Logical and Physical Versioning in Main Memory Databases

Rajeev Rastogi[1] **S. Seshadri**[2] **Philip Bohannon**[1,3] **Dennis Leinbaugh**[1]
Avi Silberschatz[1] **S. Sudarshan**[2]

[1]Bell Laboratories, Murray Hill, NJ
{rastogi,bohannon,avi}@bell-labs.com,
dleinbaugh@lucent.com

[2] Indian Institute of Technology
Mumbai, India
{seshadri,sudarsha}@cse.iitb.ernet.in

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. These schemes enable a system to *guarantee* that updaters will never interfere with read-only transactions, and that read-only transactions will not be delayed as long as the operating system provides them with sufficient cycles. Our contributions include several space saving techniques for the main memory implementation. We extend the T-tree index structure (designed for main-memory databases) to support concurrent access and latch-free traversals, and demonstrate the performance benefits of our extensions. Some of these schemes have been implemented on a widely-used software platform within Bell Labs., and the full scheme is implemented in the Dalí storage manager.

1 Introduction

While disk-based databases exhibit improved performance if the entire database can fit in the main memory buffer cache, a main memory database (MMDB) (e.g. [SGM90, JLR+94]) 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.

Many applications in telecommunications require very fast and predictable response times for transactions and, in particular, for read-only transactions. 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. However, 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 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 used timestamps to serialize readers as well as writers, but more recent *multi-version locking* schemes [CFL+82, BC92, MPL92] use timestamps to serialize read-only transactions with respect to updaters, allowing them to use old versions without locking, while requiring updaters to perform locking to serialize themselves with respect to other updaters.

However, none of the above techniques guarantees complete isolation of read-only transactions from update transactions in a system, since the access path

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

[3]A Ph.D. candidate at Rutgers University.

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 processes 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 versioning of data items (which we refer to as *logical versioning*). Our implementation is optimized for main memory, and reduces the storage space overhead required to keep track of versions as compared to versioning schemes for disk databases. 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.

The remainder of the paper is organized as follows. In sections 2 and 3, we provide an overview of logical and physical versioning, respectively. In Section 4, we develop concurrency control schemes for operations on T-trees. In Section 5 we compare the performance of T-tree algorithms with and without physical versioning. In Section 6, we discuss related work, and in Section 7, we give our conclusions and directions for future work.

We do not address recovery issues in this paper – a comprehensive treatment of this can be found in [BLR+95].

2 Logical Versioning

We refer to the (well-known) idea of maintaining multiple versions for concurrency control as logical versioning. 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.

When a version of a data item is no longer needed by any (current or future) read-only transaction, it 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.

2.1 Assigning Timestamps

To meet our design goal of read-only transactions never even acquiring a latch, read-only transactions must read the logical timestamp counter without any latching. To do so consistently,

1. the counter itself must fit in a word,[1] ensuring that a read which does not obtain latches is atomic with respect to the update,

2. the timestamp counter must be incremented by a transaction only after the stamping process is complete and the transaction has committed.

Updater transactions modify the logical timestamp counter during commit processing. Every such transaction must obtain an X latch (ignored by read-only transactions) on the logical timestamp counter before accessing it. The latch must be held until all the versions have been stamped and the counter has been incremented.

[1] Longer counters can be handled by indirecting through a pointer and not modifying the counter in place.

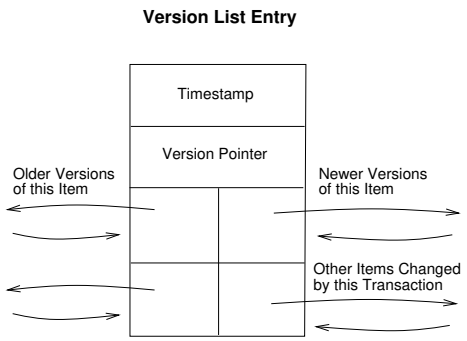

Figure 1: Structure of a Version List Entry

2.2 Version List Entries

In most disk-based schemes [BC92, 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, and space is an important constraint in main-memory databases, our goal was to impose essentially zero space overhead on data items due to versioning.

