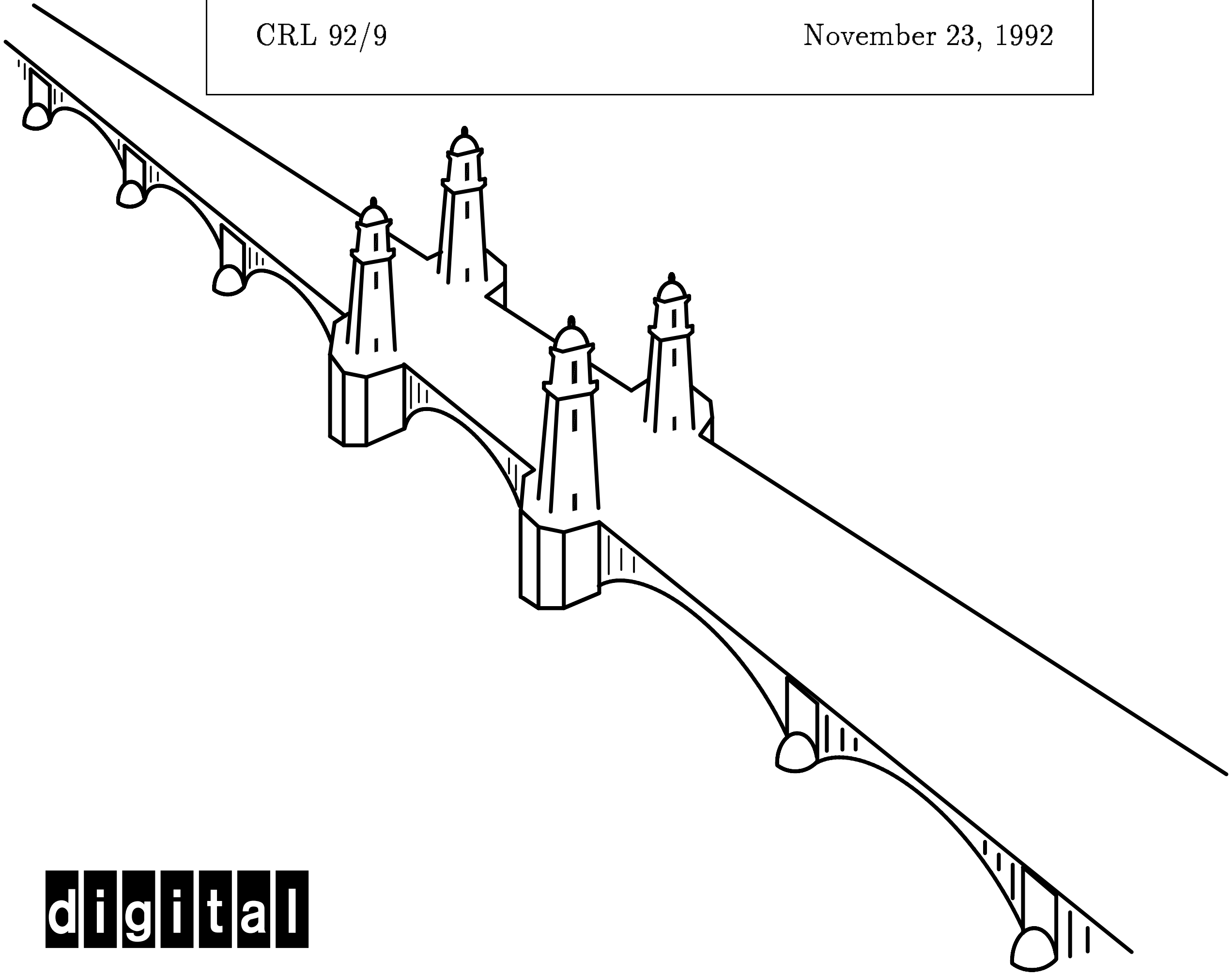
Private Lock Management

David Lomet

Digital Equipment Corporation
Cambridge Research Lab

CRL 92/9 November 23, 1992



CAMBRIDGE RESEARCH LABORATORY
Technical Report Series

Digital Equipment Corporation has four research facilities: the Systems Research Center and the Western Research Laboratory, both in Palo Alto, California; the Paris Research Laboratory, in Paris; and the Cambridge Research Laboratory, in Cambridge, Massachusetts.

The Cambridge laboratory became operational in 1988 and is located at One Kendall Square, near MIT. CRL engages in computing research to extend the state of the computing art in areas likely to be important to Digital and its customers in future years. CRL's main focus is applications technology; that is, the creation of knowledge and tools useful for the preparation of important classes of applications.

CRL Technical Reports can be ordered by electronic mail. To receive instructions, send a message to one of the following addresses, with the word **help** in the Subject line:

On Digital's EASYnet:	CRL::TECHREPORTS
On the Internet:	techreports@crl.dec.com

Cambridge Research Laboratory
One Kendall Square
Cambridge, Massachusetts 02139

Private Lock Management

David Lomet

Digital Equipment Corporation
Cambridge Research Lab

CRL 92/9 November 23, 1992

Abstract

For a data sharing database system, substantial coordination cost is incurred to cope with the global (distributed) locking needed by these systems. Lock **covering** is a way to permit component systems to perform private (local) locking. Two forms of covering locks are discussed, together with intention locks, needed to prevent covering violations. Intention locks give **permission** for lower level locking to be used, but require that this locking be global. The protocol between local lock manager and its principals is defined to permit information needed for local locking to be conveyed. Principals are notified when lock demotion may change local locks to global ones. New lock modes are defined that provide exclusion without being covering locks. These locks facilitate database cache management and private logical locking, permitting exclusion with high concurrency, and providing improved concurrency vs overhead trade-offs.

Keywords: concurrency control, locking, covering, private locking, multi-granularity locks, multi-level transactions, database cache management

1 Introduction

1.1 Data Sharing and Server Independence

There are two primary flavors of distributed database systems, each with its set of pros and cons.

Shared Nothing: Each subset (partition) of the data is accessed by only a single server (at a time) [16]. When accessing data from several partitions, messages are needed to orchestrate the execution of each transaction. Further, the two phase commit protocol is usually used to provide coordinated commit [2, 7]. With partitioned systems all updates are done by the single server for a partition, lock management for resources of the partition is done at this server, and locks are held by transactions running at the server. Other servers can acquire only copies of the data and cannot update the data directly.

