The Design and Performance Evaluation of a Lock Manager for a Memory-Resident Database System

Tobin J. Lehman
IBM Almaden Research Center
toby@almaden.ibm.com

Vibby Gottemukkala
Georgia Institute of Technology
vibby@cc.gatech.edu

January 14, 1994

Abstract

In the last fifteen years, lock managers for regular disk-based database systems have seen little change. This is not without reason, since traditional memory-resident lock managers have always been much faster than disk-based database storage managers and disk-based database systems had few alternative design options. However, the introduction of memory-resident database systems has created both new requirements and new opportunities for better lock managers. We present the design of a lock manager that exploits the special nature of the memory-resident storage component in the *Starburst* experimental database system. To achieve a performance advantage over traditional lock managers, our lock manager physically attaches concurrency control meta-data to the database data itself, thereby making the meta-data directly accessible, rather than indirectly accessible *via* a hash-table structure. Furthermore, our lock manager eliminates intention locks and changes the lock granularity of each relation dynamically to achieve high performance while ensuring a high level of sharing. Performance experiments comparing our lock manager with the default *Starburst* lock manager show that these basic design choices can provide significant performance gains.

1 Introduction

Powerful workstations employing large amounts of main memory provide fertile ground for developing a new class of database systems: general-purpose memory-resident database (MRDB) systems. As main-memory size increases, several design constraints disappear, thus making more options available. Previous work explored new designs for high-performance memory-resident database (MRDB) system subcomponents, such as the table storage manager [14], the index manager [15], and the log/recovery manager [16]. However, one vital subcomponent has remained relatively untouched: the lock manager.

Concurrency control costs are crucial to the overall transaction pathlength, and thus the implementation of the lock manager gets special attention in most production database systems.

However, although the lock manager is usually coded very carefully, with performance as the utmost concern, the basic *design* of database system lock managers has not changed since Jim Gray outlined the basic design of a lock manager in "Gray's Notes on Database Operating Systems," in 1978 [6]. In this paper we concentrate on changing the basic design of the lock manager, exploiting new design options that are made available by the base memory-resident database system.

In concurrency control (CC) mechanisms there is a direct relationship between the *degree of concurrency*, the number of simultaneously active transactions supported, and the *cost* of the CC mechanism. A CC mechanism that supports a high degree of concurrency typically costs more, both in terms of the number of raw instructions and the amount of complexity. On the other hand, a low cost CC mechanism typically limits the degree of concurrency. Thus, our objective was to design and implement a CC mechanism that could reduce the cost of concurrency control, when compared to traditional CC mechanisms, while maintaining a high degree of concurrency. Since studies have shown that lock-based concurrency control, compared to optimistic or timestamp methods, have the best overall performance in terms of throughput and resource consumption [3], we chose locking as our CC method.

There are two key aspects to MRDBs that can be exploited in implementing a CC mechanism. First, data resides permanently in memory and can be directly referenced by memory address. Thus, the MRDB data (tables and tuples) and meta-data (data about data, such as the size of the tuples, lock status of a tuple, *etc*) can be directly and uniformly accessed. Second, it has been shown that transactions accessing data in an MRDB are typically shorter in duration compared to those accessing the same data in a disk-based database (even when the data in the disk-based database was memory-resident at all times). [18] Thus, for most of the data, there is likely to be less contention in an MRDB than in a disk-based database. These two aspects of MRDBs allow us to change the basic design of the CC mechanism and achieve higher performance than that can be obtained from a traditional CC mechanism.

We implemented our design in the *Starburst* extensible Relational database system prototype. *Starburst*'s Data Management Extension Architecture (DMEA) [19] formalizes and simplifies the interface for creating new *storage methods* and *attachments*. Storage methods manage tables and their associated tuples, and attachments manage data structures related to tables, such as indexes. The DMEA interface allows data to be stored under the control of different storage methods based the needs and characteristics of the data (*e.g.* a temporary storage method for temp data, a B-tree storage method for sorted data, or a memory-resident storage method for high-performance data), while allowing uniform access to all the data through a single, SQL-based interface. Furthermore, *Starburst* provides a set of common transaction supports services such as logging, event management, memory management and concurrency control.

In this paper, we are concerned with a particular subset of the *Starburst* components. These are the traditional disk-oriented storage method (known as the Vanilla Relation Manager (VRM)), the memory-resident storage method (known as the Main-Memory Manager (MMM) [18]), and the regular *Starburst* lock manager (SB LM) that implements a traditional lock-based CC mechanism.

The MMM storage component is intended to store data for which fast response-time is crucial. For transaction support, the MMM storage component can either use the standard services provided by *Starburst* or it can use a specialized software component instead. Unfortunately, the standard *Starburst* lock manager (SB LM) is geared towards disk-based systems and, as we have pointed out, there are inherent differences between MRDB and disk-based systems which can be exploited for a better performing lock manager. For this reason, we chose to design and implement a specialized locking mechanism (MML) that is oriented towards MMM. An added benefit to implementing MML in *Starburst* is that it allows us to accurately compare the two CC mechanisms (MML and SB LM) side by side by simply varying the CC component with which MMM interacts.

The remainder of the paper is organized as follows: Section 2 describes an overview of the design of our CC mechanism, Section 3 outlines its implementation, and Section 4 presents its performance measurements. Section 5 relates previous work in the specific area of lock manager design to reduce lock calls and Section 6 describes some interesting alternative designs that, in the end, were not used. Finally, Section 7 gives our conclusions.

2 Design Overview

As stated earlier, our objective is to build a MMM lock manager (MMM LM) that has low cost and supports a high degree of concurrency. We achieve this objective in two ways. First, we exploit the uniform accessibility of data and meta-data in a MRDB to reduce lock lookup cost. Second, the MMM LM uses *dynamic, multi-granular* locking based on the demand for concurrency. When the demand for concurrency is low, the MMM LM imposes low overhead on the transactions and, when the demand for concurrency is high, it provides a high degree of concurrency, albeit at a greater cost.

In traditional disk-based database systems the *primary* copy of the data is on disk. However, to perform database operations the data has to be brought into memory and changes to the data have to be written back to the disk. Hence, the data needs to be in a *location independent* format. On the other hand, CC meta-data (lock information such as transactions holding a lock on a tuple, transactions waiting to acquire a lock, *etc*) is not in a location independent format *i.e.* the meta-data contains virtual memory pointers and hence cannot (easily) be stored on disk. Therefore, in a disk-based system the CC meta-data must be separated from the data being locked. In typical lock manager implementations (including *Starburst*), the CC meta-data is stored in a lookup table (usually a hash table) which adds to the locking pathlength because the meta-data can only be accessed through a table lookup operation. Additionally, since the lookup table is a shared, concurrently-accessible resource, extra concurrency control is needed to access it (see discussion of latches in Section 2.1).