Our design uses 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. 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-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, if a transaction aborts, the versions created by the transaction can be efficiently determined and deleted.

It is important that we can determine if a pointer points to a data item or a VLE. In our implementation, data and VLE are stored on distinct segments (alternatively, distinct pages can be used) and a single bit per segment lets us determine whether the pointer is to a data item or a VLE. An item that has only one version is stored as is without a VLE, resulting in no space overhead (except the per-segment bit) due to versioning on cold items. 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 pointer to an item could be a direct pointer or a pointer to a VLE. The code to dereference an item determines the type of the pointer, and fetches the appropriate version, if more than one

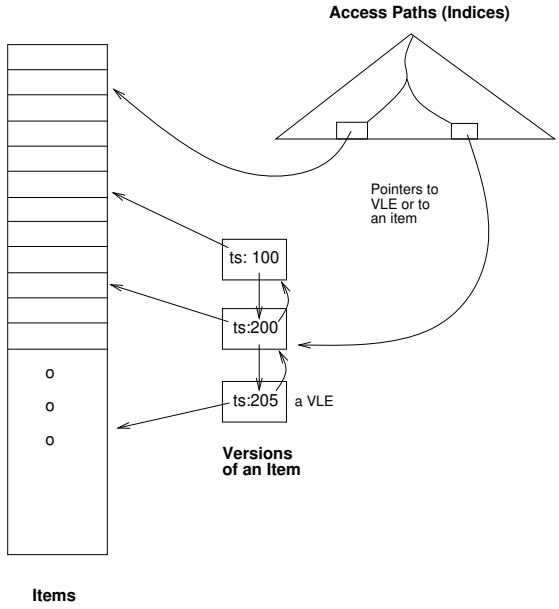

Figure 2: Pointers to Items and VLEs from Indexes

version exists.

We do not discuss the implementation of logical aging (which detects when a version is no longer needed) due to lack of space — see [BLR+95] for details. When a version is deleted, the corresponding VLE is also deleted. One remaining detail is to handle the case when only one version of an item is left. In this case, all pointers to the VLE must be updated to point to the data item, and then the VLE can be deleted.

2.3 Interaction with Indices

We next discuss how our logical versioning scheme for items can be combined with indices. The index stores entries for all existing versions of an item. An index entry for an item stores a pointer to an item, or to the VLE of the version if more than one version of the item exists (see Figure 2). As discussed earlier, when dereferencing an item, the pointer type is determined, and an extra level of indirection used if required. 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. The index need not store key values — the key values can be obtained from the data item version that is being pointed to, since in-memory pointer dereferencing is inexpensive [DKO+84].

Update to an item causes a new VLE to be allocated that points to the newly allocated version. 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 updated to point to the new VLE; else, a new pointer to the new VLE is inserted into the

index.

Read-only transactions traverse the index to obtain a pointer to an item or a VLE with the appropriate key value. 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 for the data item 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 key value for v_2 differs from the key value for v_1, then 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. A small extension consisting of VLEs with null pointers is used to handle deletes. More details about interaction with indices can be found in [BLR+95].

3 Physical Versioning

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. Trees structures, in particular, lend themselves to efficient physical versioning, allowing readers to see an operation-consistent state of the tree without obtaining any latches. In other words, the operations are each performed atomically with respect to readers.

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. 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 and T-trees can be easily defined. For example, the affected component in a B-tree split would be the path between the node in which the insert took place and the highest node in the tree to which the split propogated. Physical versioning can also be used on hash-tables with chaining, since they can be considered as a forest of lists, and a list is a special case of a tree.

We use the term *physical aging* to denote the process of reclaiming space occupied by older copies of data that have been physically versioned. The old physical versions of the data have to be preserved as long as a read-only transaction can attempt to read the 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 during an operation cannot be physically aged for the *duration of the reading operation*. Contrast physical aging with logical aging, where a version of a data item cannot be aged as long as a transaction may need to access it.