Data Sharing: Multiple servers can access data of shared resources simultaneously [10, 14]. A user may exploit a single server to access all shared resources, with different users exploiting different servers. These servers need not exchange messages with other servers for request execution or for commit coordination when accessing shared data. However, simultaneous access requires low level coordination, including distributed locking [15]. Multiple servers may hold locks on a common set of resources, either when the resources are used in active transactions or when the resources are cached at servers. Server locks for resources that are only cached can be relinquished on request.

Server independence, by which we mean the ability of servers to execute with minimal coordination, is clearly desirable, but represents a problem for data sharing systems. With these systems, one is faced with the need for coordination protocols to control the management of the database cache, recovery and locking. Recently, advances have been made that permit increased server independence, and hence reduce the overhead for data sharing systems [6, 8, 10, 11, 13, 14]. In this paper, we focus on lock management, and on the principles involved in enabling lock management to be done with more independence. In particular, we explore how each server, by holding appropriate global locks on resources, can perform lock management privately on these resources for transactions that it executes. This permits a systematic reduction in the costs associated with distributed lock management.

1.2 Some Locking Fundamentals

Locks must $CONFLICT$ whenever principals must be prevented from operating on a resource simultaneously. Lock managers detect conflicts between locks requested by separate principals. The principal that has acquired a lock is permitted to perform certain accesses that are denied to principals that do not hold the lock. The lock manager does not grant a lock to a principal when that lock conflicts with a lock that is held by another principal.

Accesses need not be all or nothing. Lock $MODES$ exist that enable or prevent certain kinds of access to resources. A lock manager detects lock conflicts when the locks are on the same resource and in conflicting modes. Classically, there have been two lock modes, exclusive (X) which permits both reading and updating by ensuring that only a single principal can access a resource guarded by an X lock; and share (S) which permits only reading but allows multiple readers to access a resource guarded by an S lock.

DEFINITION:*Lock L_1 CONFLICTS with lock L_2 if it is not possible for two principals to hold the locks on the same resource simultaneously. We say that lock mode m_1 CONFLICTS with lock mode m_2 if locks of those modes conflict when held on the same resource by different principals. Further, the set of lock modes that CONFLICT with a given lock mode is*

$$CONFLICTS[m_1] = \{m_2 \mid m_1 \ CONFLICTS \ m_2\} \tag{1}$$

For the lock modes above, $CONFLICTS(X) = \{X, S\}$ while $CONFLICTS(S) = \{X\}$. Two lock modes, m_1 and m_2, are compatible if they do not conflict. $COMPAT(m)$ is the set of lock modes compatible with m. $COMPAT(m) = ALL - CONFLICT(m)$. Lock modes can be ordered based on the sizes of their conflict sets.

DEFINITION:*Lock mode m_1 is STRONGER than lock mode m_2 if*

$$CONFLICTS(m_2) \subseteq CONFLICTS(m_1). \tag{2}$$

Hence, X is $STRONGER$ than S. A lock on a resource in a strong mode is said to be $STRONGER$ than a lock in a weaker mode on the same resource. $STRONGER$ is transitive and forms a partial ordering among lock modes. When a principal changes a lock on a resource from one lock mode to another, this is called lock conversion. We call conversion from a strong mode to a weaker one lock **demotion** and conversion from a weak mode to a stronger one lock **promotion**.

1.3 Lock Covering

Locking is usually conservative, preventing more accesses than are strictly required for correct serializable execution. This is sometimes done to prevent deadlocks, to facilitate recovery, etc. Conservative locking results in the resources being locked with larger granularities and the locking modes being more restrictive than needed. This is acceptable so long as the concurrency permitted by the locks is sufficient. Only concurrency is sacrificed. What is required is that a lock be sufficiently strong to prevent accesses that would compromise serializability. Any stronger lock that "covers" a sufficient lock is also acceptable.

DEFINITION: *Lock L_1 $COVERS$ lock L_2 if every lock that conflicts with L_2 also conflicts with L_1. Further, a set of locks $\{L_1\}$ $COVERS$ a set of locks $\{L_2\}$ if every lock that conflicts with some lock in $\{L_2\}$ also conflicts with some lock in $\{L_1\}$.*

The simple form of lock covering is direct covering.

DEFINITION:*Lock L_1 directly $COVERS$ lock L_2 if the mode of L_1 is $STRONGER$ than the mode of L_2 and both locks are on the same resource. We then say that m_1 (the lock mode of L_1) $DCOVERS$ m_2 (the lock mode of L_2). Further, the set of lock modes $DCOVER$'d by a lock mode is*

$$DCOVERS[m_1] = \{m_2 \mid m_1 \ DCOVERS \ m_2\} \tag{3}$$

It is the set of lock modes that are weaker than m_1. The relation $DCOVERS$ is transitive.

Hence, a lock on a resource in mode X directly $COVERS$ locks on the same resource of modes S or X. Further, a lock in mode S directly $COVERS$ only other S mode locks on the same resource.

It is important to note that conventional lock managers only detect conflicts among locks on a single resource. Hence, direct covering is the only covering of which a lock manager is usually aware.

1.4 Distributed Lock Management

A transaction executing at a server makes requests for data to that server. That server then needs to grant a lock on that data to the transaction. To do that when the data accessed is shared data requires that the server coordinate the request with other servers. The reason is that other servers may be using the data, either in transactions or by having the data cached. Conflicting use of the data must be prevented. Thus, a server must be prepared to make its lock requests visible to other servers. This is called distributed lock management and it is the traditional problem for data sharing systems.

Preventing conflicting accesses among servers does not require that all locks be visible globally. Covering can be exploited to partition locking responsibility between multiple lock managers on different servers. A server can acquire a strong covering lock on a resource and can then mediate and grant lock requests that it receives for this resource so long as its lock covers the locks being requested.

Direct covering provides some leverage for private lock management. However, larger possibilities arise with multi-granularity locking [3], which can be used to permit a lock on a large granule, e.g. file, to be exposed while the locks on pages of the file are managed privately [14].

We explore the principles involved with private versus global management of locks in data sharing DBMSs in the remainder of the paper. Section 2 defines covering for multi-granularity locks and how covering locks interact with intention locks. How multiple local lock managers function so as to support private locking is the subject of section 3. In section 4, we generalize covering to work with logical locking and cache management. Section 5 provides a discussion of our approach and its effectiveness.

2 Covering for Multi-granularity Locks

2.1 Resource Covering

It is possible for a lock or locks to cover another lock even when the locks are not on the same resource. The classic example of this is multi-granularity locking [3].