On the other hand, in MMM, the data is always in-memory and therefore does not need to be in a location-independent format. Hence, there is no need to store data and its CC meta-data separately. In our design, the CC meta-data is physically attached to the data being locked. This association has the disadvantage that all lockable data must have statically allocated space for attaching the meta-data. Although this is wasteful of space (4 bytes for each lockable item) we feel that the performance gained justifies it.[1] By associating the meta-data with the data item, we eliminate the lookup cost for meta-data and provide uniform access to both the data item and its CC meta-data through the identifier of the data item. Besides these cost savings, this design also provides us with the opportunity to reduce overhead of other operations, such as latching, as discussed in Section 2.1.

2.1 Latches

There are two types, or levels, of concurrency control in a database system: latches and locks. A latch, or short-term lock, is a low level primitive that provides a cheap serialization mechanism with shared and exclusive lock modes, but no deadlock detection [6]. Latches are physical and are typically built into the resource(s) that they protect. In *Starburst*, latches are used to gain exclusive or shared access to shared resources, such as buffer pool pages or internal system control blocks. A lock, on the other hand, is a logical structure that is used to isolate one transaction from the effects of another. Using the strictest isolation level, *repeatable read*, locks can be used to make a set of transactions appear to operate in a *serialized* order.

A latch is typically associated with each logical unit of data that, as a whole, needs to be accessed in a consistent state. A straight-forward implementation of a database system would have latches guarding individual pages in a table, individual nodes in an index, and all or, perhaps parts, of the lock manager data structures.

In the *Starburst* MMM storage manager, we've taken a different approach. The memory-resident nature of the database data allows for a different style of table manager. [18] First of all, index entries in MMM indexes contain pointers to the actual tuples, rather than tuple identifiers, as in disk systems. Second, lock control blocks are also attached to tuples via memory pointers. The result is a table with dependent index and lock structures. (This is explained in more detail in Section 3.) So, rather than employ multiple latches to guard the many data structures that make up a table, its related indexes and its related lock information, we use a single latch. Although the savings of any single latch call is not particularly significant[2], by reducing the number of latch calls to 1 per table access, the relative savings is significant. Furthermore, the penalty for using a single latch to guard a table, its related indexes and its related lock information (*i.e. the lack of concurrency*) was not significant [5], since the lack of any I/O operations made the latch hold times relatively short.

[1]Furthermore, RDBMSs often charge a sizable space overhead for storing data. Although the exact amount is data dependent, *Starburst* often charges an additional 50%. Given this, it would seem that an additional 4 bytes per record is not much extra space at all.

[2]A latch operation takes about 20 instructions in *Starburst*.

2.2 Locks

In section 2.1 we mentioned that we reduced latch calls to one per table access and attached lock control blocks directly to tuples to reduce the physical overhead of the concurrency control mechanism. In this section we deal with the policy of the concurrency control algorithm, namely, the dynamic decision between low cost and high concurrency.

In a uniprocessor, the cost of locking could be eliminated by serializing transactions, effectively having transactions set a database-level lock [22]. Of course, to guarantee a reasonable response time in this environment, all transactions must be small. A less extreme and more general solution would be to use table locks, and thus reduce the granularity of sharing to tables. The cost of concurrency would be small, since only one lock would be needed per table. Unfortunately, since some tables would no doubt be popular (catalogs, for instance), the level of sharing for these tables would be unacceptably low.

No single locking granularity is sufficient. Table locks cannot be used universally – they would overly restrict sharing. Tuple locks cannot be used everywhere – they would be too expensive, especially for table-level operations.

Multi-granular locking was proposed in [7] as a mechanism that combines both coarse and fine-granularity locking. In this method, transactions are allowed to lock data items of different sizes and a hierarchy of data granularities is defined (*e.g.* table-level, page-level, and tuple-level could be one such hierarchy). A transaction could lock a data item of any granularity and all the data items under that data item in the hierarchy are implicitly locked. Thus, a transaction that accesses all or a majority of the tuples in a table, as in the case of a table-scan, could lock the table[3] and avoid the cost of locking each tuple individually. On the other hand, a transaction that needs to access only a few tuples in a table could set individual tuple-locks (fine granularity). However, for the mechanism to work correctly, the transactions that lock at a certain granularity, have to also set *intention locks* on the data items that are above the data item in the hierarchy. This adds to the cost of locking in two ways. First, each fine-granularity lock involves two or more locks, at least conceptually.[4] Second, the reason for using fine-grained locks is to ensure a high degree of concurrency, and thus high transaction throughput. However, when there is *low contention* for the data the degree of concurrency does not affect the transaction throughput rate. But, under the multi-granular locking scheme, a transaction that accesses a few fine-granularity data items *always* performs multiple fine-grained lock operations, irrespective of the contention for the data. Such a transaction, in the case of low contention, could benefit by setting a coarse-granularity in place of the multiple fine-grained locks.

These observations imply that, to reduce the locking pathlength of transactions and maintain

[3]Thus, all the tuples in the table are implicitly locked.

[4]In database systems where lock calls are expensive, a transaction may cache the information about the table-lock so that it doesn't need to set the table-level intention lock repeatedly. Unfortunately, this caching moves some of the locking logic out of the lock manager and into the database storage manager which adds to system complexity.

a high degree of concurrency, locking should be coarse-grained when contention for data is low and fine-grained when contention for data is high. We define contention for a data item at any given time as a function of both the number of transactions waiting for a lock on the data and the duration of time each of those transactions spent waiting to acquire the lock. Data contention is both *space* and *time* variant, *i.e.* data contention varies between sets of data and the contention for any given unit of data varies over time.[5] Thus, it is inefficient to specify the same lock granularity for all the data in a database, or to specify a static lock granularity for each data set (say, a table).

From the arguments above it would seem that a lock manager that determines lock granularity dynamically, rather than statically, would better satisfy our design objectives. Such a lock manager would allow a transaction to lock a data item logically, while the mechanism *dynamically* decides the appropriate granularity of the physical lock. For example, a transaction could request a lock on a tuple but, based on the current amount of contention for the table, the lock manager has the choice of setting a table-lock on the table containing the tuple or locking the tuple itself. We call this mechanism *Dynamic Multi-Granularity Locking* (DMGL). The DMGL mechanism should be able to compute the degree of contention for any set of data to decide upon the granularity of locking for that set. Furthermore, the DMGL mechanism should be able to switch the lock granularity of any data set given its degree of contention. When the degree of contention is high the locking granularity is *de-escalated* to a finer granularity, and when the degree of contention is low it is *escalated* to a coarser granularity.

