Versioning Algorithms for Improving Transaction Predictability in Real-time Main-memory Databases

Rajeev Rastogi[1]
S. Seshadri[1]
Philip Bohannon[1]
Dennis Leinbaugh[1]
Avi Silberschatz[1]
S. Sudarshan[2]

[1]Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974-0636

[2]Indian Institute of Technology
Computer Science and Engineering Dept.
Bombay, India

Abstract

We present a design for multi-version concurrency control and recovery in a main memory database, and describe *logical* and *physical versioning* schemes that allow read-only transactions to execute without obtaining data item locks or system latches. Our schemes enable a system to provide the *guarantee* that updaters will never interfere with read-only transactions, and read-only transactions will not be delayed (for the purpose of ensuring data consistency) as long as the operating system provides them with sufficient cycles. Consequently, transaction executions become more *predictable* – this partially alleviates a major problem in *real-time database system* (RTDBS) scheduling, namely, significant unpredictability in transaction execution times. As a result, in addition to a transaction's deadline, a more accurate estimate of its execution time can also be taken into account, thus facilitating better scheduling decisions. Our contributions include several space saving techniques for the main-memory implementation, including improved methods for *logical aging* of data items and the introduction of *physical aging* for low-level structures. Some of these schemes have been implemented on a widely-used software platform within Lucent, and the full scheme is implemented in the Dalí main-memory storage manager.

1 Introduction

Transactions, in a *real-time database system* (RTDBS), have completion deadlines associated with them. The objective of the transaction processing component in such a system is to maximize the number of transactions that complete before their deadlines while preserving database consistency [SZ88, Ram93].

RTDBSs are extensively deployed in a number of environments like telecommunications, program stock trading, and command and control systems. Of these, telecommunications constitutes an important application domain. In a telecommunication network, RTDBSs typically are either embedded in

network elements (e.g., switches, routers) and store call routing and forwarding tables, or are employed in adjuncts to switches to perform functions like 800 number translation. The RTDBSs are frequently consulted during call setup to allocate resources (e.g., trunks, switch ports) when a circuit is established, and for mapping a dialed number to a destination number. As a consequence, since call setup is required to complete in a few milliseconds, the read-only transactions that access the RTDBSs have stringent response time requirements which are in the order of *tens of microseconds*.

In order to meet the strict deadlines associated with the above read-only transactions, the RTDBS infrastructure must provide support for fast and predictable execution times for such transactions. As pointed out in [Ram93], the two major obstacles to achieving this in conventional DBMSs are (1) disk access latency incurred when accessing a disk-resident page, and (2) blocking due to lock conflicts – this occurs when a transaction requests a lock held by a different transaction in a conflicting mode. In this paper, we employ main-memory database (MMDB) technology and version-based concurrency control mechanisms to overcome the above-mentioned challenges.

Unlike disk latency, which is variable and in the order of tens of milliseconds, main-memory accesses are fast and measured in a few hundreds of nanoseconds. Thus, storing the entire database in main memory can result in short and bounded transaction execution times. While disk-based databases exhibit improved performance if the entire database can fit in the main-memory buffer cache, a MMDB (e.g. [SGM90, LSC92, JLR+94, DKO+84]) improves performance further by dispensing with the buffer manager, and tuning algorithms to the flat storage hierarchy and the reduced cost of indirection. Also, MMDB schemes attempt to minimize *space* usage, of vital importance since main memory remains about one hundred times as expensive as disk space. Since disk I/O in an MMDB is only needed for persistence of the log, no disk activity is required on behalf of read-only transactions. As a result, response times for read-only transactions are more predictable, making MMDBs highly suitable for a large class of real-time applications in which the most time critical queries are read-only. However, as mentioned earlier, a read-only transaction may still have to wait on locks held by an update transaction, which may in turn be waiting on a different transaction, or on disk writes to the log. These waits become a serious source of unpredictability for response times.

Multiversion concurrency control methods [MPL92, Had88, AS89, BG83, BC92a, IKK90, CFL+82] prevent update transactions from conflicting with read-only transactions by providing the latter with a consistent but somewhat *out-of-date* view of the database. In order to provide this view, multiple versions of recently updated data items are retained. Early multi-version schemes [Ree78] used timestamps for readers and writers, but more recent *multi-version locking* schemes [CFL+82, AS89, BC92a, MPL92] use timestamps with read-only transactions, allowing them to use old versions without locking, while requiring updaters to perform locking. However, none of the above techniques guarantees complete isolation of read-only transactions from update transactions in a system, since the access path to the data could be modified by update transactions. Thus, read-only transactions must obtain latches (semaphores) to ensure that they read physically consistent data.

Requiring read-only transactions to obtain latches could cause update transactions to interfere with their execution. Furthermore, in a number of environments, application code is often linked directly with database code, accessing the database directly through shared memory for speed. This introduces the possibility that a process could fail while holding latches or locks, leading to long delays in any transaction waiting on one of these latches or locks while the death of the first process is detected and handled. By avoiding latches, read-only transactions will never encounter this delay. Finally, in a main-memory database system, the use of latches imposes a substantial overhead [GL92] and, by avoiding their use, significant performance gains can be obtained for read-only transactions.

In this paper, we present schemes that eliminate the need for both locking *and latching* by read-

only transactions without sacrificing *recency*, since read-only transactions see all committed updates as of their start-times. Locks are eliminated by a novel implementation of *logical versioning* for main memory, an area which has been well-studied for disk-databases. Our implementation reduces the storage space overhead required to keep track of versions. Latches are eliminated by a mechanism we call *physical versioning* [KL80], that is applied to the access paths to data items. Updates to these access paths are not made in place – instead, the updates are made on a new copy of the node, called a "physical version". The new version of the node is linked into the access path using an atomic word-write (an operation which is universally supported on standard architectures). This enables read-only transactions to traverse data structures without acquiring latches. By freeing them from getting any latches, the performance of read-only transactions is completely de-coupled from that of update transactions, and becomes a simple function of available CPU resources, making it relatively easy to guarantee the response times of these transactions.