EXAMPLE: *The purpose of a lock(X or S) on a large granule, e.g. a file, that contains other smaller granularity resources, e.g. pages, is for the file lock to also lock the pages of the file in the same mode. Thus, a file lock of X or S should cover X or S locks on the pages. This is an instance of a tree locking granularity hierarchy.*

More generally, a set of resources R_1 can jointly guard resource r_2 such that when resources in R_1 are locked, locks on resource r_2 are "covered", where r_2 is not a member of R_1. A file may have multiple secondary indexes, where it is desired to cover locks on the records of the file through the use of locks on entries in the indexes. Thus, resources can have more than one "ancestor" and the multi-granularity hierarchy is a directed acyclic graph or DAG. We denote the multi-granularity locking hierarchy, whether tree or DAG as the MGH. Then, we have

DEFINITION: *A set of locks $\{L_1\}$ on resources in set R_1 resource $COVERS$ a lock L_2 on resource r_2 where R_1 guards r_2 in a multi-granularity lock hierarchy if $\{L_1\}$ $COVERS$ L_2. If all locks in $\{L_1\}$ have the same lock mode m_1, then we say that m_1 $RCOVERS(R_1, r_2)$ m_2. We also define*

$$RCOVERS(R_1, r_2)[m_1] = \{m_2 \mid m_1 \; RCOVERS(R_1, r_2) m_2\} \qquad (4)$$

If m_1 $RCOVERS(R_1, r_2)$ m_2 for all r_2 in R_2 and m_2 $RCOVERS(R_2, r_3)$ m_3 then m_1 $RCOVERS(R_1, r_3)$ m_3. This is the transitivity form for $RCOVERS$.

By the transitivity of $DCOVERS$, if m_1 $DCOVERS$ m_2 then

$$RCOVERS(R_1, r_2)[m_2] \subseteq RCOVERS(R_1, r_2)[m_1]. \qquad (5)$$

Note that the lock modes for R_1 may be different from the lock modes for r_2. And there is not necessarily a $DCOVER$ relation between the lock modes for R_1 and those for r_2.

2.2 Intention Locks

Usually, a lock manager will not be aware of resource covering because it will not detect conflicts arising from locks on different resources. In order for a principal P_1 to lock a resource r_2 for concurrency control without first acquiring the covering locks on R_1, requires some care. No other principal P_2 can be permitted to acquire resource $COVER$ing locks on R_1 because P_2 would then believe he was entitled to access r_2 without any locking at r_2. Hence, P_2's accessing of r_2 would not be prevented by P_1's locks and conflicting accesses would not be detected.

Enforcing resource covering is thus a function of the locking protocol. What one has to ensure is that no conflicting locks, including the implicit locks [5] will be concurrently held. Implicit locks are those that are covered by currently held locks, and hence are not materialized in the lock manager as locks whose conflicts can be detected. Thus, there must be at least one resource at which conflicts are materialized as explicit locks.

Since the purpose of covering is to avoid exposing the covered locks, and indeed any locks at the guarded resource whose locks are to be covered, we require that those taking out explicit locks on r_2 first take out one or more locks on resources in R_1. Applying this pervasively requires that resources in the multi-granularity hierarchy be locked in descending order.

The locks acquired higher in the multi-granularity hierarchy are called *intention* locks. Intention locks in R_1 $PERMIT$ locking to occur at r_2 without violating resource covering by conflicting at at least one resource in R_1 with a lock needed in order to resource cover r_2. Intention locks themselves need not cover any locks. They must merely conflict with locks that do and hence prevent others from acquiring resource covering locks.

We again simplify by requiring that all locks on resources in R_1 have the same lock mode. We then have:

DEFINITION: *Lock mode m_1 $PERMITS(R_1, r_2)$ m_2 if a lock in mode m_1 on a resource in R_1 serves as an intention lock on resources in R_1 for a lock in mode m_2 on r_2. Further, as before,*

$$PERMITS(R_1, r_2)[m_1] = \{m_2 \mid m_1 \; PERMITS(R_1, r_2) \; m_2\} \tag{6}$$

As with covering, if a lock mode m_2 is in $PERMITS(R_1, r_2)[m_1]$, and if

Lock Modes	IS	IX	S	SIX	X
IS	x	x	x	x	
IX	x	x			
S	x		x		
SIX	x				
X					

Table 1: Lock Mode Compatibility for Multi-granularity Locking

m_2 $DCOVERS$ m_3, then m_3 is in $PERMITS(R_1, r_2)[m_1]$. Further,

$$RCOVERS(R_1, r_2)[m_1] \subseteq PERMITS(R_1, r_2)[m_1]. \tag{7}$$

This says that if a lock mode on R_1 is strong enough to cover locks on r_2, then it is strong enough to permit them as well.

For the classical multi-granularity locking, there are two pure intention locks, IX which permits X and S locking of guarded resources, and IS which only permits S locking. In addition, the lock mode SIX is also defined, which is a lock which provides shared access to the resource on which it is held, and permits X locking on finer grained resources within. Table 1 defines the lock compatibilities of multi-granularity lock modes.

Because a lock $DCOVERS$ another does not mean that it $RCOVERS$ that lock for any arguments to $RCOVERS$. Thus, IS $DCOVERS$ IS for the same resource, but IS does not $RCOVER$ IS. An IS lock at each resource must be acquired explicitly if only preceded by other IS locking at guarding resources and hence must be exposed. There is no resource covering provided by an IS lock.

2.3 Determining Lock Mode Conflicts

At the heart of defining lock modes is the need to understand the constraints imposed by covering and intention locking. It is clearly not sufficient to simply assert that some locks are covering locks and others are intention locks. It is necessary to define lock modes such that the conflicts between locks with these lock modes provide the desired protection.

Thus, when defining locks on resources in R_1, one needs to define lock mode conflicts that satisfy the following:

CONFLICT CONSTRAINT: *Lock mode m_1 must $CONFLICT$ with lock mode m_2 on resources in R_1 if any of the following are true:*

1. $RCOVERS(R_1, r_2)[m_1]$ $CONFLICTS$ $RCOVERS(R_1, r_2)[m_2]$
2. $RCOVERS(R_1, r_2)[m_1]$ $CONFLICTS$ $PERMITS(R_1, r_2)[m_2]$
3. $PERMITS(R_1, r_2)[m_1]$ $CONFLICTS$ $RCOVERS(R_1, r_2)[m_2]$

Note that if $PERMITS(R_1, r_2)[m_1] CONFLICTS PERMITS(R_1, r_2)[m_2]$, that this has no impact on whether m_1 and m_2 conflict.

