# 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

ABSTRACT: In the first part of the paper the problem of choosing the granularity (size) of lockable objects is introduced and the related 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.

In the second part of the paper 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 related to existing data management systems.

## I. GRANULARITY OF LOCKS:

An important problem which arises in the design of a data base management system is choosing the 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, intervals of field values, etc.

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/reset a large number of locks, hence incurring too many times the computational overhead of accessing the lock subsystem, and the storage overhead of representing a lock in memory. 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.

In the following a lock protocol satisfying these requirements will be described. Related implementation issues of scheduling, granting and converting lock requests are not discussed. They were covered in a companion paper [1].

Hierarchical locks:

We will first assume that the set of resources to be locked is organized in a hierarchy. Note that the concept of 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

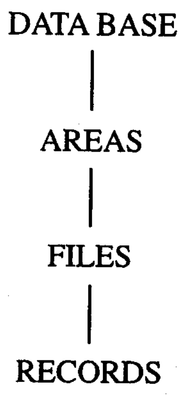

*Figure 1. A sample lock hierarchy.*

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 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 B in exclusive or share mode. So 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 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: 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). 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 [l] for a discussion of this point).

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. An equivalent approach would be to consider only four modes (IS, IX, S, X), but to assume that a transaction can request both S and IX lock privileges on a resource.

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.

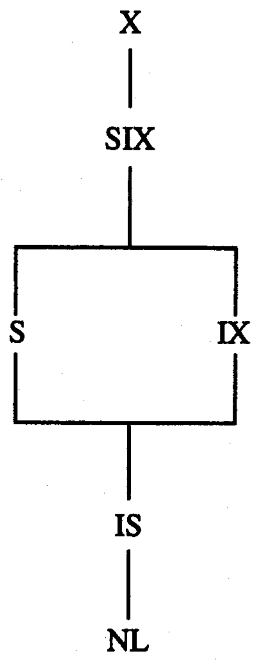

*Figure 2. The partial ordering of modes by their privileges.*

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:

I. 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 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, one should 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.

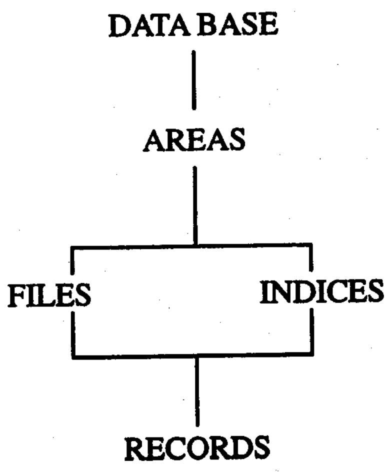

*Figure 3. A non-hierarchical lock graph.*

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. Alternatively, 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, 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, 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 one can 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.

Proof of equivalence of the lock protocol.

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 locks only atomic resources (leaves of a tree or a directed graph).

Let G = (N, A) be a finite (directed) graph 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 parent of a node n and n is a child of p if there is an arc from p to n. A node n is a source (sink) if n has no parents (no children). Let SI be the set of sinks of G. An ancestor of node n is any node (including n) in a path from a source to n. A node-slice of sink n is a collection of nodes such that each path from a source to n contains at least one of these nodes.

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. We will call c: mxm->{YES,NO} the restriction of C to m = {NL, S, X}.

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

(a) if L(n) ε {IS,S} then either n is a source or there exists a parent p of n such that L(p) ε (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) ε {IX, SIX, X} then either n is a root or for all parents p1 ... pk of n we have L(pi) ε {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 correspondingly acquired by a transaction.

The projection of a lock-graph L is the mapping 1: SI->m constructed as follows:

(a) l(n)=X if there exist a node-slice {n1...ns} of n such that L(ni) = X (i = 1 ... ns) .

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

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

Two lock-graphs Ll and L2 are said to be compatible if C(Ll (n), L2 (n)) = YES for all n ε N. Similarly two projections 11 and 12 are compatible if c (11 (n), 12 (n) ) = YES for all n ε SI.