In our design, the DMGL mechanism computes contention for individual tables. The design incorporates two lock granularities: coarse (table-level) and fine (tuple-level). The default lock granularity for a table is coarse. However, when the contention for the data in a table increases beyond a certain threshold, its lock granularity is de-escalated to tuple-level. The DMGL escalates the lock granularity of a table back to coarse only when the contention for the data in the table decreases. Thus, at any point in time all the locks on a table are either all table-locks or all tuple-locks, based on the table's lock granularity at that time. This effect of the DMGL scheme eliminates the need for intention locks. Since in a traditional hierarchical locking scheme, fine-granularity and coarse-granularity locks can exist simultaneously, transactions have to be aware of this possibility while setting locks. On the other hand, in the DMGL scheme, since the locking mechanism is aware of the lock granularities, transactions are freed from the burden of hierarchical locking.

3 Implementation of the MMM Lock Manager

In this section we describe the components of the MMM Lock Manager (MML) and identify some of the design choices we made in its implementation. Furthermore, we also describe some problems peculiar to our design and how they are handled.

Figure 1 shows the organization of MMM data and its CC meta-data. The database storage is composed of variable-sized segments where each segment represents a table in the database.

[5]There is some data, however, such as catalog data, for which there is always a high degree of contention.

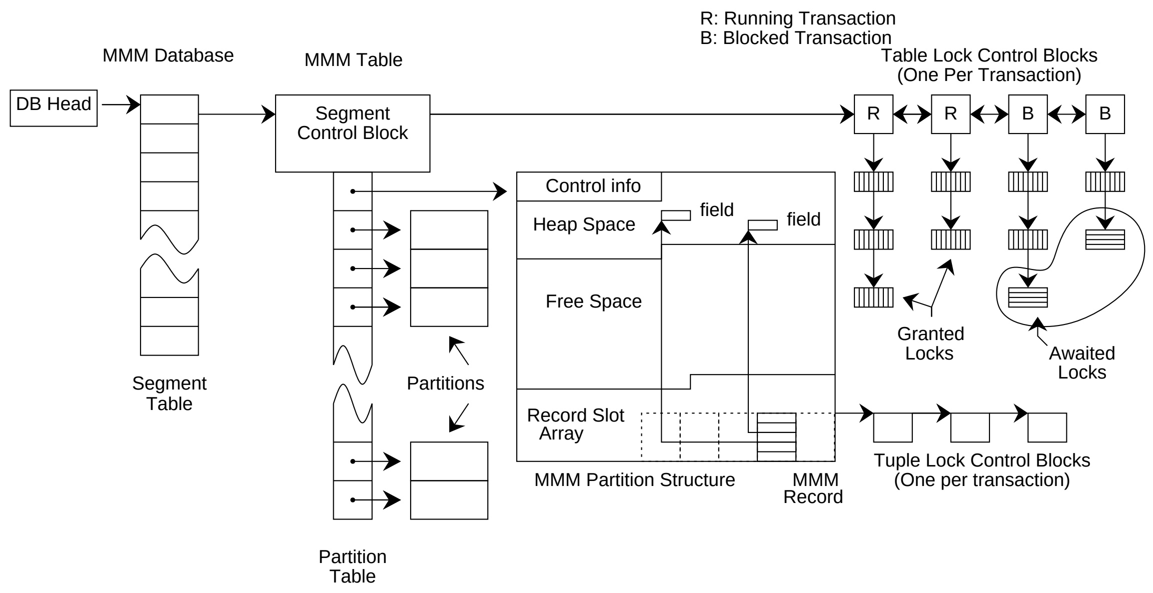

Figure 1: Organization of MMM Data and Meta-data.

Segments are composed of fixed-size partitions which store the tuples of the database. The segment control block contains most of the corresponding table's meta-data such as number of partitions, number of tuples, tuple structure, *etc.* In addition, the segment control block is the anchor point for table-level lock control blocks.

Each partition in a segment (a table) contains some control information, an array of record slots (at the bottom), and a heap for record data (near the top). The record slot array and the data heap grow toward each other. Each record slot contains a set of record field data pointers that point to the record's field values in the heap. In addition, one of the record slot pointers is used as the anchor for tuple-level lock control blocks. The organization of the CC meta-data shown in the figure will become clearer from the discussion following.

Transactions generate requests for locks on tuples or tables. The action that the MML takes when it receives a lock request is dictated by a per-table granularity flag located in the segment control block. When a transaction requests a tuple-lock, the MML checks the granularity flag for that tuple's table. If the lock granularity is "coarse", the lock request is *converted* into a table-lock request. If the lock granularity is "fine", the lock request is handled as a tuple-lock request. On the other hand, when a transaction requests a table-lock, the MML has no choice but to acquire a table-lock.[6] We call such a lock an *explicit* table-lock.

From the perspective of the DMGL scheme, an explicit table-lock is different from a table-lock set in place of a tuple-lock request. This difference in the table-locks will become clearer when we explain our de-escalation and escalation strategies. Thus, the DMGL scheme impacts

[6]This limitation is due to the fact that we have only two granularities (tuple and table) for the DMGL scheme. Although we could define more granularities we did not see any practical application for it.

fine-granularity locking (tuple-locks) the most, because the scheme has a choice in terms of the actual lock that it can grant. The scheme also reduces the overhead of fine-granularity locking by eliminating the need for setting coarse-grained intention locks for each fine-granularity lock request.

In the DMGL scheme, coarse-grained (table-level) locking represents the low overhead/low concurrency state, whereas fine-grained (tuple-level) locking represents the high overhead/high concurrency state. Since our objective is to reduce the overhead of CC, coarse-grained locking is the default. When the contention for a table is high, the lock granularity for that table is de-escalated to tuple-level and when the contention becomes low it is escalated back to table-level. In the next section, 3.1, we discuss how and when escalation and de-escalation of lock granularity for the tables in the database is done.