We associate a *physical timestamp* with each read-only transaction. The physical timestamp is ∞ if the transaction is not currently performing any operation. It is set to the 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 physical version into an physical ager's list by appending to the list an entry containing a pointer to the physical 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 ager then can de-allocate the space for an older physical version once no transaction has a physical timestamp smaller than the version's physical timestamp.

4 T-trees

In this section, we describe the algorithms for performing lookups, inserts and deletes from a T-tree index with logical and physical versioning. The salient features of our concurrency control scheme for T-trees are:

- Read-only transactions do not obtain any latches or locks.
- The tree traversals of update transactions do not obtain latches while locating the node to be updated and obtain latches only when actually performing the update. Thereby, the number of latches acquired by tree updates can be reduced, a useful property in a main memory database where latche acquisition may be relatively expensive.
- Update operations that result in modifications to a single T-tree node can execute concurrently.
- Rotations due to inserts can take place concurrently.

4.1 Overview of T-trees

In [LC86], the authors proposed T-trees as a storage efficient data structure for main memory databases. T-trees are based on AVL trees proposed in [AHU74]. In this subsection, we provide an overview of T-trees. For a detailed description, the reader is referred to [LC86]. Like AVL trees, the height of each node's subtrees may differ by at most one. A T-tree differs from an AVL tree in that each node, instead of storing a single key value, stores multiple key values in a sorted order. The leftmost and the rightmost key value in a node define the range of key values contained in the node. Thus, the left subtree of a node contains only key values less than the leftmost key value, while the right subtree contains key values greater than the rightmost key value in the node. A node with both a left and a right child is referred to as an *internal node*, a node with only one child is referred to as a *semi-leaf*, and a node with no children is referred to as a *leaf*. In order to keep occupancy high, every internal node has a minimum number of key values that it must contain (typically $k-2$, if k is the maximum number of keys that can be stored in a node). However, there is no occupancy condition on the leaves or semi-leaves.

Searching for a key value in a T-tree is relatively straightforward. For every node, a check is made to see if the key value is bounded by the leftmost and the rightmost key value in the node; if this is the case, then the key value is returned if it is contained in the node (else, the key value is not contained in the tree). Otherwise, if the key value is less than the leftmost key value, then the left child node is searched; else the right child node is searched. The process is repeated until either the key is found or the node to be searched is null.

Insertions and deletions into the T-tree are a bit more complicated. For insertions, first a variant of the search described above is used to find the node that bounds the key value to be inserted. If such a node exists, then if there is room in the node, the key value is inserted into the node. If there is no room in the node, then the key value is inserted into the node and the leftmost key value in the node is inserted into the left subtree of the node (if the left subtree is empty, then a new node is allocated and the leftmost key value is inserted into it). If no bounding node is found then let N be the last node encountered by the failed search. If N has room, the key value is inserted into N; else, it is inserted into a new node that is either the right or left child of N depending on the key value and the leftmost and rightmost key values in N.

Deletion of a key value begins by determining the node containing the key value, and the key value is deleted from the node. If deleting the key value results in an empty leaf node, then the node is deleted. If deleting the key value results in an empty semi-leaf node, then the node is merged with its child. If the deletion results in an internal node containing fewer than the minimum number of key values, then the deficit is made up by moving the largest key in the left subtree into the node, or by merging the node with its right child.

In both insert and delete, allocation/de-allocation of a node may cause the tree to become unbalanced and rotations (RR, RL, LL, LR) may need to be performed in a manner similar to rotations in AVL trees. Balancing starts from the newly allocated node (or the parent of the deleted node) proceeds upwards towards the root, and stops on reaching a node that is balanced, or a node where the heights differ by one, and can be made equal by rotation. Rotations occur at the intermediate nodes. Details may be found in [LC86].

