RC24658 (W0810-014) October 2, 2008
Computer Science

IBM Research Report

Highly Concurrent B-Trees Using Atomic Blocks

Rajesh Bordawekar
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

J. Eliot B. Moss
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Highly Concurrent B-Trees Using Atomic Blocks

Rajesh Bordawekar
IBM T. J. Watson Research Center
Hawthorne, NY 10532
bordaw@us.ibm.com

J. Eliot B. Moss
Department of Computer Science
University of Massachusetts, Amherst, MA 01003
moss@cs.umass.edu

***Abstract*— We present a new highly-concurrent B^{link}-tree algorithm and discuss our implementation of it. Our design is novel in that it supports high concurrency while using atomic blocks in the implementation. Atomic blocks impose a discipline of static declaration of regions in which the system enforces atomicity of accesses. However, their static structure precludes lock coupling down through the levels of the tree, the usual method for traversing concurrent B-trees. We consider atomic blocks because of their software engineering advantages over unstructured use of locks. For example, it is easy to see that our design is deadlock-free.**

In our approach, structure modifying operations (SMOs), such as B-tree node splits, occur as separate deferred, even asynchronous, operations at each affected level. This increases concurrency and leads to interesting data structure invariants and traversal algorithms. Our fine-grained concurrency approach locks as few nodes as possible at a time, most commonly just one node, but sometimes two or three. We also present coarser-grained strategies, which trade off locking overhead versus maximum concurrency. Our design further supports cursors, which avoid traversing the tree as much when a series of operations has locality. In sum, our design contributions consist in: using atomic blocks, deferred structure modifying operations, fine-grained locking, adjustable lock granularity, and support for cursors. We further include some informal arguments on the correctness of our design.

I. Introduction

B-trees [1] are a fundamental data structure, taught in most courses on data structures, and significant because of their $O(\log n)$ behavior for lookup, insert, delete, find next higher key, etc. They are also of great commercial importance for indexing in database systems [2], [3], which raise the question of their use in a concurrent environment.

The emergence of commodity parallelism makes concurrent B-trees of interest for a variety of other software. Concurrent use of a B-tree requires that one control access to nodes, typically using locks. One proceeds from the root of the tree toward the leaves, locking individual nodes along the way. To gain a more stable view of the tree and stronger invariants, the favored locking protocol is *lock coupling* [1], [4]. In this protocol, to move from an already locked node *A* to a child node *B*, one first locks *B* and only then releases the lock on *A*. This has the effect of preserving operation order along any given path in the tree and is deadlock-free because it always locks parent before child. However, one cannot implement lock coupling using atomic blocks, because the periods of time *A* and *B* are locked are neither independent nor properly nested. In our protocol, we release the lock on *A* before acquiring the one on *B*, which substantially relaxes what we can assume about the state of the world when we arrive at *B*.

We use atomic blocks because of their software engineering advantage of clearly delimiting the scope of protected access to a node. They match well with the synchronized blocks of Java [5], for example, though we support both shared (*S*) and exclusive (*X*) modes of access, for additional concurrency. While our atomic blocks designate a particular object (generally a B-tree node) to lock, they are quite similar to atomic blocks used for transactions [6] supported by software or hardware transactional memory (TM) [7], [8]. (In future work we hope to compare the present design to one using TM.) Atomic blocks make it somewhat easier to check that our code is deadlock-free. In particular, whenever we nest the blocks, we lock only nodes at the same level of the tree, and only in left-to-right order.

A concern that all highly concurrent implementations face is how long a given node may be locked. Typical descriptions of B-tree algorithms use an elegant recursive style, not only for searching but also for handling node splits and deletions, etc. In this style, if a node might possibly split (or become under-full, etc.), one must lock it for the duration of operating on the subtree under the node. This can seriously restrict parallelism. Hence, following the lead of others [9], [10], we *defer* structure modifying operations (SMOs). For concurrency and to avoid deadlock, we run them as *separate operations*, occurring after the primary operation that the user requested. We can perform SMOs *synchronously*, i.e., by the user thread at the end of the user operation (before returning to the application), or *asynchronously*, by some extra worker thread at an arbitrary later time.

As is the case when searching without lock coupling, here the world also can have changed significantly between the time we request an SMO and the time it occurs. For example, if a node splits, the proper parent-level node into which to insert the new node may not be the parent that we encountered on our way down the tree. Also, since the new node may exist for some time at its level before it will appear at the parent level, searching and other operations must work without exact information from the parent level. We have found it helpful to think of the interior of the tree as a helpful cache to get you *near* the desired leaf, rather than a definitive index (though in an idle tree it will be definitive).

