procedurally defined by the lock protocols which produce them. A transaction locks its inputs to guarantee their consistency and locks its outputs to mark them as dirty (uncommitted).

For this section, locks are dichotomized as *share mode locks* which allow multiple readers of the same entity and *exclusive mode locks* which reserve exclusive access to an entity. (This is the "two mode" lock protocol. Its generalization to the "six mode" protocol of the previous section should be obvious.) Locks may also be characterized by their duration: locks held for the duration of a single action are called *short duration locks* while locks held to the end of the transaction are called *long duration locks*. Short duration locks are used to mark or test for dirty data for the duration of an action rather than for the duration of the transaction.

The lock protocols are:

Definition 2:

Degree 3: transaction T *observes degree 3 lock protocol* if:
(a) T sets a long exclusive lock on any data it dirties.
(b) T sets a long share lock on any data it reads.

Degree 2: transaction T *observes degree 2 lock protocol* if:
(a) T sets a long exclusive lock on any data it dirties.
(b) T sets a (possibly short) share lock on any data it reads.

Degree 1: transaction T *observes degree 1 lock protocol* if:
(a) T sets a long exclusive lock on any data it dirties.

Degree 0: transaction T *observes degree 0 lock protocol* if:
(a) T sets a (possibly short) exclusive lock on any data it dirties.

The lock protocol definitions can be stated more tersely with the introduction of the following notation. A transaction is *well formed with respect to writes* (*reads*) if it always locks an entity in exclusive (shared or exclusive) mode before writing (reading) it. The transaction is *well formed* if it is well formed with respect to reads and writes.

A transaction is *two phase* (*with respect to reads or updates*) if it does not (share or exclusive) lock an entity after unlocking some entity. A two phase transaction has a growing phase during which it acquires locks and a shrinking phase during which it releases locks.

Definition 2 is too restrictive in the sense that consistency will not require that a transaction hold all locks to the EOT (i.e. the EOT is the shrinking phase). Rather, the constraint that the transaction be two phase is adequate to insure consistency. On the other hand, once a transaction unlocks an updated entity, it has committed that entity and so cannot be undone without cascading backup to any transactions which may have subsequently read the entity. For that reason, the shrinking phase is usually deferred to the end of the transaction; thus, the transaction is always recoverable and all updates are committed together. The lock protocols can be redefined as:

Definition 2':

Degree 3: T is well formed
and T is two phase.

Degree 2: T is well formed
and T is two phase with respect to writes.

Degree 1: T is well formed with respect to writes
and T is two phase with respect to writes.

Degree 0: T is well formed with respect to writes.

All transactions are required to observe the degree 0 locking protocol so that they do not update the uncommitted updates of others. Degrees 1, 2 and 3 provide increasing system-guaranteed consistency.

Consistency of schedules:

The definition of what it means for a transaction to see a degree of consistency was given in terms of dirty data. In order to make the notion of dirty data explicit it is necessary to consider the execution of a transaction in the context of a set of concurrently executing transactions. To do this we introduce the notion of a schedule for a set of transactions. A schedule can be thought of as a history or audit trail of the actions performed by the set of transactions. Given a schedule the notion of a particular entity being dirtied by a particular transaction is made explicit and hence the notion of seeing a certain degree of consistency is formalized. These notions may then be used to connect the various definitions of consistency and show their equivalence.

The system directly supports entities and actions. Actions are categorized as begin actions, end actions, share lock actions, exclusive lock actions, unlock actions, read actions, and write actions. An end action is presumed to unlock any locks held by the transaction but not explicitly unlocked by the transaction. For the purposes of the following definitions, share lock actions and their corresponding unlock actions are additionally considered to be read actions and exclusive lock actions and their corresponding unlock actions are additionally considered to be write actions.

A transaction is any sequence of actions beginning with a begin action and ending with an end action and not containing other begin or end actions.

Any (sequence preserving) merging of the actions of a set of transactions into a single sequence is called a schedule for the set of transactions.

A schedule is a history of the order in which actions were executed (it does not record actions which were undone due to backup). The simplest schedules run all actions of one transaction and then all actions of another transaction,... Such one-transaction-at-a-time schedules are called serial because they have no concurrency among transactions. Clearly, a serial schedule has no concurrency induced inconsistency and no transaction sees dirty data.