4.2 Latches and Versioning

Each node in the tree has a latch associated with it which is obtained in exclusive mode to prevent concurrent updates to the node. Due to physical versioning, the latch on a node is never obtained in shared mode. The tree itself has a *tree latch*, which is obtained (instead of node latches) in exclusive mode by certain operations. All update operations acquire the tree latch in shared mode.

Each node contains a *version bit* that indicates if the node is versioned (physical versioning). This bit is 1 if a newer copy of this node has been linked into the tree

in its place. Only updaters read and write the version bit. The act of *marking a node as versioned* consists of setting its versioned bit to 1 and adding the node to the physical ager's list. The node is then said to be *versioned*.

A new version of a node is created only when a key value is inserted or deleted from the node, or the node is involved in a rotation. Updates to balance information and child pointers in a node are performed directly on the node, and no new version is created since read-only transactions never look at the balance information and the child pointers are changed atomically.

4.3 Find

Find is the algorithm for traversing the tree to find the smallest key greater than or equal to a search key (see Figure 3). (Other search modes (e.g., >,=) can be supported via straightforward extensions.) Find takes the following arguments: stack, which contains the nodes on the path from the root to the current node of Find (Find starts tree traversal from the top node of the stack; if the stack is empty, the root of the tree is assumed); search_key, the key value being sought; lock_mode, a flag which indicates whether an exclusive lock, shared lock, or neither should be obtained on the key returned by Find; and latch_mode, a flag which if True indicates that the node at which Find terminates should be latched exclusively.

When Find is called on behalf of a read-only transaction lock_mode is None (indicating no lock), and latch_mode is false. In this case, no latches or locks are obtained, and no checks are made to determine if nodes are versioned. The reason for this is that a read-only transaction only needs to see the effects of updates that completed before it began. Update transactions, on the other hand, look up a key value in the index by invoking Find with lock_mode set to shared and latch_mode set to False. In procedure Find, right_ancestor(stack) is the topmost node in stack whose left child is also in stack.

Whether called on behalf of updaters or readers, the Find procedure performs a "fuzzy" traversal of the tree. By fuzzy, we mean that the Find algorithm does not obtain latches on its way down and does not check whether a node has been versioned until it reaches the node containing the satisfying key (sat_key) or a leaf or a semi-leaf node that should contain the search key (recall that all searches are greater than or equal to). After obtaining appropriate locks and latches based on input parameters (note that the lock is obtained before the latch is obtained to prevent deadlocks involving latches and locks), *validation* is performed to determine if the satisfying key value is indeed the key value to be returned. The reason to perform

```
Find(stack, search_key, lock_mode, latch_mode) {
  Proceed down the tree, beginning with the topmost
    node in stack, pushing nodes onto stack until a node
    bounding search_key is found, or until the next node
    to be visited is null;
  node = top of stack; /* at end of the above traversal */
  If search_key <= max_key(node) Then
     sat_key = smallest key in node  >=  search_key;
  Else sat_key = smallest key in right_ancestor(stack);
  If lock_mode == None and latch_mode == False Then
     return (sat_key, ptr in index entry for sat_key);
  If lock_mode not equal to None Then
     obtain appropriate lock on sat_key;
  If latch_mode == True Then
     obtain S latch on tree;
     obtain X latch on node;
  /* Validate node before returning */
  If (node is versioned)
     or (search_key  <  min_key(node) and
           left child not equal to null)
     or (search_key  >  max_key(node) and
           (right child not equal to null or
             right_ancestor(stack) is versioned)) Then
     Release lock and latches just obtained;
     Return Find(LSA(stack), search_key, lock_mode,
                 latch_mode);
  Else return (sat_key, ptr in index entry for sat_key);
}
```

Figure 3: The basic find algorithm

validation is that concurrent updaters may have inserted/deleted index entries while Find was obtaining locks/latches. Since every updater creates a new version of a node when inserting/deleting an index entry into/from the node, Find first checks to see if node has been versioned. Even if node were not versioned, if search_key < min_key(node), then a non-null left child of node could contain a newly inserted key value between search_key and min_key(node), and this (instead of min_key(node)) would be the appropriate key value to be returned by Find. Similarly, if search_key > max_key(node), then a right child may be added to node or the smallest key value in right_ancestor(stack) may be deleted, and thus it would no longer be the appropriate key value to return.