3.1 Escalation

When a table's lock granularity is “fine”, the MML attempts to escalate the granularity back to “coarse” at the earliest opportunity. The reasoning is that since coarse-grained locking has less overhead we would like to grant coarse-grained locks whenever possible. However, we have to first ensure that escalation of lock granularity does not cause immediate data contention which in turn would cause de-escalation. This condition of *no immediate contention upon escalation* is likely to be met if all the transactions accessing a table hold compatible locks. An efficient method to determine if this condition for escalation is met is to test the compatibility of the aggregate modes of the active transaction for that table.

Before we can specify when escalation can take place, we need to take a closer look at the cause for de-escalation, namely, contention. Contention for data takes place when transactions request the same lock in incompatible modes. The contention is removed only when the transactions that hold the incompatible locks terminate, which indicates that the time to explore the possibility of escalation is when a transaction terminates (commits or aborts). For example, assume that transactions T1 and T2 have shared aggregate modes with respect to a table A, and T3 has exclusive aggregate mode with respect to table A where table A's locking is fine granularity. Then the best time to escalate the lock granularity of table A is when T3 terminates (T1 and T2 have compatible aggregate modes) or when both T1 and T2 terminate.

The first step towards escalation is to identify a table for which locking granularity can be escalated. When a transaction terminates, all the tables accessed by that transaction are examined as part of the unlocking process. The lock granularity of a table can be escalated if:

- Its lock granularity is fine, and
- Aggregate lock modes of the transactions active with respect to that table are compatible.

Once a table is identified, the process of escalating its lock granularity is a simple matter of switching its lock granularity flag to indicate that the locking granularity is coarse. The data structures representing table-locks for the active transactions are already in place (Table Lock

Control Blocks in Figure 1) and they keep track of the aggregate mode of the respective transactions' tuple-locks. The aggregate mode also represents the mode in which the new table-locks are held. The tuple-locks that the transactions might have set while the lock granularity was tuple-level are left in place in case de-escalation takes place again. These locks are released only upon transaction termination.

Explicit Table-Locks

Transactions that need to access all or most of the tuples in a table request an *explicit* table-lock. In the context of DMGL, an explicit table-lock is restrictive because de-escalation cannot take place in its presence. The reason for this is that de-escalation requires that an explicit table-lock be converted to tuple-locks on *all* the tuples in the table. This defeats the purpose of de-escalation since locking all the tuples in the table and locking the table itself are identical in terms of the degree of concurrency allowed. In this respect, an explicit table-lock is different from the table-lock set on the behalf of a tuple-lock request. When a transaction sets an explicit table-lock on a table whose lock granularity is set to fine, the transaction has to wait until the lock granularity is escalated. Thus, the granularity flag provides the functionality of *intention* locks in hierarchical locking by preventing a table-lock from being set when tuple-level locking is in effect.

For fairness reasons, a transaction waiting for an explicit lock should not be kept waiting indefinitely while other transactions continue to set tuple-locks that prevent escalation from taking place. Therefore, when a blocked explicit table-lock exists in the lock queue of a table, no *additional* transactions are allowed to set tuple-locks in the table. Thus the the wait time of the explicit table-lock requestor is bounded by the termination time of all the transactions that already hold tuple-locks in that table.

3.2 De-escalation

The method chosen for de-escalation is important because it affects both transactions that already hold locks on a table as well as transactions that are blocked on lock requests for that table. It would seem that the most reasonable de-escalation method is one that does not impede the progress of running transactions, yet makes it possible for blocked transactions to run after the table-locks have been de-escalated. De-escalation could be done using any of the following methods:

1. *Roll-back* all the transactions that hold locks on the table after indicating the change in lock granularity. This is wasteful of the transaction processing already done.

2. *Wait* for the transactions currently holding table-locks to commit and then indicate the switch in the granularity of locking. There are two problems with this method. First, once the active transactions commit there may no longer be any contention at which point de-escalation is wasteful. Second, contention for the table's data could be increasing while waiting for the active transactions to commit.

3. *Remember* the tuples lock requests of the active transactions. When de-escalation is needed, convert the remembered tuple-locks into *real* tuple-locks and switch the granularity of locking. This method has the advantage that it does not affect the active transactions while reducing the average wait time for blocked transactions.

We feel that method 3 satisfies the criteria for the de-escalation method best, provided that de-escalating table-locks into tuple-locks itself is not expensive. We discuss the costs involved in this process in Section 3.3. The key requirement for this method of de-escalation is to remember the tuple-locks requested by transactions. We call these the *remembered locks.* Remembered locks are associated with the table-lock (Figure 1 shows remembered locks attached to table lock control blocks) of each transaction and are processed at de-escalation time.

3.3 De-escalation Daemon

In Section 3.1 we pointed out that the escalation process is invoked upon the occurrence of an event that is part of normal transaction processing, namely transaction termination. On the other hand the invocation of de-escalation cannot be associated with any real event in database processing, but is based on the detection of high contention for a table.

Detecting High Contention

The contention for a table can be quantitatively measured by employing a *de-escalation counter.* When a table-lock request of a transaction cannot be satisfied, the lock manager records the increase in contention by incrementing that table's de-escalation counter, before blocking the transaction. However, this alone is not sufficient because it fails to incorporate the element of time (the duration of the wait) into the measure of contention.

These factors imply the need for an independent process to handle the de-escalation requirements of the database, namely, the *De-escalation Daemon.* The daemon periodically visits each active table in the database and increments the de-escalation counter for that table by the number of transactions waiting for a table-lock. The periodicity of the daemon's visits includes the factor of wait time into the de-escalation counter. Since the de-escalation counter is a measure of data contention it should not increase monotonically. When a transaction waiting for a table-lock is awakened, the table's de-escalation counter is decremented by that transaction's contribution to the counter. A transaction's contribution is the number of times the daemon had visited that table while the transaction was blocked.

Performing De-escalation

The other duty of the daemon is to perform the actual lock de-escalation. The daemon examines the de-escalation counter of each table and in the case that it is greater than a pre-defined threshold (implying high degree of contention) decides to de-escalate the lock granularity for that table. However, in the presence of an explicit table-lock, the daemon postpones its decision to de-escalate. As mentioned earlier, the de-escalation daemon converts remembered locks into real tuple-locks and (possibly) wakes up some of the transactions that were blocked. After the conversion of locks, the daemon resets the table's de-escalation counter. During de-escalation the daemon holds the table-latch in exclusive mode. This prevents other transactions from setting locks during the transition.