Locking constrains the set of allowed schedules. In particular, a schedule is legal only if it does not schedule a lock action on an entity for one transaction when that entity is already locked by some other transaction in a conflicting mode.

An initial state and a schedule completely define the system's behavior. At each step of the schedule one can deduce which entity values have been committed and which are dirty: if locking is used, updated data is dirty until it is unlocked.

Since a schedule makes the definition of dirty data explicit, one can apply Definition 1 to define consistent schedules:

Definition 3:

A transaction runs at degree 0 (1, 2 or 3) consistency in schedule S if T sees degree 0 (1, 2 or 3) consistency in S. (Conversely, transaction T sees degree i consistency if all legal schedules run T at degree i consistency.)

If all transactions run at degree 0 (1,2 or 3) consistency in schedule S then S is said to be a degree 0 (1, 2 or 3) consistent schedule.

Given these definitions one can show:

Assertion 1:

(a) If each transaction observes the degree 0 (1, 2 or 3) lock protocol (Definition 2) then any legal schedule is degree 0 (1, 2 or 3) consistent (Definition 3) (i.e, each transaction sees degree 0 (1, 2 or 3) consistency in the sense of Definition 1).

(b) Unless transaction T observes the degree 1 (2 or 3) lock protocol then it is possible to define another transaction T' which does observe the degree 1 (2 or 3) lock protocol such that T and T' have a legal schedule S but T does not run at degree 1 (2 or 3) consistency in S.

In [1] we proved Assertion 1 for degree 3 consistency. That argument generalizes directly to this result.

Assertion 1 says that if a transaction observes the lock protocol definition of consistency (Definition 2) then it is assured of the informal definition of consistency based on committed and dirty data (Definition 1). Unless a transaction actually sets the locks prescribed by degree 1 (2 or 3) consistency one can construct transaction mixes and schedules which will cause the transaction to run at (see) a lower degree of consistency. However, in particular cases such transaction mixes may never occur due to the structure or use of the system. In these cases an apparently low degree of locking may actually provide degree 3 consistency. For example, a data base reorganization usually need do no locking since it is run as an off-line utility which is never run concurrently with other transactions.

Assertion 2:

If each transaction in a set of transactions at least observes the degree 0 lock protocol and if transaction T observes the degree 1 (2 or 3) lock protocol then T runs at degree 1 (2 or 3) consistency (Definitions 1, 3) in any legal schedule for the set of transactions.

Assertion 2 says that each transaction can choose its degree of consistency so long as all transactions observe at least degree 0 protocols. Of course the outputs of degree 0, 1 or 2 consistent transactions may be degree 0, 1 or 2 consistent (i.e. inconsistent) because they were computed with potentially inconsistent inputs. One can imagine that each data entity is tagged with the degree of consistency of its writer: Degree 0 entities are purple, degree 1 entities are red, degree 2 entities are yellow and degree 3 entities are green. The color of the outputs of a transaction is the minimum of the transaction's color and the colors of the entities it reads (because they are potentially inconsistent). Gradually the system will turn purple or red unless everyone runs with a high degree of consistency. If the transaction's author knows something about the systems structure which allows an apparently degree 1 consistent protocol to produce degree 3 consistent results then this color coding is pessimistic. But, in general a transaction must beware of reading entities tagged with degrees lower than the degree of the transaction.

Dependencies among transactions:

One transaction is said to depend on another if the first takes some of its inputs from the second. The notion of dependency is defined differently for each degree of consistency. These dependency relations are completely defined by a schedule and can be useful in discussing consistency and recovery.

Each schedule defines three relations: <, << and <<< on the set of transactions as follows. Suppose that transaction T performs action a on entity e at some step in the schedule and that transaction T' performs action a' on entity e at a later step in the schedule. Further suppose that T does not equal T'. Then:

```
 T <<< T' if   a is a write action and a' is a write action
          or   a is a write action and a' is a read  action
          or   a is a read  action and a' is a write action

 T << T'  if   a is a write action and a' is a write action
          or   a is a write action and a' is a read  action

 T < T'   if   a is a write action and a' is a write action
```

So degree 1 does not care about read dependencies at all. Degree 2 cares only about one kind of read dependency. And degree 3 ignores only read-read dependencies (reads commute). The following table is a notationally convenient way of seeing these definitions:

```
        <<<  :  W->W  |  W->R  |  R->W

        <<   :  W->W  |  W->R

        <    :  W->W
```

meaning that (for example) T <<< T' if T writes (W) something later read (R) by T' or written (W) by T' or T reads (R) something later written (W) by T'.