If any of the three validation conditions do not hold, Find restarts from the Lowest Stable Ancestor (LSA) in stack. The LSA is the node farthest from the root of the tree (and thus the highest node in stack) that has not been versioned since it was visited by the find. LSA(stack) is obtained from stack by popping each node and checking it's versioned bit until an unversioned node is found (in case all nodes in stack are versioned, then they are all popped and the latest version of the root node is pushed onto stack). Restart-

ing from the LSA is an optimization (we could restart at any node), and the intuition for it is based on the observation that no target key could "escape" from a subtree without modifying, and therefore versioning, the root of that subtree.

Find can be further optimized by checking if node is versioned before obtaining any locks or latches – this way, if node was versioned, the overhead of obtaining locks and latches can be avoided, and Find can restart earlier.

Index scans can be implemented by caching the key value returned by the last Find call and the value of stack at the end of the last Find operation (in an iterator structure), and then repeatedly invoking FindGT, a variant of Find which locates a strictly larger key, with the cached values of stack and the key value (lock_mode and latch_mode are set as for the first Find call for read-only and update transactions).

We describe the insert operation next; the deletion operation and correctness arguments for all the operations can be found in [BLR+95].

4.4 Insert

We next describe the insert procedure along with concurrency control and details of physical versioning. The concurrency control scheme described provides a high degree of concurrency; however, there are simplifications that provide lower concurrency but have lower latching overheads. A performance comparison of these alternatives is described in Section 5.

Insert first invokes Find with the key value to be inserted key_val, and input parameters lock_mode set to exclusive and latch_mode set to True (stack is set to the root of the tree). This ensures that an X lock on the *next key* value is obtained and a latch on the node involved in the insert is also held. Note that an X lock on key_val is already held when the insert call is made.

Let N be the node on which Find obtains an X latch. We consider the following three cases:

1. N bounds key_val and has room:
A copy of N, say N', is created and key_val is inserted into it. A latch on N's parent is then obtained.

Note that N's parent can be determined from stack. In order to ensure that updates are reflected in the most current version of the tree, it is important that N's parent must not be an old version. Thus, after the latch on N's parent is obtained, it is checked to see if it has been versioned. If this is the case, then (after releasing the latch), the tree is retraversed from the root to N to determine N's most current parent, and a latch on N's most current parent is obtained. This process is repeated until N's parent is found to be not versioned.

Finally the pointer to N is updated to point to N'. Node N is then marked as versioned and all latches are released.

2. N does not bound key_val:
In this case the node is a leaf or a semi-leaf. If there is room in N, then key_val is inserted as described in Case 1. Else, a new node containing key_val is allocated, a latch on the node is obtained and the left/right child of N (as appropriate) is set to point to the newly allocated node.

3. N bounds key_val and does not have room:
If the left child of N is null, then two nodes N_1 and N_2 are allocated: N_1 is a copy of N containing key_val but not containing the leftmost key in N and the left child of N_1 is set to N_2. N_2 simply contains the leftmost key in N. Latches are obtained on both N_1 and N_2. After obtaining a latch on N's parent, the pointer to N is updated to point to N_1 and N is marked as versioned.

If the left child of N is not null, then after releasing the latch on N, the tree latch is obtained in exclusive mode. Now if N has been versioned or its left child has become null in between releasing the latch on N and obtaining the tree latch, the tree latch is released and insert restarts again by invoking Find from the LSA with latch_mode equal to True and lock_mode equal to None (a lock on the next key value is already held). Otherwise (i.e., N has not been versioned and its left child remains non-null) the following actions are taken.