We are now in a position to prove the following Theorem:

If two lock-graphs Ll and L2 are compatible then their projections11 and 12 are compatible. In other words if the explicit locks set by two transactions are not conflicting then also the three-state locks implicitly acquired are not conflicting.

Proof: Assume that 11 and 12 are incompatible. We want to prove that Ll and L2 are incompatible. By definition of compatibility there must exist a sink n such that 11(n) = X and 12 (n) ε {S,X} (or vice versa). By definition of projection there must exist a node-slice {nl ... ns} of n such that Ll (nl) =...= Ll (ns) = X. Also there must exist an ancestor n0 of n such that L2 (n0) ε {S, SIX, X}. From the definition of lock-graph there is a path Pl from a source to n0 on which L2 does not take the value NL.

If Pl intersects the node-slice at ni then Ll and L2 are incompatible since Ll (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 Ll takes a value in {IX, SIX, X} on all ancestors of ni. In particular Ll (n0) ε {IX, SIX, X}. Since L2(n0) ε {S, SIX, X} we have C (Ll(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.)

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.

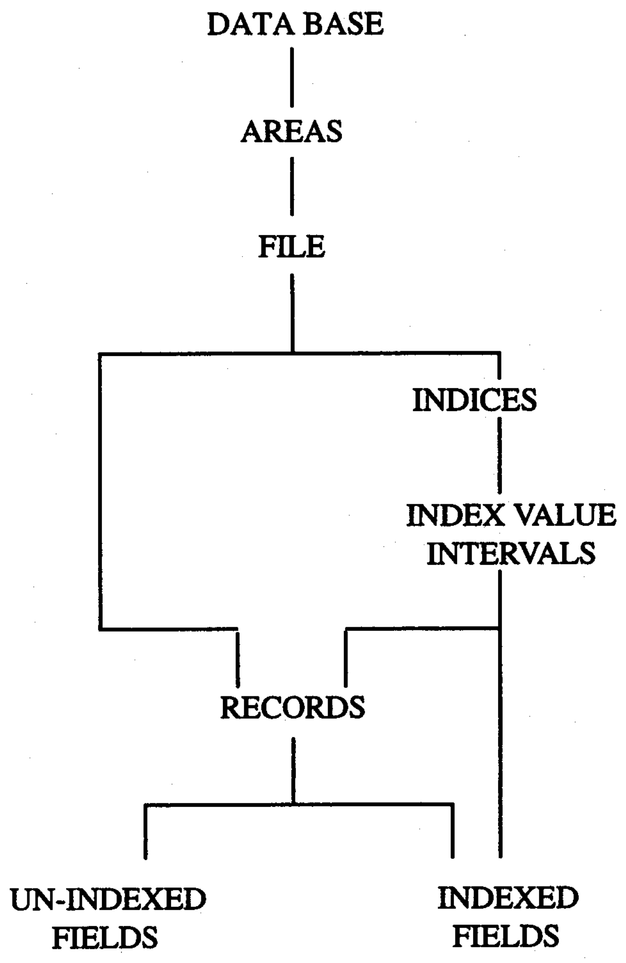

*Figure 4. The lock graph with key interval locks.*

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 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.

## II. DEGREES OF CONSISTENCY:

The data base consists of entities which are known to be structured in certain ways. This structure is 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 [2]. 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. The system may enforce the consistency assertions and undo a transaction which tries to leave the data base in an inconsistent state.

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 be either assumed by the user or delegated to the system. User controlled locking results in potentially fewer locks due to the users 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 inconsistencies, 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:

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.

These definitions have implications for transaction recovery. Transactions are dichotomized as recoverable transactions which can be undone without affecting other transactions, and unrecoverable transactions which cannot be undone because they have committed data to other transactions and to the external world. Unrecoverable transactions cannot be undone without cascading transaction backup to other transactions and to the external world (e.g. "unprinting" a message is usually impossible). If the system is to undo individual transactions without cascading backup to other transactions then none of the transaction's writes can be committed before the end of the transaction. Otherwise some other transaction could further update the entity thereby making it impossible to perform transaction backup without propagating backup to the subsequent transaction.