Lock Conversion

The process of de-escalation itself does not limit concurrency because of the inexpensive nature of the conversion of remembered locks into real tuple-locks. Before we can argue about the cost of conversion we have to distinguish between *awaited* and *granted* locks. An awaited lock represents a lock request that caused the transaction to block due to a conflict. Hence, running transactions have only granted locks whereas blocked transactions have one awaited lock each and zero or more granted locks (see Figure 1). The lock modes of all the granted locks are *compatible* with each other but are *incompatible* with those of the awaited locks. The process of conversion of remembered locks to *real locks* is handled as follows:

1. Convert all the remembered locks of running transactions into real locks.

2. Convert all the granted locks of blocked transactions into real locks.

3. Convert all awaited locks of blocked transactions into real locks.

The distinction between awaited and granted locks is important because the conversion of the two types of locks to real locks is handled differently. Since all the granted locks have compatible lock modes, steps 1 and 2 only involve some pointer manipulation to add lock control blocks (LCBs) to the tuple lock chains; no compatibility checks are needed. Also, no new tuple LCBs need to be allocated because they have already allocated as remembered locks (both remembered and real tuple locks share the same LCB data structure). On the other hand, the conversion of awaited locks (step 3) requires a check of the compatibility of the requested lock mode with other locks that might already exist on the tuple. This is similar to a normal tuple lock request. If the lock request can be satisfied the corresponding transaction is unblocked, thus satisfying the purpose of de-escalation. Thus, the distinction between awaited and granted locks allows us to reduce the cost of de-escalation. The only costly operation in the process is the conversion of awaited locks. However, the number of awaited locks will be low—equal to the number of blocked transactions.

3.4 Implications of the Design

In this section we discuss some issues that are artifacts of our design decisions for MML, specifically the DMGL mechanism. Furthermore, we also discuss how these issues are handled.

Thrashing

A potential problem with the process of de-escalation and escalation based on contention is that *thrashing* might occur, *i.e.* the process could keep repeating while very little transaction processing is accomplished. Our design prevents this from happening by imposing two conditions on the DMGL mechanism. First, escalation is done only when it is not going to cause any immediate contention. Second, de-escalation does not take place at the first occurrence of a lock conflict at the coarse-granularity, but happens only when contention builds up and crosses a threshold. Our definition of contention ensures this by accounting for both the number of transactions waiting for a lock and the duration of their wait. Thus, there is a form of hysteresis in the escalation–de-escalation process.

Deadlocks

Deadlock detection and resolution are integral to lock-based CC mechanisms. In our environment, where table-locks may be set when tuple-locks are requested, false deadlocks can occur. For example, if transactions T1 and T2 both hold *shared* locks on table A to access tuples r1 and r2 respectively, and then each transaction tries to lock its respective tuple in exclusive mode, they will deadlock even though both transactions actually locked different tuples.

To prevent transactions from being terminated because of false deadlocks, the MML first checks to see if any of the deadlock victims are blocked on a table whose lock granularity is coarse. In the case of coarse granularity, the list of remembered locks are checked to see if there really is a conflict. The absence of conflict indicates a false deadlock, and the lock manager de-escalates that table's lock granularity to resolve the deadlock.

Forced Escalation

When a transaction locks more than a threshold[7] number of tuples in a table it is effectively locking the entire table. In such a case it is more efficient to grant an explicit table-lock after which the individual tuple-locks do not have to be remembered. The MML keeps track of the number of tuple-locks on a per transaction per table basis. When a transaction crosses the threshold for a table, the lock manager tries to set an explicit table-lock for the transaction. Forced escalation can also be used when there is no more space to allocate lock control blocks (LCBs) for the tuple-locks or the remembered locks.

[7]The threshold is a pre-defined ratio between the number of tuples locked by the transaction and the number of tuples in the table.

4 Performance

In this section we compare the performance of the MMM Lock Manager (MMM LM) with that of the *Starburst* Lock Manager (SB LM). The SB LM is a "traditional" lock manager that stores lock data in a hash-table and supports general-purpose hierarchical locking in the form of intention locks.

The hardware used for our experiments was a model 530 IBM Risc System 6000 workstation with a 25 MHz processor and a 64 Kbyte I&D cache. The peak integer performance of a model 530 Risc System 6000 is nominally 25 MIPS, but in practice we have estimated that 10 MIPS is closer to the mark for database workloads because of poor instruction and data locality, which causes processor cache misses. Our machine was configured with 128 megabytes of memory, which was more than enough to keep all data cached in memory during test runs. All our tests were run under AIX Version 3.1.

A real-time hardware clock on the Risc System 6000 with a resolution of 256 nanoseconds was used to measure query execution time during test runs. We instrumented *Starburst* with assembly-routine macros that made use of the Risc System 6000's hardware clock facility to time key code segments in *Starburst*. Thus, we were able not only to obtain the overall execution time of a query, but we were also able to obtain an accurate breakdown of where the time was being spent in the query's execution.

4.1 The Results

As we were interested in the most common case where there was no lock contention, we measured lock cost by setting non-conflicting read locks. The execution times in Table 1 are presented in four groups. The first group shows the execution times of the lock, unlock, and combined (lock/unlock) operations performed by the regular *Starburst* lock manager (SB LM). The second group shows the execution times of the lock, unlock and combined operations performed by the MMM lock manager (MMM LM) for setting "remembered" locks. A remembered lock is set instead of a real tuple lock when the lock granularity level of a table is at the table level. The third group shows execution times of the same operations performed by the MMM lock manager for setting real tuple locks, as would be the case when the table granularity flag is at the tuple level. The last group shows the cost of de-escalating a table lock (essentially, a remembered lock) into a real tuple lock.

The execution times presented in Table 1 represent the *minimum* execution times for both SB LM and MMM LM. Since the SB LM is a "traditional" lock manager that stores lock data in a chained-bucket hash table, its performance is sensitive to the number of locks set. *Starburst* memory constraints prevented us from creating a lock hash table larger than 1,000 slots, so lock numbers greater than 500 caused SB LM's performance to decrease.[8] The best performance was

[8]By using a dynamic hashing algorithm, such as linear hashing [20] or modified linear hashing [14], we could eliminate this problem for any number of locks.

Operation	Cost (microseconds)
SB LM Lock	25 μs
SB LM Unlock	12 μs
SB LM Combined	37 μs
MMM LM (seg + Remembered) Lock	6 μs
MMM LM (seg + Remembered) Unlock	5 μs
MMM LM (seg + Remembered) Combined	11 μs
MMM LM (tuple) Lock	9 μs
MMM LM (tuple) Unlock	15 μs
MMM LM (tuple) Combined	24 μs
MMM LM De-escalate Op	2.3 μs

Table 1: Execution times of a Read Lock/Unlock call for *Starburst* and MMM

obtained from SB LM when setting and releasing 100 locks, 25 microseconds and 12 microseconds, respectively.