Let N_1 be the node that contains the largest key value in the left subtree of N. If N_1 has room, then a copy of N_1 is made, the leftmost key value in N is inserted into the copy, the pointer in N_1's parent is updated to point to the new version, and N_1 is marked as versioned (we do not need a node latch here since we already hold the tree latch.) If N_1 has no room, then a new node containing only the leftmost key value in N is allocated and N_1's right child is set to point to the newly allocated node. After this is completed, a copy of N is made from which the leftmost key value is deleted, key_val is inserted and N's parent's pointer to N is updated to point to the new copy, following which N is marked as versioned.

The lock on the next key value is released at the end of the insert procedure once the key has been inserted, as in [Moh90].

Note in Step 3 above that, by inserting the leftmost key value in N into N's left subtree before deleting it from N, we ensure that any Find traversing the tree will see the key. A Find or an index scan may however see the key twice. For Find, this is not a problem since the traversal would have followed the same path irrespective of whether it encountered N or its new copy. For an index scan, this case can be handled by ignoring key values that are less than or equal to the previous

key value returned.

4.4.1 Balancing

In case a new node that is not a version of an existing node is allocated, the T-tree may need to be balanced. Balancing is done by traversing the tree upwards from the lowest unbalanced node, and performing rotations as appropriate.

The insert procedure described above ensures that every time a new node is allocated, latches are obtained and held on both the newly allocated node and the parent, or a tree latch is held. In case a tree latch is held, the traversal upwards toward the root is simply performed as described earlier in Section 4.1 (the only difference is that every time a parent node is accessed, a check is made to see if it is versioned, and if it is, then the tree is retraversed in order to determine the parent). In the case that the tree latch is not held, then before a parent node is examined to determine if it can be rotated, a latch is obtained on it (retraversing may be required if, after obtaining the latch, it is determined that the parent has been versioned).

Note that latches on tree nodes are obtained in a bottom-up fashion. Furthermore, no node latches are held when an attempt is made to acquire the tree latch. Thus, a deadlock involving only latches is not possible. (Locks are not acquired while holding a latch, so latch-lock deadlocks are not possible either.)

While traversing the tree upwards toward the root, balances on the appropriate nodes on the path are adjusted to account for the newly allocated node. Note that balance information can be updated in place, since readers never examine balance information. While performing a rotation, physical versioning only requires that the three nodes involved in the rotation are copied.

5 Performance Results

In order to determine the effects of physical versioning and latching overheads on performance, we implemented four variations of the T-Tree operations find, insert, and the rotation operations needed for rebalancing. (In each case, insert invokes find to determine the target node.) The four variations corresponded to whether physical versioning was used or not and the granularity at which latches were obtained (node level as is common in disk based systems or tree level as suggested by [LC86]) and are described below:

1. **Tree latch with no versioning**: A single latch at the granularity of the tree itself is obtained in X mode by inserts and S mode by finds. No physical versions of nodes are created by inserts.

2. **Node latch with no versioning**: In addition to the tree latch, a latch per node is maintained. Find obtains the tree latch in shared mode and performs latch crabbing when traversing the tree (obtaining each node latch in shared mode). For simple inserts that require no *structure modification*, a shared latch is obtained on the tree and exclusive latches are obtained on the nodes being modified. If a structure modification such as a rotation is required, an exclusive latch on the tree is obtained instead of node latches. No physical versions of nodes are created.

3. **Tree latch with physical versioning**: Physical versions of nodes are created – as a result, finds do not obtain any latches. Inserts, however, do obtain an exclusive latch on the tree before performing any updates.

4. **Node latch with physical versioning**: Physical versions of nodes are created and finds do not obtain any latches. Inserts obtain an exclusive latch on the tree if structure modifications take place; else, they simply obtain a shared latch on the tree and an exclusive latch on the updated node(s).