Let <* be the transitive closure of <, then define:
```
  BEFORE1(T) = {T'| T' <* T}
  AFTER1(T) = {T'| T  <* T'}.
```

The sets BEFORE2, AFTER2, BEFORE3 and AFTER3 are defined analogously from << and <<<.

The obvious interpretation for this is that each BEFORE set is the set of transactions which contribute inputs to T and each AFTER set is the set of transactions which take their inputs from T (where the ordering only considers dependencies induced by the corresponding consistency degree).

If some transaction is both before T and after T in some schedule then no serial schedule could give such results. In this case concurrency has introduced inconsistency. On the other hand, if all relevant transactions are either before or after T (but not both) then T will see a consistent state (of the corresponding degree). If all transactions dichotomize others in this way then the relation <* (<<* or <<<*) will be a partial order and the whole schedule will give degree 1 (2 or 3) consistency. This can be strengthened to:

Assertion 3:

A schedule is degree 1 (2 or 3) consistent if and only if the relation <* (<<* or <<<*) is a partial order.

The <, << and <<< relations are variants of the dependency sets introduced in [1]. In that paper only degree 3 consistency is introduced and Assertion 3 was proved for that case. In particular such a schedule is equivalent to the serial schedule obtained by running the transactions one at a time in <<< order. The proofs of [1] generalize fairly easily to handle assertion 1 in the case of degree 1 or 2 consistency.

Consider the following example:

```
          T1 LOCK    A
          T1 READ    A
          T1 UNLOCK  A
          T2 LOCK    A
          T2 WRITE   A
          T2 LOCK    B
          T2 WRITE   B
          T2 UNLOCK  A
          T2 UNLOCK  B
          T1 LOCK    B
          T1 WRITE   B
          T1 UNLOCK  B
```

In this schedule T2 gives B to T1 and T2 updates A after T1 reads A so T2<T1, T2<<T1, T2<<<T1 and T1<<<T2. The schedule is degree 2 consistent but not degree 3 consistent. It runs T1 at degree 2 consistency and T2 at degree 3 consistency.

It would be nice to define a transaction to see degree 1 (2 or 3) consistency if and only if the BEFORE and AFTER sets are disjoint in some schedule. However, this is not restrictive enough; rather one must require that the before and after sets be disjoint in all schedules in order to state Definition 1 in terms of dependencies. Further, there seems to be no natural way to define the dependencies of degree 0 consistency. Hence the principal application of the dependency definition is as a proof technique and for discussing schedules and recovery issues.

Relationship to transaction backup and system recovery:

A transaction T is said to be recoverable if it can be undone before 'EOT' without undoing other transactions' updates. A transaction T is said to be repeatable if it will reproduce the original output if rerun following recovery, assuming that no locks were released in the backup process. Recoverability requires system wide degree 1 consistency, repeatibility requires that all other transactions be at least degree 1 and that the repeatable transaction be degree 3.

The normal (i.e. trouble free) operation of a data base system can be described in terms of an initial consistent state S0 and a schedule of transactions mapping the data base into a final consistent state S3 (see Figure 11). S1 is a checkpoint state, since transactions are in progress, S1 may be inconsistent. A system crash leaves the data base in state S2. Since transactions T3 and T5 were in progress at the time of crash, S2 is potentially inconsistent. System recovery amounts to bringing the data base in a new consistent state in one of the following ways:

(a) Starting from state S2, undo all actions of transactions in-progress at the time of the crash.

(b) Starting from state S1 first undo all actions of transactions in progress at the time of the crash (i.e. actions of T3 and T4 before S1) and then redo all actions of transactions which completed before the crash (i.e. actions of T2 and T3 after S1).

(c) starting at S0 redo all transactions which completed before the crash.

Observe that (a) and (c) are degenerate cases of (b).

```
|      T1|-------------|  |          >              |
|         T2|------------|---|       <              |
|         T3|------------|------------>----|        |
|                        | T4|---|    <              |
|                        |   T5|----->-----|        |
S0                       S1         S2             S3
```

Figure 11. System states, S0 is initial state, S1 is checkpoint state, S2 is a crash and S3 is the state that results in the absence of a crash.