There are numerous benefits to making an entire class of read-only transactions in a RTDBS environment completely non-blocking. Since transaction executions become more predictable, their running times can be estimated more accurately. Consequently, in addition to a transaction's deadline, a more accurate estimate of its execution time can also be taken into account when making scheduling decisions. For instance, for a certain aborted transaction before it is re-executed, it may be possible to conclude that it will not meet its deadline even if it had exclusive access to all the resources in the system – as a result, the transaction can be discarded early, thus resulting in better utilization of resources [Ram93]. Similarly, transactions with smaller execution times may be given a higher priority than those with longer running times – this could result in a larger number of transactions meeting their deadlines.

Since memory conservation is far more critical in a main-memory database, we have developed two orthogonal techniques for garbage collecting unneeded versions, without affecting the recency of data seen by read-only transactions. First, we present a novel technique for *aging* old versions of data items, in which effort can be traded for memory utilization, allowing the database management system to adapt to currently available memory. Second, we introduce our schemes for *physical aging* which allow memory used for low-level data structures to be more quickly returned to system control.

We also describe techniques for performing transaction rollback and recovery from system crashes. Our recovery schemes do not require the generation of physical undo log records, thereby reducing disk I/O and overhead due to logging. Variants of the schemes presented in this paper are implemented in Lucent's 2NCP product's database subsystem which performs a number of tasks, including 800 number translation, in the long distance telecommunications network. The main memory and versioning architecture enable the 2NCP to meet the stringent time constraints of real-time transactions that are executed during call setup. We have also implemented the complete schemes in the Dalí main-memory storage manager at Bell Laboratories [JLR+94].

The remainder of the paper is organized as follows. In sections 2 and 3, we provide an overview of logical and physical versioning, respectively. We present the design of our logical version manager in Section 4. In Section 5, we discuss related work, and in Section 6, we give our conclusions and directions for future work.

2 Logical Versioning

The idea of maintaining multiple versions of an item was first proposed by [Ree78] and is known as multi-versioning or just versioning. In this paper, we refer to this as logical versioning to differentiate it from physical versioning, which is described further in Section 3. In this section, we describe the

basic structure of our multi-version locking scheme [CFL+82, AS89, BC92a, MPL92]. We also propose a design for version management that takes advantage of the fact that data is resident in memory.

2.1 Overview

In a system that supports multi-versioning, transactions are classified as *read-only transactions* – those that only read items, and *update transactions* – those that update or write some item, or simply want access to the most current data. When an update transaction updates a data item, a new version of that item is created. Update transactions follow the two-phase locking protocol by locking items they read or write. When an update transaction T, commits, it is assigned a timestamp denoted by $\mathsf{tsn}(T)$ which is obtained by incrementing a global *logical timestamp counter*. As part of commit processing, before any locks held by the transaction are released, the transaction stamps each version it has created with $\mathsf{tsn}(T)$. Thus, the versions of an item can be ordered according to their timestamps. A read-only transaction is assigned a timestamp by reading (but not incrementing) the logical timestamp counter when it starts. Subsequently, for each item, the read-only transaction reads the latest version whose timestamp is less than or equal to its timestamp.

At startup, read-only transactions read the logical timestamp counter without any locking implying that 1) the timestamp counter must be incremented by a transaction only after the stamping process is complete and the transaction has committed and 2) the counter itself either fits in a word or is read through a pointer ensuring that the non-locking read is atomic with respect to the update. Furthermore, in order to prevent multiple updaters from interfering with each other, every updater must obtain an X latch (ignored by read-only transactions) on the logical timestamp counter before accessing it during commit processing. The latch is held until all the versions have been stamped and the counter has been incremented.

A version that is no longer needed by any read-only transaction can be deleted and the space reclaimed. This action is called *aging* that version. A version can be aged safely if no read-only transaction exists which has a timestamp equal to or larger than that of the version in question, but smaller than the next newer version of the item. Algorithms for aging are presented in Section 4.9.

2.2 Logical Versioning in Main Memory

We now discuss the design of a version manager for a main-memory database system. The basic difference between a main-memory based design and a disk-based design is that the versions of an item need not be physically clustered together for efficient access. This fact, as we will see shortly, allows us to minimize the space overhead of versioning in a main-memory system.

In most disk-based schemes [BC92b, MPL92], storage space for a certain number of versions is pre-allocated on each page for efficient access which could result in under-utilization of storage space (e.g., each item on a page has a single version). In our design, on the other hand, space for versions is dynamically allocated as they are created. Furthermore, since a database could consist of millions of "cold" items that have only one version, we eliminate the space overhead due to versioning which is imposed on these items by not using *any* space inside a version for bookkeeping information. Instead, we use an auxiliary data structure called a Version List Entry (VLE), shown in Figure 1, to maintain the bookkeeping information and link the versions of an item together. An item that has only one version is stored as is without a VLE. VLEs are dynamically allocated as subsequent versions of the item are created, and for items with more than one version, a VLE exists to represent each version.

A VLE contains the timestamp of the transaction that created it, and a pointer to the version itself. The VLEs of an item are linked together as a doubly linked list ordered by timestamp. Read-

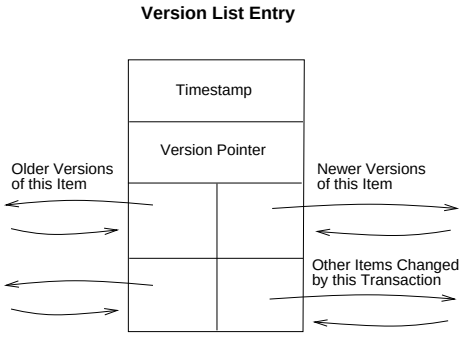

Figure 1: Structure of a Version List Entry

only transactions traverse the VLE chain of an item in order to find the required version. Each VLE is also on a list of versions created by the same transaction while the transaction is active; this facilitates easy update of timestamps of versions created by the transaction when it commits. Also, when a transaction aborts, the versions created by the transaction can be efficiently determined and deleted.