In each case, the T-Tree was configured to have 10 keys in each node. The keys for insert and find were integers uniformly chosen from the range 0 to 2,000,000. The percentage of inserts was varied from 1% to 75%. For each percentage of inserts value, the running time was 5 minutes, and the throughput measured was the sum of the total number of lookups and inserts performed. The experiments were performed on a Sun SPARCstation 20 with 2 processors and 256 MB of RAM.

In order to estimate the overhead of obtaining latches and performing physical versioning, we first conducted our experiments with a single process. For very low percentage of inserts (1, 2 and 4%), the physical versioning schemes perform the best since no latches are obtained by tree traversals. However, as the percentage of inserts goes beyond 4%, their performance falls below that of the tree latch and no versioning scheme, due to the high cost of creating versions. The node latch with no versioning scheme performs the worst inserts due to the high cost of latch crabbing when traversing the tree.

With 4 processes, we are also in a position to measure the effects of the increased concurrency that results due to node level latches and physical versioning. The tree latch and physical versioning scheme outperforms all the other schemes due to latch-free traversals and low latching overheads for inserts. Furthermore, as long as the percentage of inserts is below 30%, the node latch and physical versioning scheme outperforms

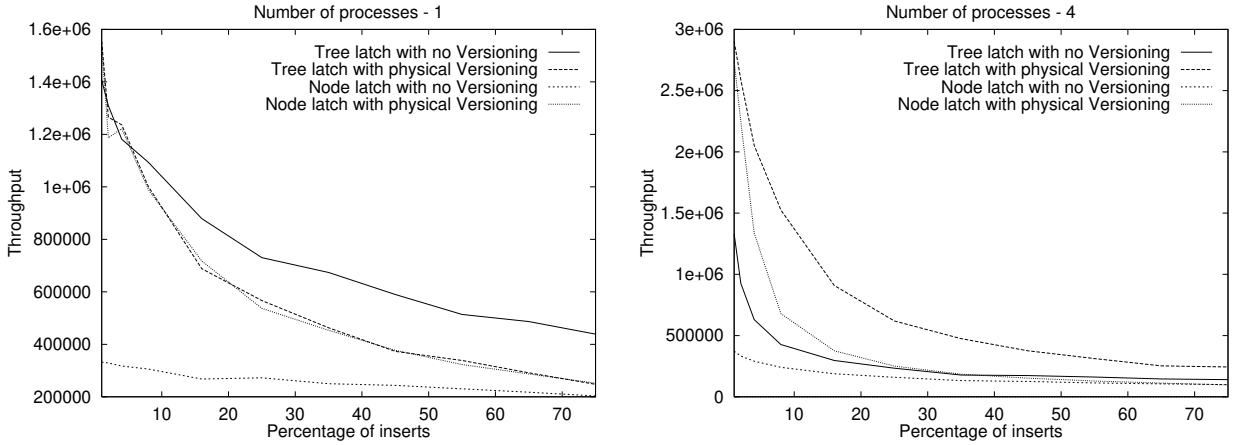

Figure 4: Throughput v/s percentage of inserts

the tree latch and no versioning scheme due to the enhanced concurrency and decreased latching overheads (since tree traversals do not obtain latches when physical versioning is used). Beyond 30% inserts, however, due to the overhead of creating versions and additional latches obtained by inserts, the performance of the node latch and physical versioning scheme falls below that of the tree latch and no versioning scheme. The node latch and no versioning scheme performs the worst due to excessive latching overheads.

6 Related Work

In this section, we discuss related work on main memory databases, multi-version concurrency control schemes and concurrency control schemes for indices. A number of versioning schemes have been proposed for disk-based databases [CFL+82, MPL92, BC92]. Our logical versioning scheme is tailored for main memory systems since it eliminates storage space overheads for items with a single version, and allows latch-free traversal of version control information by read-only transactions. In addition, our schemes include the interaction between versioning and index management.