Unless all transactions run at least degree 1 consistency, system recovery may lose updates. If for example, T3 writes a record, r, and then T4 further updates r then undoing T3 will cause the update of T4 to r to be lost. This situation can only arise if some transaction does not hold its write locks to EOT.

(a) If all the transactions run in at least degree 1 consistency then system recovery loses no updates of complete transactions. However there may be no schedule which would give the same result because transactions may have read outputs of undone transactions.

(b) If all the transactions run in at least degree 2 then the recovered state is consistent and derives from the schedule obtatined from the original system schedule by deleting incomplete transactions. Note that degree 2 prevents read dependencies on transactions which might be undone by system recovery. of all the completed transactions results in a meaningful schedule.

(c) If a transaction is degree 3 consistent then it is reproducible.

Transaction crash gives rise to transaction backup which has properties analogous to system recovery.

Cost of degrees of consistency:

The only advantage of lower degrees of consistency is performance. If less is locked then less computation and storage is consumed. Further if less is locked, concurrency is increased since fewer conflicts appear. (Note that the granularity lock scheme of the first section was motivated by minimizing the number of explicit locks set.)

We will make some very crude estimates of the storage and computation resources consumed by the locking protocols as a function of the consistency degree. For the remainder of this section assume that all transactions are identical. Also assume that they do R reads and W writes (and hence set approximately R share mode locks and W exclusive mode locks). Further we assume that all the transactions run at the same consistency degree.

Each outstanding lock request consumes a queue element. The maximum per-transaction space for these queue elements as a function of consistency degrees is:

Table 4. Consistency degrees vs storage consumption.

CONSISTENCY DEGREE	STORAGE (in queue elements)
0	1
1	W
2	W+1
3	W+R

Observe that degrees 1 and 2 consume roughly the same amount of storage but that degree 3 consumes substantially more storage. This observation is aggravated by the fact that reads are typically ten times more common than writes.

The estimation of computation (CPU) overhead is much more subtle. We make only a crude estimate here. First one may consider the overhead in requesting and releasing locks. This is shown in Table 5 as a function of consistency degrees.

TABLE 5. Computational overhead vs degrees of consistency.

CONSISTENCY DEGREE	CPU (in calls to lock sys)
0	W
1	W
2	W+R
3	W+R

Table 5 indicates that the computational overhead of degrees 2 and 3 are comparable and are greater than the overhead of degrees 0 or 1. These pairs of degrees set the same locks, they just hold them for different durations.

Table 5 ignores the observation that some lock requests are trivially satisfied (the request is granted immediately) while others require a task switch and hence are quite expensive. The probability that a read lock will have to wait is proportional to the number of conflicting locks (write) currently granted. The probability that a write lock will have to wait is proportional to the number of conflicting (read or write) locks that are currently granted. Table 4 gives a guess of the maximum number of locks of each type held by each transaction. If there are 2*N+1 transactions one can multiply the entries of Table 4 by N to get an average number of locks held by all others. If a wait lock request is C+1 times as expensive as an immediately granted request and if P is the probability that two different requests are for the same resource then the relative computational costs are roughly computed:

degree 0 overhead:	W	cost of setting locks
	P*C*N*W	cost of waits
degree 1 overhead:	W	cost of writes
	P*C*N*W*W	cost of waits
degree 2 overhead:	W+R	cost of setting locks
	P*C*N*W*(W+1)	cost of write waits
	P*C*N*R*W	cost of read waits
degree 3 overhead:	W+R	cost of setting locks
	P*C*N*W*(W+R)	cost of waiting for writes
	P*C*N*R*W	cost of waiting for reads

TABLE 6. Computational overhead vs degrees of consistency.

CONSISTENCY DEGREE	CPU (in calls to lock sys)
0	W+P*C*N*W*(1)
1	W+P*C*N*W*(W)
2	W+R+P*C*N*W*(W+R+1)
3	W+R+P*C*N*W*(W+2*R)

To consider a specific example, a simple banking transaction does five reads (R=5) and six (W=6) writes. A transaction accesses a random account and there are millions of accounts so the probability of collision, P, is roughly .000001. Suppose there are one hundred transactions per second. A lock takes one hundred instructions and a wait requires five thousand instructions; hence, C=50. So the term P*C*N*W evaluates to 0.015. This implies that Table 5 gave a good estimate of the CPU overhead because the last term in Table 6 is miniscule compared to the term W+R. Of course this analysis is very sensitive to P and one must design the data base so that P takes on a very small value.