Thus, holding a lock L_1 at R_1 covers a lock L_2 at r_2 by preventing locks on R_1 that either permit or cover locks on r_2 that conflict with L_2. Holding L_1 ensures that no locking is needed at r_2 because all conflicting locks are stopped by it at R_1. Holding an intention lock L_1 at R_1 permits a lock L_2 at r_2 by preventing locks at R_1 that cover locks at r_2 that conflict with L_2. Intention locks, and covering locks for locks at r_2 that do not conflict with L_2 are not prevented. L_1 ensures that conflicts will be detected at r_2 rather than being subsumed by conflicts at R_1.

EXAMPLE: *The compatability matrix for multi-granularity locking was given in Table 1. Recall that the same lock modes are defined at each level of the resource hierarchy. Thus, we can drop the resource parameters of $RCOVERS$ and $PERMITS$ below. Then we have the following:*

- $RCOVERS[X] = \{X, SIX, S, IX, IS\} = PERMITS[X]$
- $RCOVERS[SIX] = \{S, IS\}$
- $PERMITS[SIX] = \{X, S, SIX, IX, IS\}$
- $RCOVERS[S] = \{S, IS\} = PERMITS[S]$
- $PERMITS[IX] = \{X, S, SIX, IX, IS\}$
- $PERMITS[IS] = \{S, IS\}$
- $RCOVERS[IX] = RCOVERS[IS] = \phi$

Thus, $RCOVERS$ and $PERMITS$ constrain lock conflicts. IS and IX can be compatible as neither covers any locks. On the other hand, IX must conflict with S because IX $PERMITS$ X, S $RCOVERS$ S, and S $CONFLICTS$ X. Since IX $PERMITS$ X, IX $CONFLICTS$ with modes that $RCOVER$ any lock. Similarly, SIX $CONFLICTS$ with S since SIX $PERMITS$ X, and S $RCOVERS$ S.

We can also use lock mode conflicts on R_1, and what lock modes are covered on r_2, to derive $PERMITS(R_1, r_2)[m_1]$ for any lock mode m_1 on resource set R_1. Mode m_1 $PERMITS$ any lock on r_2 not $RCOVER$'d by any other lock in a conflicting mode when m_1 is held. Since IX is compatible only with IS and IX, which $RCOVERS$ nothing, it permits everything. IS is compatible with $\{IX, IS, S, SIX\}$, which $RCOVERS$ $\{S, IS\}$. Hence, everything in $PERMITS(R_1, r_2)[IS]$ must be compatible with $\{IS, S\}$, which is this set itself.

2.4 Protocols for Multi-Granularity DAGs

Multi-granularity locking need not be restricted only to tree hierarchies. The MGH can also be a directed acyclic graph or DAG. There is only one protocol when the MGH is a tree. The resources are locked in tree order from the root of the tree down, and unlocked in the reverse order. An explicit lock is never held on a resource without a lock also being held on its parent.

The same "style" of protocol is needed for an MGH DAG, but the multiple parents of DAGs adds a complication. Conflicts required by covering and intention locking must be explicit at at least one ancestor in the MGH in the form of locks with conflicting modes. So locks on multiple parents may be required. Quorum algorithms attack this problem very generally [1]. They guarantee that sets constituting quorums for conflicting activities have non-null intersections.

In our case, locks on parents in the intersections will expose the conflicts. Each lock on a parent resource is assigned a weight. Quorums are defined so that each activity requires some weighted vote. The sum of the weights of the quorums for conflicting activities exceeds the sum of the weights of the parent locks. This forces conflicting operations to need conflicting locks at at least one parent. These locks on parents need not be explicit locks. A parent can be locked with an implicit lock resulting from its being covered by locks on its ancestors.

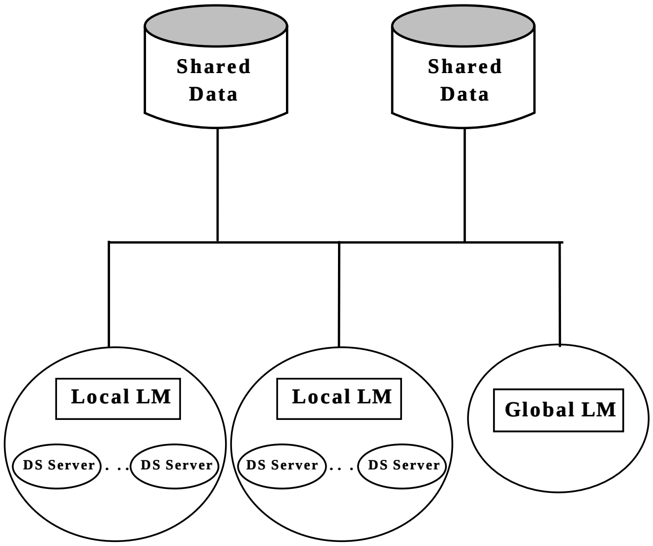


Figure 1: Data sharing system in which local LMs cooperate with a global LM to efficiently handle concurrency control. Data sharing database servers (DS Servers) use the Local LMs as their owning LMs.

3 Local Lock Managers

3.1 Private Locking

A local lock manager (LM) services lock requests from some subset of the principals, e.g. the clients of one server. It coordinates its access to shared resources (among other local LMs) by making lock requests to the global LM(s). An architectural picture of such a system is given in Figure 1.

Private locking is where requests for locks that are received by a local LM are handled by the local LM itself, without the need for communication with a global LM. Private locking local LMs can yield a dramatic reduction in lock overhead while preserving concurrency.

Usually, LMs are simple conflict detectors where the conflict is based on the detection of locks with conflicting lock modes for a given resource. The interdependencies between resources that occurs in multi-granularity locking, are typically handled by principals. Here, we describe how to realize private locking local LMs that do not know the specifics of the MGH. This requires that principals must know the MGH, observe its protocol, and "advise" the LM about it.

A local LM must hold locks at global LMs that cover the local locks acquired by any of its principals. Only then can an LM be sure that implicit and hence undetected lock conflicts do not occur due to the distributed nature of the locking. There are a number of ways that an LM can hold global covering locks on the resources that it manages:

1. for a system with only one LM, this LM implicitly has a permanent exclusive lock on ALL resources.

2. for a partitioned distributed system, each LM manages the locks of a partition and implicitly has a permanent exclusive lock on the partition.

3. for a data sharing system with multiple LMs, the locks held by each LM can change over time and are not necessarily exclusive locks. Hence, we need to be quite explicit about how locks are managed. This is discussed below.