Of the disk-based schemes, our logical versioning scheme is most similar to [CFL+82], in which a linked list of versions is maintained and aged versions are collected from a single pool. However, versioning in this design is at the page level, the garbage collection is very simplified for disk I/O considerations, and indexing problems are not considered. In [BC92], the authors extend the scheme in [CFL+82] to record-level versioning by allocating part of each page as a "version area" and clustering the versions of an item together on the same page [BC92, MPL92]. However, the optimization of clustering versions in the same page as the stable copy of the item is not required in a main memory database since there is no extra cost to accessing a different page and thus the schemes in [BC92, MPL92] would be wasteful of storage space in a main memory environment.

Among the multi-versioning schemes proposed, only [MPL92] considers the interaction between versioning and indexing. However, the scheme in [MPL92] preallocates space for information about a fixed number of versions in index nodes, adding a substantial amount of space overhead even for non-versioned items.

We next shift our attention to schemes for performing concurrent operations on B trees and binary trees that have been proposed in the literature, such as [SG88, BS77, Moh90, ML92, KL80, ML82]. All of the schemes, excepting [KL80, ML82] and [SG88] require traversals to obtain latches on each node.

These two schemes implement forms of physical versioning. However, the index techniques of [KL80] do not address concurrency control issues needed to implement transaction semantics, while the treatment of [ML82] requires preordering all accesses to a tree by an entire transaction by key value.

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 issues (ignored in [KL80]) 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 distinguish our work. In [BLR+95] we describe techniques to interrupt long operations (e.g., scans) to allow old physical versions to be reclaimed earlier.

The idea of using atomic actions to avoid latching is related to the study of wait-free data structures in operating systems, e.g. [Her89], which attempt to ensure that some process will make progress on the data structure, even if others are blocked. However, transactional semantics, which is central to our work, is not

considered.

The notion of performing next key locking and validation after obtaining a lock was presented for B+ trees in [Moh90, ML92]. However, in order to prevent insert/delete operations from taking place in a subtree that is involved in a *structure modification* (e.g., split) and at the same time, to permit traversals (that obtain latches) to execute concurrently on the subtree, a tree latch is obtained in exclusive mode during structure modifications. This could hurt concurrency since no two structure modification operations can execute concurrently. In our scheme, on the other hand, many structure modifications (e.g., balancing during inserts) obtain and retain only local latches on updated nodes until the structure modification completes, and all could do so since our tree latch is an optimization to reduce the number of latches, and is not otherwise involved in correctness. Further, stucture modification does not block traversals since in our scheme, traversals do not obtain any latches.

7 Conclusion

We have presented a design for multi-version concurrency control and index management in a main memory database system. We show how to support real-time performance for read-only transactions by freeing them from obtaining locks, by using logical versioning, *as well as* latches, by using physical versioning.

We have 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. 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) concurrent rotations on the tree are possible. Our performance results indicate that latch-free traversals enable our scheme to outperform other schemes. Both the logical and physical versioning schemes have been implemented in the Dalí main memory storage manager.

References

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

[BC92] P. Bober and M. Carey. On mixing queries and transactions via multiversion locking. In *Procs. IEEE Intl. Conf. on Data Engineering*, February 1992.

[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.

[BS77] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9(1):1–21, 1977.

[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 *Procs. of the ACM SIGMOD Conf. on Management of Data*, 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. *Procs. of the ACM SIGMOD 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 *Procs. of the International Conf. on Very Large Databases*, pages 533–544, August 1992.

[Her89] Maruice Herlihy. A methodology for implementing highly concurrent data structures. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, March 1989.

[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.

[LC86] T.J. Lehman and M.J. Carey. A study of index structures for main memory database management systems. In *Procs. of the International Conf. on Very Large Databases*, pages 294–303, August 1986.

[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 *Procs. of the ACM SIGMOD Conf. on Management of Data*, June 1992.

[Moh90] C. Mohan. Aries/kvl: A key-value locking method for concurrency control of multiaction transactions operating on btree indexes. In *Procs. of the International Conf. on Very Large Databases*, September 1990.

[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 *Procs. of the ACM SIGMOD Conf. on Management of Data*, June 1992.

[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.