3 Physical Versioning

In this section, we describe physical versioning and discuss interactions between physical and logical versioning.

3.1 Overview

Physical versioning is a technique that permits read-only transactions to access data structures without getting any latches or locks, even while other update transactions are updating the data structure. Physical versioning is based on atomic reads and writes of words, operations which are universally supported on current generation architectures.

The main motivation for physical versioning comes from real-time systems in which read-only transactions are time critical, and cannot afford unpredictable delays that may occur if they have to wait for latches. Whereas logical versioning eliminates the need for read-only transaction to wait for locks, physical versioning eliminates the need for read-only transactions to wait even on latches. With physical versioning, the response time of a read-only transaction depends only on the number of CPU cycles available to it, which helps make it's finish time highly predictable.

To demonstrate the physical versioning idea, we first consider an access structure such as a singly linked list or a tree, which has a single root. Assume that the sole pointer to the root is stored in a single word and that updaters are serialized. Updaters then copy the entire data structure and make changes on the copy. On completion, they update the pointer to the root; since the write is atomic, readers either see the old version of the data structure or the completed new version of the data structure. Thus the entire update to the data structure appears to be atomic to read-only transactions, even though they have not obtained any latches or locks. Locking protocols are of course required to prevent updaters from interfering with each other.

Once a data structure has been copied and the root pointer to the data structure has been overwritten by the new pointer, new read-only transactions cannot access the old data. However, there may be existing read-only transactions that are accessing the old version of the data structure. Therefore, the

old version cannot be deleted immediately after the root pointer has been updated. Instead, it must be deleted only after all read-only transactions that were accessing it are no longer accessing it.

The process of detecting that an old version of data is no longer being accessed, and actually deleting it, is called *physical aging*. Physical aging is complementary to logical aging, and a different set of techniques are used to implement it. Physical aging is best implemented by a separate process which we refer to as the *physical ager*. We discuss physical aging in detail in Section 3.4.

3.2 Purely Physical Versioning

In this section, we show how data structures in the form of a tree lend themselves to *efficient* physical versioning while allowing readers to see an operation-consistent state of the tree. In other words, the operations are each performed atomically with respect to readers. This is a fairly important class of data structure since B-trees, T-trees[1] and even hash tables with collision chains[2] fall in this category. We assume nodes in the tree are fixed size entities and for every edge out of the node, a pointer to the other node in the edge is stored within the node itself. The techniques in this section work irrespective of whether the system is using logical versioning or not, hence the name purely physical versioning.

We define a *component* to be any connected set of nodes of the tree. Given an update operation, the component *affected* by the operation is the set of nodes changed by the operation, plus any other nodes which may be necessary to connect the changed nodes. The *root of the component* is defined in the obvious way as the root of the smallest subtree that contains the component.

Let N be the root of the component affected by an operation. Then, physical versioning is performed as follows:

1. First copy the component; let N' be the copy of N. The data in each node in the copy is exactly the same as the data in the corresponding nodes in the original tree, except that pointers to nodes in the component now point to the new copies of the nodes.

2. Perform the update on the new copy of the component. This can create new nodes, and update or delete existing nodes in the new copy of the component. However, no node in the original tree (including the old copy of the component) is affected by the update.

3. Atomically update the pointer to N to point to N' instead (if N is the root, the pointer to N is the root pointer for the tree, otherwise it is from the parent of N).

The final atomic update of the original pointer to N to point to N' exposes the update to read-only transactions, and it is easy to see that read-only transactions do not see partial updates. The affected component for many well known operations on B-trees, T-trees and hash tables can be easily defined. For example, consider an insert into a B-tree, which attempts to insert a new entry into a leaf node L. The insert can cause a split, which can propagate several levels up in the tree, say up to node I. Then the affected component consists of the path from I to L. Copying this set of nodes and then performing the update starting from the new copy of I will not affect the original copy of the tree. Furthermore, by toggling the pointer to I in I's parent, to the new copy of I, the entire split operation can be made to execute atomically with respect to read-only transactions.

[1] T-trees are an index structure based on AVL trees [AHU74]. They were proposed in [LC86] as a storage efficient data structure for main-memory databases.

[2] The main hash table can be treated as a single large tree node.

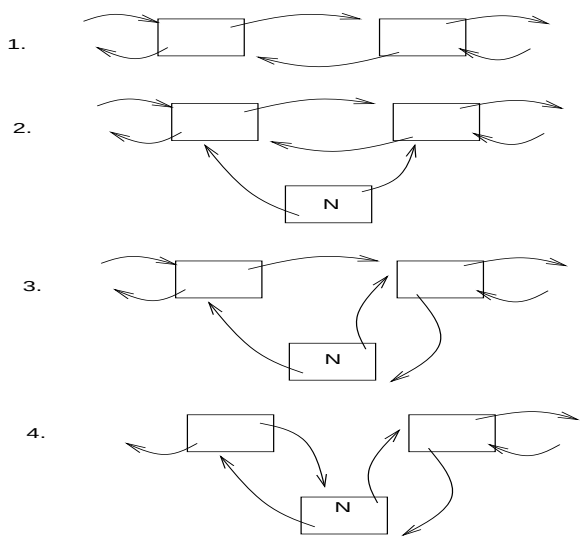

Figure 2: Temporary Inconsistency in a Doubly Linked List

3.3 Combining Physical and Logical Versioning

So far we have considered the data structure to be a tree. If the data structure is not a tree, purely physical versioning cannot easily be applied. Common examples of such structures include doubly linked lists, (for example, the VLE chain of Section 2), and B-trees whose leaf nodes are linked together. Also, the affected component of an operation may be large, with many nodes that are not updated but have to be included to make the component connected. For example, a delete operation on a T-tree may have to update both a node and it's in-order predecessor node in the tree, in which case the affected component contains all nodes in the path between the nodes.