As expected, the MMM lock manager tuple lock cost is less than that for the *Starburst* lock manager, as the direct access to the Lock Control Block (LCB) avoids the cost of the hash-table lookup step. Additionally, because of the direct access, MMM LM is not affected by the number of locks set on a table. Overall tuple locking cost for the MMM lock manager is derived from a combination of the two lock granularity modes: table and tuple. When there is no contention for the table, all of the locks set are table-level, and the tuple locks are remembered. In this mode, the unlock cost reflects processing the table-level LCB (checking for waiting transactions, *etc.*) and then throwing away (recycling) the remembered tuple-lock data structures. Thus the cost of unlock for remembered tuple locks is relatively small (5 μs), as no checking is needed. When there is contention for the table, some of the locks set are real tuple-level locks, although, recall that only one lock is set per tuple, as table-level intention locks are not needed. The tuple-unlock cost of the MMM lock manager and the *Starburst* lock manager are similar, as they perform similar functions. Both lock managers traverse the LCB chains, set a latch on each LCB, and check for waiting transactions. In fact, the MMM LM is slightly slower when unlocking a tuple because of the extra checking done to test for possible lock granularity escalation.

Not shown in Table 1 is the time used by the MMM lock manager to set a fixed table lock, which is 10 microseconds for lock and 10 microseconds for unlock (20 combined). The first lock call in a table, for either a remembered or real tuple lock, also incurs the cost of setting the initial table lock. Hence the *first* tuple lock (and subsequent unlock) operation costs approximately 31 (20 + 11)

Trans Type	Transaction time (msec)	MMM Component time (msec)	MMM percentage	Transaction description
T1	864 ms	450 ms	52%	Table Scan, count 100%
T2	794 ms	383 ms	48%	Table Scan, return 10%
T3	1,006 ms	586 ms	58%	Index Scan, count 100%
T4	445 ms	107 ms	24%	Index Scan, return 10%
T5	7 ms	3 ms	4%	Fetch, Insert or delete

Table 2: Results of benchmarks run on *Starburst*.

and 44 (20 + 24) microseconds, for remembered and tuple locks respectively.[9] Thus, using regular locking and intention locks, the *Starburst* lock manager would require 74 microseconds to lock and unlock the first tuple, whereas the MMM lock manager would require either 31 or 44 microseconds. Subsequent tuple-lock calls to the *Starburst* lock manager would still cost 74 microseconds, whereas subsequent calls to MMM LM would cost 11 or 24 microseconds, for remembered or real tuple locks, respectively. If we were to move the intention lock checking logic into the *Starburst* storage components that use the regular *Starburst* lock manager, then we would be able to reduce the lock cost somewhat, although the amount is difficult to quantify. Notice also that a table-scan operation does not have the problem of repeatedly resetting the table-level intention lock, as the logic of the table-scan code is such that the intention lock is set exactly once. However, repeated probes with an index, or any modifying operation, such as insert, delete, or update, do repeatedly set table-level intention locks in *Starburst*.

Finally, we computed the cost of de-escalating remembered locks into tuple locks. At first glance, the cost of de-escalation might appear to be the cost of setting a regular tuple-lock for each remembered lock, spending 24 + 11 = 35 microseconds total per tuple lock. However, this is not the case because the MML distinguishes remembered locks into granted and awaited locks (recall this discussion from Section 3.2). In fact, we found that the cost of remembering a lock and then de-escalating it later appears to be slightly less than the cost of setting the tuple lock in the first place. We've seen this effect before in our benchmark experiments. When an operation is repeatedly done in "batch" mode, it runs faster — most likely due to a better hit ratio in the processor instruction and data caches.

4.2 The Bottom Line

Consider the transaction execution numbers presented in Table 2. The MMM Component times represent the amount of time spent in the MMM storage component. If we compute the number of locks set by these transactions and then multiply that by the lock cost, we'll be able to compare the

[9]The first lock call is actually a bit less than this, as both operations are done in a single lock call — however it's easier to explain it this way.

Trans Type	Transaction time	MMM time	Number of Locks set	SB LM cost	MMM LM hi cost	MMM LM lo cost
T1	864 ms	450 ms	1	0.074 ms (.02%)	.024 ms (.01%)	.011 ms (0%)
T2	794 ms	383 ms	1	0.074 ms (.02%)	.024 ms (.01%)	.011 ms (0%)
T3	910 ms	496 ms	10,000	370 ms (42%)	240 ms (33%)	107 ms (17%)
T4	433 ms	85 ms	1,000	37 ms (30%)	24 ms (22%)	11 ms (11%)
T5	7 ms	0.3 ms	1	0.074 ms (20%)	0.044 ms (12%)	0.031 ms (9%)

Table 3: Comparing lock costs with MMM storage component costs.

total time spent locking with the total time spent in the MMM storage component. The interesting cases are ones in which locking cost plays a major role, such as index scan or update transactions. To keep the numbers consistent, the MMM storage component and transaction times in Table 3 do not include any lock manager time. The lock times measured during the runs to generate Table 2 were subtracted from the MMM and transaction times.

Table 3 shows the bottom line. For the three lock costs, namely, SB LM lock/unlock, MMM LM fine-grained (tuple) lock/unlock, and MMM LM coarse-grained (table + remembered) lock/unlock, we display the cost of the lock call in milliseconds, and also display the lock overhead as a percentage of the total MMM time used for locking ($\frac{lock\ time}{MMM\ time\ +\ lock\ time}$). The benefits vary depending on transaction type, the specific lock mode needed by the MMM LM (remembered or real tuple lock), and the amount of unnecessary intention-mode locking performed by SB LM. However, except for the case of table scans where locking cost is not significant, the MMM lock manager can reduce SB LM locking cost by 30% to 60%, as a result of using more efficient data structures and setting cheaper coarse-grained locks when fine-grained sharing is not needed. These savings equate to a 15% to 30% reduction in MMM time and a 5% to 20% reduction in overall transaction time. Additional savings can be produced by MMM LM in those cases where SB LM is called to set redundant table-level intention locks.

5 Related Work

In this section we identify related research and discuss its applicability to our work. Compared to the large body of work in the locking family of concurrency control methods (for example, [1, 2, 3, 4], there is little or no published work in the area of changing the locking mechanism itself. The System R lock manager described in [Gray 78] appears to be the basic design choice of most database systems.