The striking thing about these estimates is that degree 2 and degree 3 seem to have similar computational overhead which seems to be substantially larger than the overhead of degree 0 or 1 consistency. We suspect that this conclusion would survive a more careful study of the problem.

ISSUE	DEGREE 0	DEGREE 1	DEGREE 2	DEGREE 3
COMMITTED DATA	WRITES ARE COMMITTED IMMEDIATELY	WRITES ARE COMMITTED AT EOT	SAME AS 1	SAME AS 1
DIRTY DATA	YOU DON'T UPDATE DIRTY DATA	0 AND NO ONE ELSE UPDATES YOUR DIRTY DATA	0,1 AND YOU DON'T READ DIRTY DATA	0,1,2 AND NO ONE ELSE DIRTIES DATA YOU READ
LOCK PROTOCOL	SET SHORT EXCL. LOCKS ON ANY DATA YOU WRITE	SET LONG EXCL. LOCKS ON ANY DATA YOU WRITE	1 AND SET SHORT SHARE LOCKS ON ANY DATA YOU READ	1 AND SET LONG SHARE LOCKS ON ANY DATA YOU READ
TRANSACTION STRUCTURE	WELL FORMED WRT WRITES	(WELL FORMED AND 2 PHASE) WRT WRITES	WELL FORMED (AND 2 PHASE WRT WRITES)	WELL FORMED AND TWO PHASE
CONCURRENCY	GREATEST: ONLY WAIT FOR SHORT WRITE LOCKS	GREAT: ONLY WAIT FOR WRITE LOCKS	MEDIUM: ALSO WAIT FOR READ LOCKS	LOWEST: ANY DATA TOUCHED IS LOCKED TO EOT
OVERHEAD	LEAST: ONLY SET SHORT WRITE LOCKS	SMALL: ONLY SET WRITE LOCKS	MEDIUM: SET BOTH KINDS OF LOCKS BUT NEED NOT STORE SHORT LOCKS	HIGHEST: SET AND STORE BOTH KINDS OF LOCKS
TRANSACT-ION BACKUP	CAN NOT UNDO WITHOUT CASCADING TO OTHERS	UN-DO ALL INCOMPLETE TRANSACTIONS IN ANY ORDER	UN-DO ANY INCOMPLETE TRANSACTIONS IN ANY ORDER	SAME AS 2
PROTECTION PROVIDED	LETS OTHERS RUN HIGHER CONSISTENCY	0 AND CAN'T LOSE WRITES	0,1 AND CAN'T READ BAD DATA ITEMS	0,1,2 AND CAN'T READ BAD DATA RELATIONSHIPS
SYSTEM RECOVERY TECHNIQUE	APPLY LOG IN ORDER OF ARRIVAL	APPLY LOG IN < ORDER	SAME AS 1: BUT RESULT IS SAME AS SOME SCHEDULE	2 AND SCHEDULE IS SERIAL
DEPENDENCIES	NONE	W->W	W->W W->R	W->W W->R R->W
ORDERING	NONE	< IS AN ORDERING OF THE TRANS-ACTIONS	<< IS AN ORDERING OF THE TRANS-ACTIONS	<<< IS AN ORDERING OF THE TRANS-ACTIONS

Table 7. Summary of consistency degrees.

III. LOCK GRANULARITY AND DEGREES OF CONSISTENCY IN EXISTING SYSTEMS:

IMS/VS with the program isolation feature [2] 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) [2, pages 3.18-3.27]. Segment instances can be locked in share or exclusive mode. Segment type locks are requested 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). In general, IMS/VS has a notion of intention mode and does implicit locking but does not recognize all the modes described here. It uses a static two level lock tree.

IMS/VS with the program isolation feature basically provides degree 2 consistency. However degree 1 consistency can be obtained on a segment type basis in a PCB (view) by specifying the EXPRESS READ option for that segment. Similarly degree 3 consistency can be obtained by specifying the EXCLUSIVE or UPDATE options. IMS/VS also has the user controlled share locks discussed above which a program can request on selected segment instances to obtain additional consistency over the degree 1 or 2 consistency provided by the system.