Degree 0 consistent transactions are unrecoverable because they commit outputs before the end of transaction. If all transactions see at least degree 0 consistency, then any transaction which is at least degree 1 consistent is recoverable because it does not commit writes before the end of the transaction. For this reason many data base systems require that all transactions see at least degree 1 consistency in order to guarantee that all transactions are recoverable.

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.

Degree 3 consistency isolates the transaction from dirty relationships among entities. 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.

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).

Lock protocol definition of consistency:

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 operationally 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). Degrees 0, 1 and 2 are important because of the efficiencies implicit in these protocols. Obviously, it is cheaper to lock less.

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. 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 so that the transaction is always recoverable and so that 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 originally 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 are executed (it does not record actions which are 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.

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.

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. 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 |

The following table is a notationally convenient way of seeing these definitions:

| | | | |
|---|---|---|---|
| <<< | W->V | 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 for << 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 [2]. 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 [2] 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 Tl and T2 updates A after Tl reads A so T2<Tl, T2<<Tl, T2<<<Tl and Tl<<<T2. The schedule is degree 2 consistent but not degree 3 consistent. It runs Tl 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:

As mentioned previously, system wide degree 1 consistency allows transaction backup and system recovery without lost updates (i.e. without affecting updates of transactions which are not being backed up). The transaction is unrecoverable after its first commit of an update (unlock) and so although degree I does not require it, the shrinking phase is usually deferred to the end of transaction so that the transaction is recoverable.

Given any current state and a time ordered log of the updates of transactions, one can return to a consistent state by un-doing any incomplete transactions (uncommitted updates). Given a checkpoint at time T0 and a log which records old and new values of entities up to time T0 + e, one can construct the most recent consistent state by undoing all updates which were made before time T0 but were not yet committed at time T0 + e; and by redoing all updates which were made and committed in the interval T0 to T0 + e. If the schedule (log) is degree 0 consistent then the actions can be re-done LOG order (skipping uncommitted updates). If the schedule (log) is degree 1 consistent then the actions can be sorted by transaction in <* order and recovery performed with the sorted log. The outcome of this process will be a state reflecting all the changes made by all transactions which completed before the log stopped.

However, degree 1 consistent transactions may read uncommitted (dirty) data. Transaction and system recovery may undo uncommitted updates. So if the degree 1 consistent transaction is re-run (i.e. re-executed by the system) in the absence of the undone transactions it may produce entirely different results than would be obtained if the transaction were blindly re-done (from the updates recorded in the log). If the system is degree 2 consistent then no transaction reads uncommitted data. So if the completed transactions are re-done in log order but in the absence of some undone (incomplete) transactions they will give exactly the same results as were obtained in the presence of the undone transactions. In particular, if the transactions were re-run in the order specified by the log but in the absence of the undone transactions the same consistent state would result.

| ISSUE | DEGREE 0 | DEGREE 1 | DEGREE 2 | DEGREE 3 |
|---|---|---|---|---|
| COMMITTED DATA | WRITES ARE COMMITTED IMMEDIATELY | WRITES ARE COMMITTED AT EOT | SAME | SAME |
| 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 see [ 1] | 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 INCOMPLETE TRANSACTIONS IN ANY ORDER | SAME | SAME |
| 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 | RERUN TRANSACTIONS IN <<< ORDER |
| 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 2. Summary of consistency degrees.*

## III. LOCK HIERARCHIES AND DEGREES OF CONSISTENCY IN EXISTING SYSTEMS:

IMS/VS with the program isolation feature [3] 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) [3 page 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 option. 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 [4, 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 DRS 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 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 5). 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 5 only for pedagogical reasons. Figure 5 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 5 is not a tree. Heavy use is made of the techniques mentioned in the section on locking DAGs. 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.