To handle the above problems, we relax the requirement that read-only transactions see an operation consistent view of data. We only require that the data structure be consistent enough for traversal – however, a read-only transaction may be exposed to a partially executed operation. This does not create problems *in a system which supports logical versioning* since a read-only transaction is not interested in any update that takes place after it begins. A read-only transaction, T, essentially executes against a transaction consistent snapshot that existed as of some time before it began execution. We therefore exploit logical versioning to weed out partial effects of updates executing concurrently with read-only transactions. We will demonstrate this by considering an insert into a doubly linked list. Let us assume that logical versioning provides some mechanism for detecting whether a given node in the doubly linked list is to be read by a read-only transaction or not. N cannot be atomically linked into the list since there are two pointers (N's successor and predecessor) that have to be atomically updated. However, N can be linked in by first making the successor of N's predecessor N and then the predecessor of N's successor N, as show in Figure 2. Notice that the linked list can be traversed consistently at all points of time except that the node N may be visible while traversing in one direction but not in the other. Notice that regardless of the traversal, physical consistency is ensured, and a read-only transaction sees every node that belongs to the consistent snapshot for it. It does not matter whether the read-only transaction sees N or not since in any case the logical versioning mechanism would not let the read-only transaction read N.

3.4 Physical Aging

The process of reclaiming space occupied by older copies of data that have been physically versioned is called physical aging. The old versions of the data have to be preserved as long as a read-only transaction can attempt to read the data (this is similar to logical aging). The crucial difference between logical and physical aging is, however, in when a read-only transaction ceases to see some data. We assume that each operation traverses an access structure afresh and pointers to nodes are not cached across operations. Therefore, a piece of data that is visible to a read-only transaction can not be logically aged for the *duration of the reading transaction* while a piece of data that is visible to a read-only transaction during an operation cannot be physically aged for the *duration of the reading operation*.

For example, the old version of an index node that is physically versioned after a read-only transaction initiated an index lookup operation may be visible to the transaction until the lookup completes and thus, cannot be aged until then. However, the old version can be aged once the lookup completes since a subsequent index lookup operation performed by the same read-only transaction retraverses the index (beginning with the most recent root node) and thus, does not see the old version of the index node. Thus, the physical ager has to ensure before aging a node that no operation of a read-only transaction began before the node was physically versioned.

We associate a *physical timestamp* with each read-only transaction (this is different from the timestamp assigned to the transaction by logical versioning). The physical timestamp is ∞ if the transaction is not currently performing any operation. It is set to the value of a global *physical timestamp counter* before starting an operation and reset to ∞ afterwards. An updater, after making an update that physically versions a piece of data and makes it unreachable for future read-only transactions increments the global physical timestamp counter while holding a latch. The updater also adds the older version into a physical ager's list by appending to the list, an entry containing a pointer to the version being aged and the value of physical timestamp counter when the version was aged (that is, after the older version was unlinked and the physical timestamp counter was incremented). The physical ager then can de-allocate the space for an older version once no transaction has a physical timestamp smaller than the version's physical timestamp.

The above scheme works reasonably well if operations are short. However, in the presence of long index operations like an index scan, very old versions can remain in the system for a long time. The reason for this is that treating an entire scan as a single operation could potentially result in long delays in the aging of data. This problem can be alleviated by decomposing a scan into a number of smaller operations as follows. The key idea is to force the scan to start at the top of the tree to find the next node (retraverse the tree) occasionally if it is holding up the reclamation of old versions for a long time. More specifically, with every scan, two bits are maintained – a retraverse bit and an in_progress bit. The retraverse bit is used by the physical ager to force the scan to perform a retraversal of the tree. The Next operation for a scan first sets the in_progress bit to 1 and checks to see if the retraverse bit for it has been set to 1. If so, it discards it's current state (cached from the previous Next call), sets the retraverse bit to 0, obtains a new physical timestamp and retraverses the tree from the current root node. Before returning, Next sets the in_progress bit to 0. In order to reclaim data held up by a long running scan, the physical ager notes the current physical timestamp then sets the retraverse bit for the scan to 1. It then waits until the retraverse bit or the in_progress bit for the scan becomes 0, following which it frees data assuming the physical timestamp for the scan is the maximum of noted timestamp and the scan's current physical timestamp.

Note that, unlike logical versioning, physical versioning can take place on a per-access-structure

basis, decreasing contention on the global timestamp and speeding recovery of space.

4 Logical Version Manager

In this section, we describe in detail the design of a main-memory based logical version manager. Recall from Section 2 that the versions of an item are doubly linked using VLEs, an auxiliary data structure (see Figure 1). We will assume that inserts and deletes into these doubly linked lists are performed using physical versioning as described in Section 3.3. This enables traversals and updates to the VLE chain to be performed without obtaining any latches. Furthermore, versions of items that are no longer needed (and the VLE entries that point to them) are placed on the physical ager's list and aged in the manner described in Section 3.4.

We assume for this presentation that each item has a primary key that uniquely identifies it. We will also assume that all access to items is through index (primary or secondary) structures. In the case that an item has a single version, an index entry for that item is a direct pointer to the version. If the item has multiple versions, the index entry points directly to one of the VLEs. If more than one of an item's versions have the same key value, then the index entry points to the *latest version with the key value* (see Figure 3). When we refer to the "version pointed to by the index entry" this is assumed to involve an indirection if the entry actually points to a VLE. Note that the key value can be obtained from the version that is being pointed to (possibly indirectly through the VLE) [LC86, DKO$^+$84] rather than explicitly storing the key values in the index. This is possible since the database is memory resident and pointer dereferencing is inexpensive.