On the other hand, there are a few methods published for reducing the *number* of lock calls, and thus reducing overall locking cost [10, 11, 14, 9, 13, 21]

5.1 Cheap First Locks

The idea behind "Cheap-First-Locks", used by IMS FASTPATH [10, 11] is that the first transaction to lock a data item does not pay the full cost of a lock. Instead, it simply leaves behind a flag, warning all other transactions that the item is really locked. If a second transaction tries to lock that item, it notices the flag, and then acquires locks for *both* transactions. If the modes are incompatible, the second transaction then blocks on the lock, waiting for the first transaction to finish.

In a memory-resident database environment, where transactions are short (no I/O operations), it is quite likely that most tuples will be locked by no more than one transaction at any given time. If locking could be made almost free for this case, it could be a major cost savings. However, a problem arises when the cheap-first-locks have to be freed when a transaction terminates. We thought of two solutions:

1. Record in the lock manager the names (or locations) of the data items that were flagged. During transaction termination, those flags could be reset or removed.

2. Rather than a flag, mark the data item with a (monotonically increasing) transaction id. Then, using *lazy evaluation*, a transaction determines if a Cheap-First-Lock still exists by looking at the transaction id and seeing if the corresponding transaction has terminated.

Although method 2 appears to be a workable solution, we did not implement it for two reasons. First, *Starburst* does not have monotonically increasing transaction id's. Second, in our case, the cost of setting a regular lock is not significantly more than the cost of checking if a transaction has committed. This was not the case for IMS FASTPATH, where the shortcut for a Cheap-First-Lock was significantly cheaper than a regular lock call, as the lock call required a context switch.

5.2 Amoeba

Amoeba [23], a data sharing project at the IBM Almaden Research Center, created a lock manager protocol that reduced the number of global lock calls generated. *Amoeba* is a data sharing system comprising many nodes and a collection of disks. The term *node* denotes a single machine boundary that may contain multiple CPU's. All nodes of the system connect to all of the disks, or data, hence the name *data sharing.* Each node in *Amoeba* contains a local lock manager, and at least one node also contains a global lock manager. As calls to the global lock manager are much more expensive than calls to a local lock manager, the *Amoeba* project concentrated on minimizing calls to the global lock manager.

In *Amoeba*, the database locks a *page set*,[10]and then locks some number of pages in the page set. The database requests these locks from the local lock manager, which then, in turn, passes *some* of the requests on to the global lock manager. Specifically, it passes on all page set locks, and

[10]A *page set* is a number of pages, usually covering 2 to 20 megabytes, that corresponds to a physical file.

then, in addition, it passes on those page locks that correspond to pages that belong to a page set that is currently being shared by more than one node. When a page set is not being shared, all of the page lock requests stay in the local lock manager and are not sent to the global lock manager, thus saving the message costs of those global page lock calls.

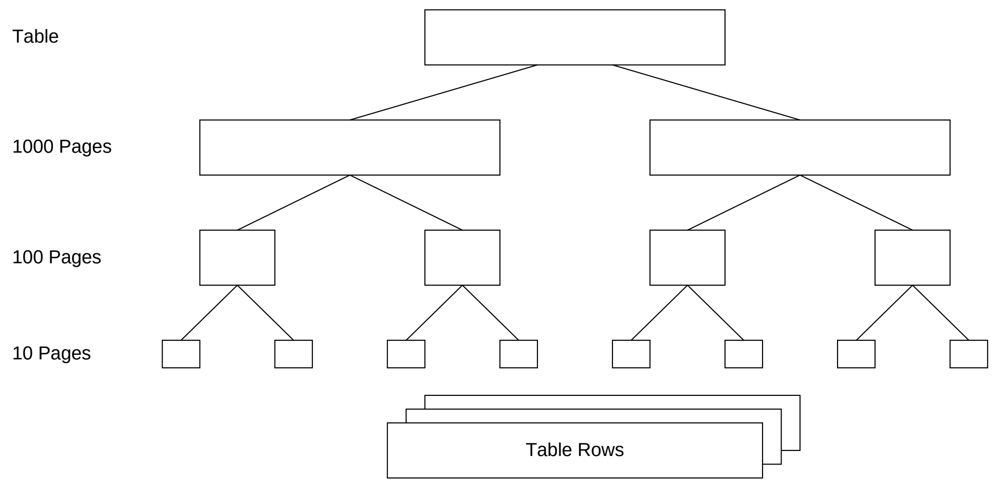

Figure 2: Rdb's Adjustable Lock Granularity Tree.

5.3 Adjustable Lock Granularity

Digital Equipment Corporation's Rdb/VMS database product, which runs on VAX Clusters, uses a somewhat different scheme to reduce the number of lock calls [9, 13]. In Rdb, there is only one global lock manager, the VMS Distributed Lock Manager (DLM). Rather than use a fixed set of lock granule sizes, such as table, page, and tuple, an Rdb application can set arbitrary Lock granule sizes. Figure 2 shows an example instance of the Adjustable Lock Granularity (ALG) Tree.

In Rdb, a lock requestor has the option of specifying an Asynchronous System Trap (AST) routine that will be invoked whenever there is a conflicting request for the lock. AST's are used to replace a flexible lock with a fixed intention-mode lock of the same granularity level plus a set finer granularity flexible locks. Thus, each time there is lock contention at a lock level that is internal to the ALG tree, the DLM calls the AST to replace the flexible lock with an intention-mode lock and some number of finer-granule locks.

In our implementation we have chosen to wait for the contention to *build up* before deciding to de-escalate rather than de-escalate at the first occurrence of a conflict as in the case of Rdb. This makes sense, especially in the MRDB environment with short transactions, because the conflict could be short-lived. De-escalation when contention is high amortizes its cost. It has to be noted that our implementation also provides the flexibility of allowing de-escalation to take place under any degree of contention by varying the de-escalation threshold. Furthermore, when several Rdb nodes access the same table at the tuple level, the cost of de-escalating a node's locks from table-level all the way down the ALG tree to tuple-level may become high.

5.4 The Original MRDB Work

In his PhD thesis, Lehman presented a design for a lock manager that uses lock de-escalation to reduce the number of lock calls [14, 17]. The lock manager described in this paper is based on that original design.

6 Interesting Alternatives

Besides work that was directly related to the design of our concurrency control mechanism, there were other various works that caught our attention because they might help us to improve the design. Predicate Locks an example of something that, at least initially, appeared to be tailor made to our needs.