3.2 Owners and Holders

For each resource, a principal needs to direct lock requests to a system component that can play the role of an LM. We call this LM the *owning* LM or simply the *owner*. The owner may grant the lock to any of several principals. A principal that holds a lock is called the current *holder* of the lock. An owner is responsible for keeping track of who currently holds locks on its owned resources and for detecting lock conflicts. The owner must also prevent deadlock in some manner, e.g. detection or time-out.

In a data sharing system with multiple LMs, ownership is context dependent. A local LM on a node acts as owner for all locks in so far as its principals (processes or transactions executing on that node) are concerned. A local LM does not permanently hold resources. Before it grants a request for one of its "owned" resources, it must acquire a covering lock on the resource at the global level of the system, i.e. from the global owning LM, hence becoming the lock holder at that level. Thus, it is the local LM, not its principals, that holds locks at the global system level and then owns them for its local principals. This kind of hold/own configuration can occur at multiple levels of a system.

While who holds a lock can change rather rapidly, which LM is the owning LM for any given principal is usually relatively static. This permits principals to readily know the LM to which a lock request should be made for some resource. There are a number of ways that ownership might be handled at the global level in a distributed system. Two examples are:

- A single global LM owns all locks in the system. Then, any local LM would ask the global LM for the lock.
- Each local LM owns some known subset of all the locks in the system and plays the role of a global LM for those locks. This subset may change with time, but it does so slowly, e.g. when an owner LM crashes. Other LMs can then learn to which LM requests should be directed.

3.3 Local and Global Locks

Locks requested by principals of a local LM that are covered by global locks already held by the LM can be granted locally. If the local LM does not have a covering lock for the locally requested lock, then it must acquire it. Once the local LM holds the covering lock(s), it can grant the requested lock to its principal, as a local lock.

Direct covering is the easiest covering to deal with. For direct covering, an LM needs to hold a lock on the same resource as is requested by its principal, in a mode that is at least as strong as the requested mode. Thus, a local LM needs to record, for each resource, the mode of the lock that it holds for the resource at the owning global LM. In addition, it must keep track of the local locks on a resource and the local principals that hold them.

For resource covered locks, the local LM need not have any lock directly on the requested resource itself. This is important as it permits such locks to be managed entirely locally, without any locks on the resource being visible at a global LM. The result, however, is that the local LM does not know whether locks being requested are resource covered without the requesting principal informing it.

Thus, one required extension to the functionality of LMs is that information be conveyed across the LM interface that permits a principal to know about covering locks. Then, the principal, using its knowledge of the MGH, can instruct the LM in its task of managing resource covered locks.

3.4 Acquiring Locks

3.4.1 Global Locks

The local LM, in its request to the global LM for a covering lock, asks for a lock mode that is, at a minimum, equal to the lock mode desired by its principal. To maximize private locking, the global LM grants the strongest lock mode that it can consistent with the current disposition of the lock that is at least as strong as the mode requested. For example, if the request is for an S lock, the local LM indicates to the global LM that this is the minimum acceptable lock mode. If the resource is currently not locked, the global LM grants an X lock to the local LM. If the resource is currently held in some mode compatible with S, but not X, then the global LM grants an S lock. If an S lock cannot be granted, then the request blocks.

When the global LM blocks the request, it then notifies the local LMs that are holding the lock in conflicting modes, asking them to release or demote the lock. The global LM supplies these LMs with the resource name and lock mode requested, so that they can respond appropriately (See section 3.5.)

3.4.2 Local Locks

When a principal requests a lock from its local LM, in addition to lock mode and resource identifier arguments, a requester indicates whether the lock request can be purely *local*, i.e. whether a lock on another resource (in the MGH) resource covers the requested lock. This tells the local LM that it need not hold globally a lock that directly covers the requested lock.

To treat resource covering and hence to enable purely local locking, whenever the local LM grants a lock, it returns to the principal the mode of the lock that the LM holds globally for that resource. This permits the principal to decide whether subsequent lock requests for finer grained resources need to have global covering locks or not. That is, the local principal knows whether the local LM holds a global covering lock on a resource higher in the MGH.

EXAMPLE: *A local principal is told, when it requests an IX lock on a file, that the local LM holds a global X lock on the file. Hence, when the principal requests X locks on records of the file, it tells the local LM that the requested locks can be purely local.*

3.5 Demoting Locks

Whenever a covering lock is demoted or released, all uncovered locks need to be posted to the owning LM prior to (or simultaneously with) this demotion. For example, if one demotes a file lock from S (a covering lock) to IS, then all S locks on pages of the file need to be posted to the owner of the page (and file) locks. This is the standard lock de-escalation already practiced by systems like Rdb/VMS [14].

Lock demotion can be very simple, e.g. when no lower level locks in the MGH are uncovered. The mode of the lock as held by the requesting principal is simply reduced. In this case the demotion is always possible. The need is to make sure that resource covering requirements are satisfied. And local locking introduces some complication. We describe two demotion situations.

3.5.1 Local Demotion

When a principal releases a local lock, the LM can choose to retain the lock itself. By retaining such locks, the LM can privately grant requests for the locked resources. This exploits high-water mark locking, where the LM retains a lock with the highest lock mode that any of its principals has recently requested. Subsequent requests for lesser lock modes on the same resource can be managed locally.

Lock retention of this sort may violate the multi-granularity locking protocol in so far as how local LMs hold locks at global LMs. A local LM may retain a lock on a resource lower in the MGH without necessarily having retained the intention lock needed to hold that lock. This is acceptable so long as local principals observe the multi-granularity locking protocol and local LMs demote their global locks on request when local principals are not holding locks that require these retained stronger locks. This is discussed in section 3.5.3.

When the local LM receives a conflicting request for a lock from a local principal, it informs the local principals currently holding the lock of this request, indicating that the demotion request is a local request. In this case, the local holders must post local locks for the resources they are accessing lower in the MGH that have been uncovered. The global locks held by the local LM need not change.

When a local lock is relinquished or demoted, it permits increased concurrency among the local principals. There is no impact on global concurrency because the local LM continues to hold the same global locks. Only changes in global locks, held by the local LM, can impact global concurrency.

3.5.2 Global Demotion

A local LM can unilaterally release global locks on resources without local locks. These locks cannot resource cover other locks. To release or demote a global lock on a resource that is currently locked by one of its principals, even if in a weaker mode, requires that we determine whether finer grained locks are uncovered by this.