For a primary index, the key value for an entry is basically the primary key value stored in the version that the entry points to. We assume for simplicity of presentation that user secondary indices allow duplicates, however we implement them using unique index code by the way the key is constructed. For a secondary index, the key value for an entry is obtained by concatenating the secondary key with the primary key for the item. This ensures that the key values in the index (both primary and secondary) are unique. This allows us to compare key values and handle cases where there are more items with a particular secondary key value than can fit in a single index node easily. Key values supplied by the user for a secondary index can be supplemented by minimum or maximum primary key values in a straightforward manner to maintain the semantics of the operation. Also, for a secondary index, using the primary key to disambiguate duplicates ensures that an update to the latest version that does not modify the secondary key value leaves the key value in the index itself unchanged. Thus, the index entry pointing to the old version simply needs to be *toggled* to point to the new version rather than deleted and reinserted.

Locks on keys are implemented by locking the item (its collection and primary key) that the index entry for the key points to. Thus, following the terminology in [ML92], we perform *data only locking*.

For delete, update and read operations belonging to update transactions, we assume that a pointer to a version or VLE has been obtained by traversing the index and a lock in the appropriate mode on the item is held. Also, before an item is inserted, a lock on the item is obtained. Finally, for a read operation belonging to a read-only transaction, we assume that a pointer to the version/VLE to be read has been obtained by traversing the index – no lock is needed. In this design, no latches are required by read-only transactions when searching a VLE chain for the correct version, and for update transactions, ensuring the consistency of the chain is piggybacked on the lock on the item itself for speed.

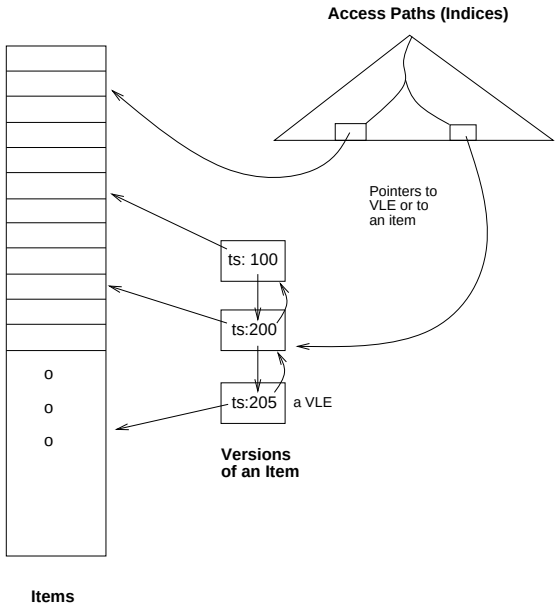

Figure 3: Pointers to Items and VLEs from Indexes

4.1 Update

The following protocol is followed when updating an item:

1. If a pointer to a version is passed as a parameter to update (that is, no VLE exists for the item), then two VLEs v_1 and v_2 are allocated. The timestamp in v_1 is set to $-\infty$ (a value smaller than the minimum of the timestamps of currently running read-only transactions), the version pointer is set to point to the item being updated and the next pointer is set to point to v_2. In v_2, the timestamp is initially set to ∞, ensuring a recoverable schedule since by definition no reader is old enough to read it. Also, the version pointer is set to point to the newly allocated version. For indices on attribute(s) for which the key value in the new version is the same as that in the old version, the pointer in the index to the item is simply toggled to point to v_2; for the remaining indices, the pointer to the version is toggled to point to v_1 and a new pointer to v_2 is inserted into the index. Thus, we ensure that for an item, for a particular key value, only one index entry points to a version of the item. While the update is being performed, read-only transactions either see the pointer to the version in the index or a pointer to v_1 or v_2. In all three case, as we will see shortly, the read-only transaction correctly reads or ignores the version of this item that it is supposed to read.

2. If a pointer to a VLE is passed as an argument to update, then a new VLE is allocated that points to the newly allocated version and has timestamp of ∞. The new VLE is linked at the (rightmost) end of the VLE chain for the item. For every index, for the key value in the new version, if a pointer to a previous version of the item with the same key value is contained in the index, then the pointer is toggled to point to the new VLE; else, a new pointer to the new VLE is inserted into the index.

4.2 Delete

Deletion of an item creates a "delete-VLE" (with timestamp ∞ and null version pointer) to record the deletion of the item. If a pointer to a VLE is passed as an argument to delete, then the delete-VLE is simply linked at the end of the VLE chain. If a pointer to a version is passed as an argument (that is, no VLE exists), an additional VLE with timestamp $-\infty$ and which points to this version is allocated, and the delete VLE is appended to it. For every index, the pointer to the item in the index is toggled to point to the new VLE for the item. Transactions that start after the commit of the transaction that deleted the item will find the delete-VLE when they attempt to access the item, and thereby detect that it is deleted.

4.3 Insert

We first consider the case that the primary key for the item (to be inserted) is already present in the primary index. If the pointer in the index is 1) directly to a version or 2) to a VLE such that the last VLE on the chain is not a delete-VLE, an error is returned (since an item with the same primary key value logically exists). If the duplicate is not detected, Update is simply invoked with the pointer (to the VLE) in the primary index and the contents of the new item to be inserted.

In case the primary key for the item is not contained in the primary index, a VLE v_1 is allocated, the timestamp is set to ∞ and the version pointer is set to point to the item. A pointer to v_1 is inserted into every index. In addition another VLE v_2 is allocated, the timestamp in v_2 is set to $-\infty$, the version pointer is set to null, and next pointer is set to v_1. The VLE v_2 is added to the logical ager's list when the transaction commits. The logical ager uses v_2 to access the VLE chain for the item and deletes all VLE information for the item if no updates to the item have been performed for a while. This is further described in Section 4.8.

4.4 Read (Read-Only Transactions)

From a pointer in the index, the item to be returned is determined as follows. If the pointer is a direct pointer to an item, then the item pointer is returned. Else, if the pointer is to a VLE, say v_1, then the VLE chain is traversed to determine the VLE, say v_2, with the largest timestamp less than or equal to the timestamp for the transaction. If no such VLE exists or the version pointer in the VLE is null, then null is returned. In case the key value for v_2 (in the index) differs from the key value for v_1, then too, null is returned (since in the transaction consistent database state for the read-only transaction, the item does not have a key value equal to that for v_1). Else, the version pointer in v_2 is returned. Note that if null is returned by the version manager to a read-only search, the appropriate action may be to continue the search after disqualifying this particular item.