6.1 Predicate Locks

A particular implementation of predicate locks, known as *Precision Locks* [12] also seemed appealing, both for the novelty and for the opportunity to lock at the column level. Furthermore, in certain cases, it appeared that a transaction could set a single lock over an entire range of values, thereby saving a considerable number of instructions that would have been needed to lock each individual tuple in the range.

Briefly, precision locking works as follows:

- A reader creates a predicate that represents the range that it is scanning. Thus, a table scan would lock the entire table, but an index scan would lock only the range of the key predicate.
- A writer creates a read predicate that represents the value that it is writing (and reading), as well as a list of values that it is writing.
- All readers, before posting their read predicate, first test their read predicate against the list of all outstanding write values. If there is no conflict, the reader then posts the read predicate and then is free to continue without further concurrency control activity.
- All writers test their new write value against all outstanding read predicates, including those read predicates posted by writers. If there is no conflict, the writer posts a read predicate representing the newly written value, and also posts a new value to the write list.

In the cases where there are mostly readers and few writers, this scheme seems very appealing. A reader simply posts a simple read predicate rather than set multiple locks. However, there are also a few drawbacks to this scheme. First, regular locking is still needed for non-value oriented locks, such as those used for page-space reservation, or data structures such as indexes. Second, it is not clear how one can efficiently test the right set of read predicates. It appears that a separate list of read predicates per index is needed. Although a set of read predicates can be searched with an interval search tree (or skip list) [8], the bookkeeping for all of the predicates begins to look unwieldy. Finally, updaters need to post both the old and new values as both write values and read predicates. The cost savings of this scheme seems less when examined closely. With great disappointment, we decided to drop the idea of incorporating predicate locks into our design.

7 Conclusions

In this paper we described the design and implementation of a lock manager that is suited for a memory resident database. Our approach to this problem was to reexamine the traditional design, identify specific areas for improvement, and redesign them. This approach led us to Dynamic Multi-Granularity Locking, simplified addressing of lock data, and table-latches. All of the above have a significant impact on both performance and the complexity of the locking mechanism.

Through our performance results, we have shown that the pathlength of fine-granularity locking can be greatly reduced when there is low contention. In the presence of high contention, too, we show pathlength reductions, although not to the same extent. Our implementation of the MMM lock manager improved the performance of the MMM storage component by upto 30%, and the overall transaction response-time by upto 20%. Furthermore, we have shown that the DMGL mechanism can be implemented efficiently. The major tradeoff in our scheme is the extra space required for the physical attachment of lock data to the data being locked. However, the gains in performance adequately justify this "wasted" storage.

Various heuristics and thresholds such as the de-escalation threshold and forced escalation threshold have been used in our implementation. Further experimentation is needed to evaluate heuristics for computing data contention and to determine practical thresholds, to better achieve the goals of our design.

8 Acknowledgments

Our thanks to Stefan Dessloch for his suggestions on improving this paper.

References

[1] R. Agrawal, M. Carey, and M. Livny. Models for Studying Concurrency Control Performance: Alternativves and Implications. In *Proceedings of the ACM SIGMOD Conference*, May 1985.

[2] P. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2), June 1981.

[3] M. Carey and M. Stonebraker. The Performance of Concurrency Control Algorithms for Database Management Systems. In *Proceedings of the Int. Conference on Very Large Data Bases*, August 1984.

[4] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11), November 1976.

[5] V. Gottemukkala and T. Lehman. Locking and Latching in a Memory-Resident Database System. In *Proceedings of the Int. Conference on Very Large Data Bases*, August 1992.

[6] J. Gray. Notes on Database Operating Systems. In *Operating Systems, An Advanced Course*. Springer-Verlag, New York, 1978.

[7] J. N. Gray, R. A. Lorie, and G. R. Putzolu. Granularity of Locks and Degrees of Consistency in a Shared Database. In *Proceedings of the Int. Conference on Very Large Data Bases*, 1975.

[8] E. Hansen. The interval skip skip list: A data structure for finding all intervals that overlap a point". Technical Report WSU-CS-91-01, Wright State University, 1991.

[9] L. Hobbs and K. England. *Rdb/VMS A Comprehensive Guide*. Digital Press, 1991.

[10] IBM. *IMS Version 1, Release 1.5 Fast Path Feature Description and Design Guide*. IBM World Trade Systems Centers (G320-5775), 1979.

[11] IBM. *Guide to IMS?VS V1 R3 Data Entry Database (DEDB) Facility*. IBM International Systems Centers (GG24-1633-0), 1984.

[12] J. Jordan, J. Bannerjee, and R. Batman. Precision Locks. In *Proceedings of the ACM SIGMOD Conference*, May 1981.

[13] A. Joshi. Adaptive Locking Strategies in a Multi-node Data Sharing Environment. In *Proceedings of the Int. Conference on Very Large Data Bases*, August 1991.

[14] T. Lehman. *Design and Performance Evaluation of a Main Memory Relational Database System*. PhD thesis, University of Wisconsin-Madison, August 1986.

[15] T. Lehman and M. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the Int. Conference on Very Large Data Bases*, August 1986.

[16] T. Lehman and M. Carey. A Recovery Algorithm for a High-Performance Memory-Resident Database System. In *Proceedings of the ACM SIGMOD Conference*, May 1987.

[17] T. Lehman and M. Carey. A Concurrency Control Algorithm for a Memory-Resident Database System. In *Proc. of the Foundations of Data Organization and Algorithms (FODO)*, 1989.

[18] T. Lehman, E. Shekita, and L. F. Cabrera. An Evaluation of the Starburst Memory-Resident Storage Component. *IEEE Trans. on Knowledge and Data Engineering*, 4(6), December 1992.

[19] B. Lindsay, J. McPherson, and H. Pirahesh. A Data Management Extension Architecture. In *Proceedings of the ACM SIGMOD Conference*, June 1987.

[20] W. Litwin. Linear Hashing: A New Tool for File and Table Addressing. In *Proceedings of the Int. Conference on Very Large Data Bases*, October 1980.

[21] C. Mohan and I. Narang. Recovery and Coherency-Control Protocols for Fast Inter-system Page Transfer and Fine-Granularity Locking in Shared Disks Transaction Environment. In *Proceedings of the Int. Conference on Very Large Data Bases*, August 1991.

[22] K. Salem and H. Garcia-Molina. Crash Recovery Mechanisms for Main Storage Database Systems. Technical Report CS-TR-0340-86, Computer Science Dept., Princeton University, April 1986.

[23] K. Shoens. Data Sharing vs Paritioning for Capacity and Availability. In *Int. Workshop on High Performance Transaction Systems*, September 1985.