A local LM has no need to demote global locks on resources being used by local principals unless these locks cause conflicts at the global LM. When the global LM receives a conflicting request for a global lock from some local LM, it notifies local LMs holding the lock. The global lock cannot be demoted without the local LM acquiring global locks on the uncovered locally locked lower level resources of the MGH. But local LMs do not have direct information about resource covering and the MGH.

A local LM hence may need to notify its principals holding locks on the resource for which demotion is requested, so that its principals can inform it as to resource covered locks that are uncovered by the demotion. Depending on the demotion requested, some local principals need notification while others do not. The notification requirements are indicated in Figure 2 for multi-granularity locking. To deal with an LM that supports arbitrary lock modes, it may be necessary to notify all holders of local locks on a resource whose global lock is to be demoted, so as to conservatively deal with the unknown change in covering.

Local principals, when they are notified concerning a demotion of a global covering lock must, before they accede to the demotion, ensure that uncovered locks are posted appropriately. These principal must then tell the local LM which additional locks need global posting. Once each of these locks is directly covered by a global lock, the principal notifies the local LM that the requested global lock demotion is acceptable.

If the principal itself holds a covering lock that lock demotion turns into an intention lock, it needs to acquire uncovered locks locally and cause the local LM to acquire covering global locks by indicating that these resources

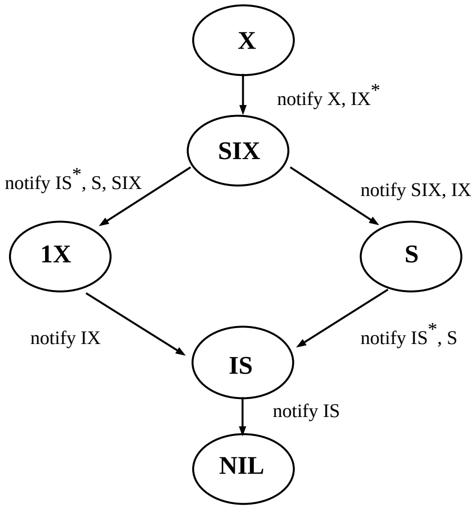


Figure 2: Holders of local locks on a resource need to be notified by a local LM when global lock demotion is requested. Both holders of locks that would no longer be directly covered and holders of intention locks, indicating resource covered locks are being held, need to be notified. The later are asterisk'd.

are not to be purely locally locked. If local locks are already held, then the principal instructs the local LM as to which locks are uncovered and directs the local LM to acquire global locks that directly cover them.

The impact of global lock demotion on local and global locks is illustrated in Figure 3.

3.5.3 Honoring the Multi-granularity Lock Protocol

A local LM (LM_A) may not always hold an intention lock on a resource on which it chooses to retain a lock that is stronger than any of the locks held by or needed by its local principals. This potentially violates the multi-granularity protocol. It is possible, for example, for LM_B to obtain a lock on a resource higher in the MGH that covers LM_A's retained lock. We need to understand why our locking protocol remains correct.

The essential observation is that no local principal of LM_A can exploit the retained lock without itself observing the multi-granularity locking protocol. This will cause the principal to request an intention lock higher in the MGH, and eventually on the resource that is locked by LM_B. That lock request

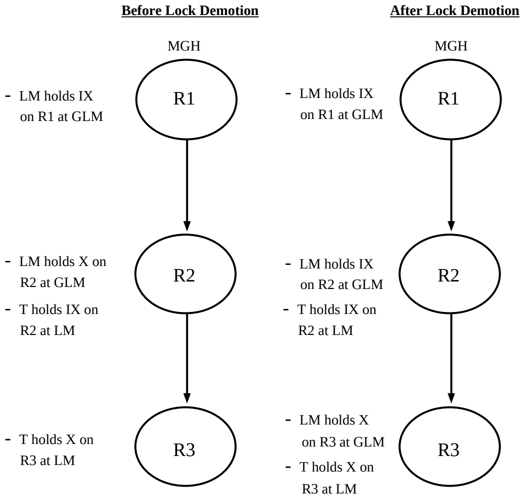


Figure 3: When a local LM is asked to demote its X lock on R_2 to IX, the LM notifies transaction T since the LM holds a covering lock and T has an intention lock. T tells the LM that its X lock on R_3 has been uncovered and needs a global covering lock.

will block.

The second potential danger is that a principal of LM_B will request the lock retained by LM_A, and will block, causing needless deadlocks. This is avoided by LM_A demoting (or releasing) its retained lock when it is notified of conflicting global requests. This it can always do as none of its local principals can be holding the lock in a way that precludes this demotion.

Finally, a principal of LM_B can hold a covering lock, which is permitted by LM_B's covering lock, and can exploit it by not locking resources lower in the MGH. Such a principal may access resources "locked" by the retained lock of LM_A. This is acceptable as no local principal of LM_A can acquire the retained lock without waiting for LM_B's covering lock to be demoted.

4 Generalizing Covering

We would like to extend the notions of covering and private locking to arbitrary lock modes. In this section, we show how to apply covering to logical locking and to data sharing cache management. In both these cases, we need to reconcile mutual exclusion with covering.

4.1 Logical Locks

4.1.1 What are Logical Locks?

We would like to extend the notion of lock covering to what have been called "logical" locks. For our purposes, a logical lock is one that applies to a "logical" resource that does not necessarily directly map to a physical unit, e.g. some particular disk block, disk area, etc. Logical resources, and their "logical" locks, are usually at a higher level of abstraction than physical resources and their locks.

A distinguishing feature of physical locks is that they are frequently NOT held for transaction duration. Rather, physical locks are acquired, some localized action is performed, logical locks are acquired, and then physical locks are released [4]. The notions of logical vs physical locks is captured more generally in the notion of multi-level transactions [9, 17], where subtransactions use low level locks during their execution, acquire high level locks before they commit(to their parent transaction) and then release the low level locks.

One would like for low level physical locks to be higher in the locking granularity hierarchy than logical locks. (Thus, "higher level" in the level of abstraction sense is opposite to the ordering in the MGH.) Then the physical locks can be made to cover the logical locks, permitting private logical locking. However, the early release of physical locks, while retaining the logical locks, precludes physical locks from covering logical locks. One transaction's new logical locks can conflict with another's retained logical locks, even though the first holds the physical locks with a covering lock mode.