4.5 Read (Updaters)

If read is passed a pointer to an item, then the item pointer is simply returned. Else, the version pointer contained in the last VLE on the VLE chain is returned (note that if the item has been deleted, then null is returned since the version pointer in a delete-VLE is null).

4.6 Transaction Commit

VLEs for versions created by a transaction and delete-VLEs allocated by the transaction are linked together in a separate chain for the transaction (VLEs with timestamp $-\infty$ are not in the chain).

When a transaction commits, for all VLEs in the transaction's chain, the timestamp in the VLE is set (from ∞) to $\mathsf{tsn}(T)$, as described in Section 2. In addition, the VLE preceding every VLE in the transaction's chain is appended to the logical ager's list. Thus, VLEs in the logical ager's list are sorted based on the timestamp of their successor versions.

4.7 Transaction Abort

For every version created by the transaction, all newly inserted entries in the indices are deleted. All index entries that were toggled by the transaction are reset to their initial value before they were toggled. Every VLE allocated by the transaction is first unlinked from the VLE chain, following which the VLEs and versions that were allocated by the transaction are placed on the physical ager's list. Note that the actions to be performed during transaction abort can be easily deduced from the contents of the VLE chain for VLEs on the transaction's chain. As a result, no explicit undo log records are generated in our scheme.

4.8 Logical Aging

The logical ager has a linked list of VLEs whose associated versions are candidates to be aged. As described earlier, a version is added to the list when its successor version is committed. The logical ager decides to age a version and its VLE when it does not find a reader with timestamp between the VLE's timestamp and the timestamp of the version following it in the VLE chain. We outline an efficient algorithm for determining the versions to age in the next subsection.

In our design, in addition to the task of placing versions of items that are no longer required on the physical agers list, the logical ager has a secondary function of checking if, on deletion of a version, only one version of that item remains. This version can then be *stabilized* by removing the VLE associated with it and toggling index entries to point to the version instead of the VLE.

A version/VLE is aged as follows. First, in order to ensure that no updaters are accessing the item, an X lock on the item is obtained. The primary key for the item is determined from the version pointed to by the VLE. In case the version pointer in the VLE is null, the VLE chain is traversed, and the primary key contained in one of the versions of the item is used (if no versions are found, then no lock is obtained since the item does not exist). The following actions are then performed.

1. If the version pointer in the VLE is not null, then for every index that contains a pointer to the VLE, if a previous version of the item contains the same key value as the current version, then the pointer to the VLE in the index is toggled to the previous version's VLE; else, the pointer to the VLE is deleted from the index.

2. If no other VLE precedes the VLE and a single VLE, say v', follows the VLE in the VLE chain for the item, then the item is stabilized by performing the following actions: 1) if the version pointer in v' is non-null, then the pointer to v' in every index is toggled to the version pointed in v' 2) v' is placed on the physical ager's list.

3. The VLE is unlinked from the VLE chain for the item and both the VLE and the version pointed to by it (if non-null) are placed on the physical ager's list following which the lock on the item is released.

4.9 Algorithm for Determining Versions to Age

In this subsection, we describe an efficient algorithm employed by the logical ager for detecting unneeded versions. The algorithm *adapts* to become more aggressive if memory is scarce. In a main-memory based versioning scheme, the efficiency of collecting old versions which are no longer needed is a tradeoff between space and time: if you spend less time to find unneeded versions, more of them sit around taking up space. This situation is quite different from disk-based systems, where space on each page is usually pre-allocated to hold versions of items on that page [MPL92, BC92a], and efforts to garbage collect versions can wait until memory on that particular page is needed. To balance these contradictory goals, our aging scheme uses simple parameters to vary the level of eagerness exhibited by the algorithm. By tying these parameters to statistics about memory usage, the garbage collection scheme has a low CPU overhead when memory is plentiful, yet becomes more aggressive if memory resources become scarce.

The aging algorithm assumes a timestamp-sorted list of active readers called readers, and a list of non-current item versions called oldver sorted by the timestamp of the transaction which created the *successor* version of the item. We call this the *finish* time of the version, though it would actually be obtained by following one link in the VLE list and examining the successor version's timestamp.

This algorithm works with a timestamp parameter, L, which determines how aggressively the logical ager will attempt to age old versions. Only versions whose finish timestamp is *smaller* than L will be considered for collection. This corresponds to the intuition that there are a significant number of "short" readers, and that it is more cost-effective to attempt to age versions which are older than the average age of these short readers. Of course, in a system with very few readers, or mostly very long readers, this parameter would be less meaningful.

On each pass, the logical ager simultaneously traverses the readers and oldver lists in a manner similar to a merge, using pointers to the current entry in the respective lists. The algorithm begins by initializing one pointer to the reader with the largest timestamp smaller than L in readers and one to the version with the largest finish timestamp smaller than L in oldver. It then works backwards to the beginning of the readers and oldver lists. A temporary set of versions of items, open, is maintained as a max-heap on the version timestamps. As an invariant, open contains versions which are not needed by any reader yet encountered.

At each step in the algorithm, the largest of three timestamps is selected: the finish time of the current version in the oldver list, the timestamp of the current reader, and the largest version timestamp in the set open. Depending on which of the three is the largest, one of the following actions is performed:

1. If the finish time from oldver is the largest, then the version is added to open, and the pointer is moved down the list by one (toward versions with earlier finish times).

2. If the timestamp of the reader is the largest, then all versions currently in open are needed by this reader, so open is discarded and none of the versions in it are aged.

3. If the timestamp from open is the largest, then it is not needed by any reader, and the version with the largest timestamp is aged (as described in Section 4.8) and deleted from open.

Once the earliest reader is encountered, the remaining versions are all aged.