IMS/VS without the program isolation feature (and also the previous version of IMS namely IMS/2) doesn't have a lock hierarchy since locking is done only on a segment type basis. It provides degree 1 consistency with degree 3 consistency obtainable for a segment type in a view by specifying the EXCLUSIVE option. User controlled locking is also provided on a limited basis via the HOLD option.

DMS 1100 has a two level lock hierarchy [4]: areas and pages within areas. Areas may be locked in one of seven modes when they are OPENed: EXCLUSIVE RETRIEVAL (which corresponds to our notion of exclusive mode), PROTECTED UPDATE (which corresponds to our notion of share and intention exclusive mode), PROTECTED RETRIEVAL (which we call share mode), UPDATE (which corresponds to our intention exclusive mode), and RETRIEVAL (which is our intention share mode). Given this transliteration, the compatibility matrix displayed in Table 1 is identical to the compatibility matrix of DMS 1100 [3, page 3.59]. However, DMS 1100 sets only exclusive locks on pages within areas (short term share locks are invisibly set during internal pointer following). Further, even if a transaction locks an area in exclusive mode, DMS 1100 continues to set exclusive locks (and internal share locks) on the pages in the area, despite the fact that an exclusive lock on an area precludes reads or updates of the area by other transactions. Similar observations apply to the DMS 1100 implementation of S and SIX modes. In general, DMS 1100 recognizes all the modes described here and uses intention modes to detect conflicts but does not utilize implicit locking. It uses a static two level lock tree.

DMS 1100 provides level 2 consistency by setting exclusive locks on the modified pages and and a temporary lock on the page corresponding to the page which is "current of run unit". The temporary lock is released when the "current of run unit" is moved. In addition a run-unit can obtain additional locks via an explicit KEEP command.

The ideas presented 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 12). 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 12 only for pedagogical reasons. Figure 12 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 12 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.

The described system supports concurrently consistency degrees 1,2 and 3 which can be specified on a transaction basis. In addition share locks on individual tuples can be acquired by the user.

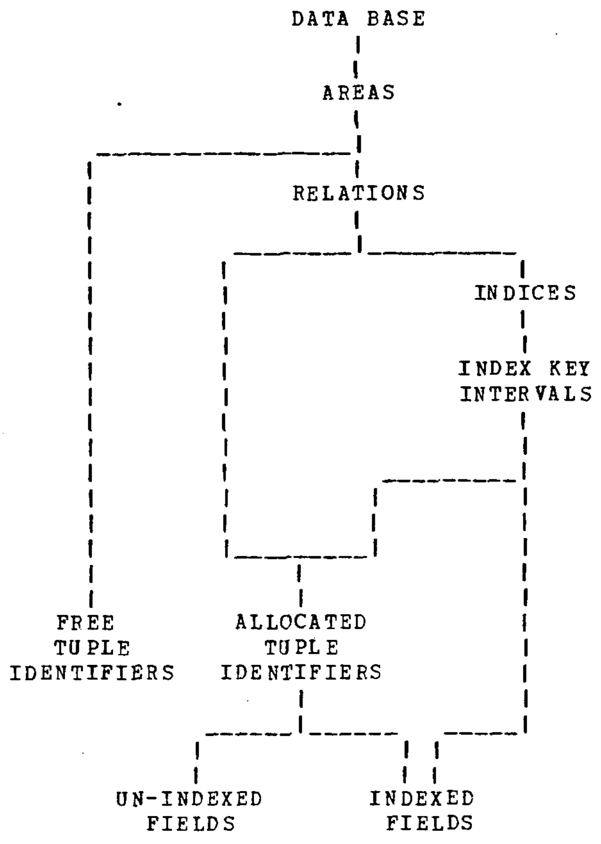

Figure 12. A lock graph.

ACKNOWLEDGMENT

We gratefully acknowledge many helpful discussions with Phil Macri, 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 in this regard.

REFERENCES

[1] K.P. 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. (to appear CACM).

[2] Information Management System Virtual Storage (IMS/VS). System Application Design Guide, Form No. SH20-9025-2, IBM Corp., 1975.

[3] UNIVAC 1100 Series Data Management System (DMS 1100). ANSI COBOL Field Data Manipulation Language. Order No. UP7908-2, Sperry Rand Corp., May 1973.