EXAMPLE: *A lock(latch perhaps) is acquired on a page, a record lock is then acquired and the record updated. The page lock is released but the record lock still protects the record. Another transaction, when it acquires the page lock cannot assume that the page lock covers record locks for records on the page. It does not. There has been no intention lock left on the page by the earlier transaction. Hence, the new transaction must perform explicit record locking, even when it holds the page lock.*

4.1.2 Logical Lock Covering

A different lock protocol permits the covering of logical locks. Physical locks can be retained for as long as the outermost transaction whose subtransaction acquired them remains active. This is, in fact, the protocol required for nested transactions. Retained X and S locks on physical resources would thus become covering locks for the logical locks. The problem here is the obviously reduced concurrency.

The usual way to increase concurrency when covering impedes it too much is to replace covering locks by intention locks, and then to explicitly acquire locks at the next lower level in the MGH. However, X and S are frequently needed to provide exclusion on the physical resource itself, enabling the resource to be correctly read or updated, not simply to act as covering locks. Used in this way, a principal would acquire an X (or S) locks, update (or read) a page, and then demote the lock to IX (or IS) when exclusion was no longer required. These intention locks indicate that records of the page were being locked individually.

Unfortunately, the above protocol does not improve concurrency. An IX intention lock prevents the acquisition of the X (S) lock needed for exclusion. Indeed, IX must conflict with X (S) because X (S) is a covering lock. The difficulty here is that X and S lock modes are being used both for covering and for exclusion.

A solution is to introduce new lock modes that provide the exclusion needed on the physical resource without covering the logical locks. These new lock modes can then be compatible with the intention lock modes IX and IS. We name these new lock modes M (for modify) and R (for read). The lock compatibility matrix for our expanded set of lock modes is given in Table 2. Note that $M\ DCOVERS\ IX$, and hence is an intention lock for X, and similarly, R is an intention lock for S. M locks conflict with other M locks to provide the required exclusion. An R lock is similar to an IS lock but conflicts with M for exclusion.

Our protocol then becomes the following:

1. Request a covering (X or S) lock on a resource , e.g. page. If successful, keep locks lower in the MGH, e.g. logical locks, private, reducing lock overhead. Should there be a conflicting request after the subtransaction needing these locks has committed, demote these locks to IX or IS and simultaneous post all locks lower in the MGH that they were covering.

Lock Modes	IS	R	IX	M	S	SIX	X
IS	x	x	x	x	x	x	
R	x	x	x		x	x	
IX	x	x	x	x			
M	x		x				
S	x	x			x		
SIX	x	x					
X							

Table 2: Compatibility Matrix Including M and R.

2. If the covering lock request fails, request exclusion via a non-covering M or R lock on the resource. M and R lock modes require locks lower in the MGH to be posted publicly, exactly as with IX and IS intention locks. At subtransaction commit, demote M and R lock modes to IX and IS respectively, permitting other principals to acquire M and R locks and hence to access the resource.

4.2 Cache Management

There is an interaction between locking and cache management in data sharing systems. Locking strategies can either enable or disable certain forms of cache management. In this section, cache management strategies are described, with particular attention to their locking interaction. It should be clear that cache management fits into a multi-level transaction system, with the records of a page being the logical resources covered by the physical page locks. The cache management strategies differ as to the states of pages that are made available between stays at local systems, where a local system includes a local cache, a cache manager, and a local LM.

4.2.1 Transaction Consistent Pages(TCP)

With TCP, a page is transferred among systems only between transactions. The page made available is always the most recent version, and it reflects all and only updates of committed transactions.

TCP is realized by holding covering locks on the pages for transaction duration, X locks for updating and S locks for reading. Larger granule MGH locking, both covering and intention can also be exploited. Only these "physical" locks need be posted globally. Between local systems, the smallest resource granule is the page. Within a local system, locking with arbitrary granularity can be used, hence providing high concurrency, when the local system has its own LM. But the record/operation (i.e. logical) locking at local systems is private.

4.2.2 Updating of Current Pages(UCP)

UCP guarantees that whenever a local system updates a page, that the version of the page updated reflects all prior updates. Thus, updates are serialized, not concurrent, and the updating local system sees all the preceding updates. Pages transferred between systems are always current pages, but can contain uncommitted updates.

UCP is realized by holding a non-covering exclusion lock, either M or R, on the page while it is is being acquired or transformed and record locks are acquired. But this does not preclude other systems from holding record locks. Pages are only transferred or written to disk by the current holder of an M or R lock, hence ensuring that only the current page is transferred. M and R locks can be demoted to IX or IS when exclusion is no longer needed. Record locks need to be posted globally, as M and R are only intention locks. So, the penalty for the increased concurrency of UCP is additional global locking.

Systems that continue to hold IX or IS locks on a page must also continue to hold their record locks globally. However, even though they no longer may hold the current version of a page, they can continue to access records on the out-of-date page in their cache, confident that the records that they are protecting have not been altered elsewhere.

The UCP strategy does not preclude attempting to acquire covering locks on pages. As with logical locking, one can request a covering lock (X or S) and be satisfied with an exclusion lock (M or R) if a covering lock cannot be granted. This strategy avoids the need to choose on a system wide basis between low lock overhead and concurrency. Thus, it provides increased capability compared to the techniques described in [11, 12].

4.2.3 Concurrent Updating of Pages(CUP)

It may be desirable to permit multiple systems to update the same page without the UCP serializing of updates. Thus, the CUP strategy does not require that a current copy of the page be updated. Rather, this strategy can be accomplished by having a page manager (PM) merge the updates for all records on a page, and read and write the page to disk. An updater's requirement is to ship its record update to PM prior to releasing its X lock on it. A way to think about this is that, instead of reading and writing pages to disk, "reading" and "writing" records is directed to the appropriate PM, as if the granularity of the data were the record, not the page. This is like TCP applied to records, not pages.

CUP needs the locks held by concurrent updaters on a page to be compatible, while preventing the page from being locked with a conflicting covering lock. This is the role of intention locks (IX and IS). Hence, updaters acquire IX locks on the page, while X locking the records updated. Locks on records must be posted globally.

A PM may sometimes need to read the prior version of a page from disk to merge an update into it, but this is transparent to the updaters except for its performance impact. A smart PM will try and keep in its cache copies of pages for which there are outstanding locks that permit updates so as to avoid the need for the read from disk.

4.3 Local Locking

The existence of local LMs and local locking permits us to provide very high efficiency concurrency control. Distributed cache management and logical locking are examples where improvements can be made. Prior solutions either did not exploit local LMs and private locking or required that the smallest locking granule globally locked be the page. Our solution permits several alternative locking scenarios, depending on the pattern of requests. Pages can be locked:

1. globally by the local LM and locally by a principal with a covering lock, either X or S. No record locks need be posted. This is a low lock overhead option. In fact, a covering file lock can make individual page locks unnecessary.