The above algorithm ages any non-viable versions with finish time smaller than L by performing a single pass over readers and oldver. If L is increased, then the ager becomes more aggressive in attempting to free storage, at the expense of checking a longer list of transactions and versions, and increasing the likelihood that a version is checked for aging multiple times before actually being collected. Thus,

if memory is short, a larger value of L is appropriate. If L is decreased, all potential readers of a version are more likely to have finished before an attempt is made to reclaim that version's space.

4.10 Summary

The logical versioning scheme just described is designed to give good performance for readers and updaters while minimizing space usage and allowing latch-free traversal by read-only transactions. In particular, no space overhead is imposed on stable items which have only one version in the system. No latches are required by read-only transactions when searching a VLE chain for the correct version, and for update transactions the consistency of the chain is piggybacked on the lock on the item itself for speed. Logging is minimized by dispensing with undo logs which are physically encoded in the VLE chains. We have also outlined the services required of the index manager, which include insertion, deletion, toggling, and the ability to extract keys from pointers to versions (or indirectly through the VLEs). The design of one such index manager, for T-trees, which also supports latch free traversal by read-only transactions, is described in [BLR+95].

We must point out that our logical versioning scheme can easily be extended to handle the case when transactions have associate deadlines. Each transaction is assigned a priority based on its deadline (if transactions have values, then these can be incorporated into the priority assignment as described in [HSRT89, HCL93]). Since read-only transactions do not obtain locks, they are never involved in conflicts, and so are irrelevant from a concurrency control perspective. Updaters, however, do obtain locks and may be involved in conflicts. These conflicts can be resolved in favor of the transaction with the higher priority as is done in the 2PL-HP scheme [AGM88]. Alternately, if the transaction holding the lock has a lower priority, then it can *inherit* the priority of the blocked transaction as described in [HSRT91].

5 Related Work

In this section, we discuss related work on multi-version concurrency control schemes and the relationship of our work with deadline cognizant concurrency control algorithms proposed in the real-time database systems literature [SZ88, AGM88, AGM89, HCL90b, HCL90a, HSRT89, HSRT91, HCL93, Ram93]. A number of versioning schemes have been proposed for disk-based databases [BHR80, CFL+82, MPL92, AS89, BC92b]. Our logical versioning scheme is tailored for main-memory systems since it eliminates storage space overheads for items with a single version and it garbage collects old versions aggressively. We also present the actions performed on indices when items are updated.

The idea of using atomic updates to avoid latches while performing lookups in binary trees was originally proposed in [KL80]. We extend this work to T-trees and general tree structures, address transaction level concurrency control and recovery issues (see [BLR+95] for details) and show additional advantages from using these techniques in a multi-version concurrency control system. Schemes similar to our physical aging scheme have been presented in [ML82, SG88]. Our requirement of completely non-blocking readers, and techniques to interrupt long operations (e.g., scans) to allow efficient garbage collection distinguish our work.

Our work on making read-only transactions completely non-blocking and their execution times more predictable is complementary to the time-cognizant transaction processing schemes proposed in the real-time database systems literature [AGM88, AGM89, HCL90b, HCL90a, HSRT89, HSRT91, HCL93]. Our use of main-memory technology and version-based concurrency control schemes enable transaction running times to be estimated more accurately. Consequently, scheduling algorithms can generate

better schedules, that is, schedules in which more transactions meet their deadlines. Further, our logical versioning scheme can easily be extended using the schemes in [HCL93] to handle the case when transactions have associated values. Each transaction is assigned a priority which is a function of its deadline and value. Conflicts involving update transactions in which a higher priority transaction requests a lock held in a conflicting mode by a lower priority transaction can be resolved in one of several ways. Either the lower priority transaction can be aborted or it can inherit the priority of the blocked transaction thus enabling it to complete sooner than with its own priority.

The problem of supporting predictability in real-time database systems is also addressed in [KS96]. Transactions are classified into three categories: periodic transactions with hard deadlines, transactions with critical timing constraints and finally, real-time transactions with soft deadlines. Transactions of the first category are assumed to be completely predictable, that is, all data and run-time requirements are known in advance. Further, they are assigned the highest priority in the system. Transactions for whom a priori knowledge of resource requirements is not available are assigned lower priorities and belong to the latter two categories. The versioning algorithms proposed in this paper can aid in reducing the unpredictability of these latter transaction types.

In [LSLH98], the authors observe that in order to satisfy the timing constraints of real-time transactions, it may be desirable to relax the serializability requirement in RTDBSs. To this end, they propose a new notion of consistency, *view consistency*, for read-only transactions. In a nutshell, view consistency requires that for each read-only transaction, only the concurrent execution involving update transactions and the read-only transaction is serializable. Thus, it is possible for the overall schedule to be non-serializable and for different read-only transactions to percieve different serialization orders of update transactions. However, view consistency guarantees that every read-only transaction sees a consistent database state, that is, one that results due to the serial execution of some subset of update transactions.

6 Concluding Remarks

We have presented a design for multi-version concurrency control, recovery and index management in a main-memory database system. We have shown how this design supports real-time performance for read-only transactions by freeing them from obtaining locks (using logical versioning) *and latches* using the technique of *physical versioning*, a general method for eliminating reader's latches from tree-like data structures. Our design considers in depth the practical requirements of recovery and deadlock-free operation, fitting well with recovery schemes designed for the Dalí main memory storage manager [JLR+94]. Some of the salient features of our design are 1) read-only transactions do not obtain latches while performing lookups, 2) update transactions perform latch-free traversals on the tree, and 3) no physical undo log records are generated. We used the technique of *physical aging* to collect physically versioned information more quickly than versions of data items.

There are numerous benefits to making an entire class of read-only transactions in a real-time database environment completely non-blocking. Since transaction executions become more predictable, their running times can be estimated fairly accurately. Consequently, in addition to a transaction's deadline, a more accurate estimate of its execution time can also be taken into account when making scheduling decisions. For instance, for a certain aborted transaction before it is re-executed, it may be possible to conclude that it will not meet its deadline even if it had exclusive access to all the resources in the system – as a result, the transaction can be discarded early, thus resulting in better utilization of resources.

Variants of the logical and physical versioning schemes are implemented in Lucent's 2NCP prod-

uct, and the Dalí main-memory storage manager. We have also applied these techniques to design a concurrent implementation of T-trees, an index structure for main-memory systems, and demonstrated experimentally the performance improvement due to physical versioning in [BLR+95].

References

[AGM88] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *Procs. of the International Conf. on Very Large Databases*, 1988.

[AGM89] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions with disk-resident data. In *Procs. of the International Conf. on Very Large Databases*, 1989.

[AHU74] A. Aho, J. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[AS89] D. Agrawal and S. Sengupta. Modular synchronization in multiversion databases: Version control and concurrency control. *ACM SIGMOD Conf. on the Management of Data 89, (Portland OR), -Jun.*., May 1989.

[BC92a] P. Bober and M. Carey. On mixing queries and transactions via multiversion locking. In *Proc.IEEE CS Intl.Conf. on Data Engineering 8, Tempe, AZ.*, February 1992.

[BC92b] P.M. Bober and M.J. Carey. Multiversion query locking. In *Proceedings of the Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA) 18, Vancouver.*, August 1992.

[BG83] P.A. Bernstein and N. Goodman. Multiversion concurrency control — theory and algorithms. *ACM Transactions on Database Systems* ., 8(4):465–483, December 1983.

[BHR80] R. Bayer, H. Heller, and A. Reiser. Parallelism and recovery in database systems. *ACM Trans. on Database Systems*, 5(2):139–156, June 1980.

[BLR+95] P. Bohannon, D. Leinbaugh, R. Rastogi, S. Seshadri, A. Silberschatz, and S. Sudarshan. Logical and physical versioning in main memory databases. Technical Report 113880-951031-12, AT&T Bell Laboratories, Murray Hill, 1995.

[CFL+82] A. Chan, S. Fox, W-T.K. Lin, A. Nori, and D.R. Ries. The implementation of an integrated concurrency control and recovery scheme. In *ACM SIGMOD Conf. on the Management of Data 82, Orlando FL.*, pages 184–191, June 1982.

[DKO+84] D. J. DeWitt, R. Katz, F. Olken, D. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *Proc. ACM-SIGMOD 1984 Int'l Conf. on Management of Data*, pages 1–8, June 1984.

[GL92] V. Gottemukkala and T. Lehman. Locking and latching in a memory-resident database system. In *Proceedings of the Eighteenth International Conference on Very Large Databases, Vancouver*, pages 533–544, August 1992.

[Had88] Thanasis Hadzilacos. Serialization graph algorithms for multiversion concurrency control. In *Proceedings of the ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*, pages 135–141, March 1988.

[HCL90a] J. Haritsa, M. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1990.

[HCL90b] J. Haritsa, M. Carey, and M. Livny. On being optimistic about real-time constraints. In *Proceedings of the ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*, 1990.

[HCL93] J. Haritsa, M. Carey, and M. Livny. Value-based scheduling in real-time database systems. *VLDB Journal*, 2(2):117–152, 1993.

[HSRT89] J. Huang, J. Stankovic, K. Ramamritham, and D. Townsley. Experimental evaluation of real-time transaction processing. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1989.

[HSRT91] J. Huang, J. Stankovic, K. Ramamritham, and D. Townsley. On using priority inheritance in real-time databases. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1991.

[IKK90] T. Ibaraki, T. Kameda, and N. Katoh. Multiversion cautious schedulers for database concurrency control. *IEEE Transactions on Software Engineering (SE), ; ACM Computing Reviews 9012-0981.*, 16(3), March 1990.

[JLR+94] H.V. Jagadish, Dan Lieuwen, Rajeev Rastogi, Avi Silberschatz, and S. Sudarshan. Dali: A high performance main-memory storage manager. In *Procs. of the International Conf. on Very Large Databases*, 1994.

[KL80] H.T. Kung and P.L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems* ., 5(3):354–382, September 1980.

[KS96] Y. Kim and S.H. Son. Supporting predictability in real-time database systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, 1996.

[LC86] T.J. Lehman and M.J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA) 12, Kyoto.*, pages 294–303, August 1986.

[LSC92] T. Lehman, E. J. Shekita, and L. Cabrera. An evaluation of Starburst's memory resident storage component. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):555–566, December 1992.

[LSLH98] K. Lam, S.H. Son, V. Lee, and S. Hung. Using separate algorithms to process read-only transactions in real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1998.

[ML82] U. Manber and G.D. Ladner. Concurrency control in dynamic search structures. *ACM Proc.on Database Systems, Boston.*, pages 268–282, April 1982.

[ML92] C. Mohan and F. Levine. Aries/im an efficient and high concurrency index management method using write- ahead logging. In *ACM SIGMOD Conf. on the Management of Data 92, San Diego.*, June 1992.

[MPL92] C. Mohan, H. Pirahesh, and R. Lorte. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *ACM SIGMOD Conf. on the Management of Data 92, San Diego.*, June 1992.

[Ram93] K. Ramamritham. Real-time databases. *International Journal of Distributed and Parallel Databases*, 1:199–226, 1993.

[Ree78] D. P. Reed. Naming and synchronization in a decentralized computer system. Technical Report MIT-LCS-TR-205, Massachusetts Institute of Technology, Cambridge, September 1978.

[SG88] D. Shasha and N. Goodman. Concurrent search structure algorithms. *ACM Transactions on Database Systems , no.1.*, 13:53–90, March 1988.

[SGM90] K. Salem and H. Garcia-Molina. System M: A transaction processing testbed for memory resident data. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):161–172, March 1990.

[SZ88] J. Stankovic and W. Zhao. On real-time transactions. *ACM Sigmod Record*, 17(1):4–18, 1988.