2. globally by the local LM with a covering lock, and locally by a principal with an intention lock. No record locks need be posted globally. Several local principals can acquire record locks on the same page, hence improving local concurrency but increasing local lock overhead.

3. globally by the local LM with an intention lock, one of IX, IS, M, or R. Now record locks must be posted globally. Exclusion access to the page can be realized using M and R intention locks, demoted to IX or IS when exclusion is no longer required. This alternative permits serial access to data on a page by even remote principals without necessarily waiting to end of transaction. Lock overhead is highest because of the global locking but global concurrency is improved.

5 Summary

Lock managers that perform private locking can improve dramatically the trade-off between concurrency and lock overhead. A local LM can adjust its locking strategy between using covering and intention locks. For example, if too much global lock conflict occurs, covering global locks can be de-escalated to intention locks. This does require that the uncovered local locks be posted globally. If system lock overhead is too high, it can be reduced by making formerly global locks private via the acquisition of covering locks. Usually, global intention locks should be used for hot data, improving concurrency, while global covering locks are used for cold data, reducing lock overhead, with minimal impact on concurrency.

Rdb/VMS [6] supports this kind of boundary changing currently, but does not perform local locking. Instead, it uses the VMS cluster wide distributed lock manager(DLM), where all locks are global. Thus, all users experience the same level of concurrency. Resource covering is employed by principals to entirely avoid the posting of locks. When a conflicting request occurs, the DLM notifies principals holding the lock, who must then decide whether demotion is possible. In the Rdb/VMS case, the principals are processes executing the database system for a user.

Being able to manage locks locally is crucial for reducing lock overhead. The overhead of obtaining a global lock in a distributed system, e.g., via the DLM is at least an order of magnitude larger than when the locks are managed locally and privately. So local lock management can both reduce lock overhead and improve concurrency for principals of the local LM.

Physical locking, e.g. for cache management, or more generally, multi-level transaction locking, permits locks to be relinquished early. This is important for high concurrency. Concurrency is increased as principals can acquire and release locks on subtransaction boundaries. Our M and R lock modes facilitate this by allowing exclusion without covering.

References

[1] Bernstein, P., Hadzilacos, V. and Goodman, N. *Concurrency Control and Recovery in Database Systems* Addison Wesley, Reading MA (1987)

[2] Gray, J.N. Notes on data base operating systems. *Lecture Notes in Computer Science* 60, Springer-Verlag (1978), 393-481 also in IBM Research Report RC2188 (Feb. 1978), Almaden Research Center, San Jose, CA..

[3] Gray, J.N., Lorie, R. A., Putzulo, G. R., and Traiger, I. L. Granularity of locks and degrees of consistency in a shared data base. *IFIP Working Conf on Modeling of Data Base Management Systems* (1976) 1-29.

[4] Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzulo, F., Traiger, I. The recovery manager of the System R database manager. ACM Computing Surveys 13,2 (June 1981) 223-242.

[5] Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques* Morgan Kaufmann (1992) (final draft)

[6] Joshi, A. Adaptive locking strategies in a multi-node data sharing model environment. *Proc. Very Large Databases Conf.*(Sept. 1991) Barcelona, Spain, 181-191.

[7] Lampson, B. and Sturgis, H. Crash recovery in a distributed system. Xerox PARC Research Report (1976)

[8] Lomet, D. Recovery for shared disk systems using multiple redo logs. Digital Equipment Corp. Tech Report CRL 90/4 (Sept. 1990) Cambridge Research Lab, Cambridge, MA.

[9] Lomet, D. MLR: a recovery method for multi-level systems. *Proc. ACM SIGMOD Conf.*(June 1992) San Diego, CA, 185-194.

[10] Mohan, C. and Narang, I. Recovery and coherency-control protocols for fast intersystem page transfer and fine-granularity locking in a shared disks transaction environment. *Proc. Very Large Databases Conf.*(Sept. 1991) Barcelona, Spain, 193-207.

[11] Mohan, C. and Narang, I. Efficient locking and caching of data in the multisystem shared disks transaction environment. IBM Research Report RJ 8301 (Aug 1991) Almaden Research Center, San Jose, CA.

[12] Mohan, C., Narang, I., and Silen, S. Solutions to hot spot problems in a shared disks transaction environment. *Workshop on High Performance Transaction Systems* (Sept. 1991) Asilomar, CA.

[13] Rahm, E. Concurrency and coherency control in database sharing systems. U. Kaiserslautern Tech Report (Nov 1991) 6750 Kaiserslautern, Germany.

[14] Rengarajan, T., Spiro, P., and Wright, W. High availability mechanisms of VAX DBMS software. *Digital Technical Journal* 8, (Feb. 1989), 88-98.

[15] Snaman, W. et al The VAX/VMS distributed lock manager. *Digital Technical Journal* 5 (Sept. 1987) 29-44.

[16] Stonebraker, M. The case for shared nothing. *IEEE Database Engineering Bulletin* 9,1 (1986).

[17] Weikum, G. and Schek, H.-J. Architectural issues of transaction management in multi-layered systems. *Proc. Very Large Databases Conf.*(August, 1984) Singapore, 454-465.

Contents

1 Introduction **1**

1.1 Data Sharing and Server Independence 1

1.2 Some Locking Fundamentals 2

1.3 Lock Covering 3

1.4 Distributed Lock Management 4

2 Covering for Multi-granularity Locks **5**

2.1 Resource Covering 5

2.2 Intention Locks 6

2.3 Determining Lock Mode Conflicts 7

2.4 Protocols for Multi-Granularity DAGs 9

3 Local Lock Managers **10**

3.1 Private Locking 10

3.2 Owners and Holders 11

3.3 Local and Global Locks 12

3.4 Acquiring Locks 13

3.4.1 Global Locks 13

3.4.2 Local Locks 13

3.5 Demoting Locks 14

3.5.1 Local Demotion 14

3.5.2 Global Demotion 15

3.5.3 Honoring the Multi-granularity Lock Protocol 16

4 Generalizing Covering **17**

4.1 Logical Locks 18

4.1.1 What are Logical Locks? 18

4.1.2 Logical Lock Covering 19

4.2 Cache Management 20

4.2.1 Transaction Consistent Pages(TCP) 20

4.2.2 Updating of Current Pages(UCP) 21

4.2.3 Concurrent Updating of Pages(CUP) 22

4.3 Local Locking 22

5 Summary **23**