We further recognize that user code often requests a series of accesses that have locality in the B-tree, notably sequential

scans. Therefore we support *cursors*, which cache information about nodes encountered on a previous search and reduce the need to search from the root each time. The general pattern of access via a cursor involves accessing a leaf node, then nodes at succeedingly higher levels until finding one whose range includes the search key, and proceeding back down the tree from there. This could lead to deadlock in classical implementations, but because we access each level separately, there is no problem.

We summarize our contributions as follows: a B-tree design that (1) offers a very high degree of concurrency; (2) controls concurrency with atomic blocks; (3) supports various granularities of locking; (4) supports cursors; and (5) allows asynchronous processing of deferred structure updates. We next describe related work and some preliminaries, and then present our design, followed by some correctness arguments. We describe our prototype implementation, then conclude.

II. Related Work

Over the past 30 years concurrency control of B-trees has been studied extensively, and the literature contains a wide variety of concurrent algorithms for them [2]. We broadly classify concurrent B-tree algorithms by the underlying locking schemes and structural enhancements to the basic B-tree data structure [11]. One can characterize the locking schemes by lock access type (shared, exclusive, and their intentional versions), duration (locks or latches), direction (top-down vs. bottom-up), scope (hierarchical vs. single node), and policy (pessimistic, optimistic, and two-phase locking [12]).

Bayer and Schkolnick [13] discussed the basic class of lock-based concurrent B-tree algorithms. The first improvement over locking the entire tree for every operation was to lock tree nodes via top-down pessimistic *lock-coupling*, i.e., lock a child of a node before releasing the lock on its parent. The optimistic descent algorithm improved on the basic design by allowing insert and delete operations to use *S* locks on internal nodes. If the leaf operation leads to restructuring, the process restarts and locks all participating nodes in *X* mode. Other generalizations of the optimistic descent algorithm include optimistic descent with variable critical level and with intention-exclusive locks. The ARIES family of concurrency and recovery algorithms [14] uses a variation of the optimistic descent algorithm with shared and exclusive modes, and their intention versions. The Mond-Raz algorithm [15] uses *X* locks via lock-coupling for insert and delete operations. However, the algorithm never holds more than two locks at a time, which requires top-down restructuring of the tree. Graefe [16] proposed improvements to the traditional B-tree locking schemes via hierarchical key-range locks and locks on separator keys. Graefe's proposal also enables automatic derivation of lock modes for key-range locks and dynamic locking hierarchies for separator locks.

Most structurally-enhanced concurrent algorithms use the B^{link}-tree structure proposed by Lehman and Yao [17]. The B^{link}-tree enhances the basic data structure by allowing each node (other than the right-most node of a level) to refer to its right neighbor. All operations place shared *S* locks from the root to the parent of the leaf nodes. Search operations lock the leaf in *S* mode, while insert and delete lock the leaf in *X* mode. If the leaf is going to split, the algorithm creates a new sibling leaf, connects it in the linked list, and moves half the keys from the original leaf to its new sibling. It then releases the *X* lock on the leaf and upgrades the parent lock to *X* mode. After that it updates the parent to include the pointer to the newly inserted child. If the parent is full, the process continues. All operations lock at most one node at a time in *X* mode since *physical* updates to a node may occur in any order [9]. Sagiv [9] described a B^{link}-tree based algorithm that supports concurrent compression. However, in the presence of this concurrent compression, the algorithm associates a *set* of locks with each node, and even a search must lock at least two nodes via lock coupling. The alternative solution requires restarting some processes. Analytical and experimental performance evaluations of concurrent B-tree algorithms have demonstrated that implementations that use B^{link}-trees exhibit higher concurrency [18], [19].

Sagiv's enhancements to the B^{link}-tree enable implementation of search and update operations using a sequence of atomic actions [9]. Eswaren et al. [20] introduced the notion of atomic action in the context of database operations, which Lomet expanded [21]. Lomet and Salzberg later used atomic actions to improve concurrency and recovery of index tree structures [22]. Recently, Lomet [10] proposed using atomic actions to implement deferred execution of structured modifying operations caused by deleting B^{link}-tree nodes. Our algorithm builds on these ideas. In particular, we use data structures similar to Lomet's *todo* queues to order deferred SMOs. We differ from prior work in: using atomic actions for *all* operations; giving details for ordering SMOs; supporting cursors; and offering several lock granularities.

III. Preliminaries

It is convenient to offer a brief review of B-trees, for clarity of concepts and terminology. We also describe the assumptions we use concerning atomicity of accesses to B-tree nodes, etc.

A. B-Trees

B-trees are a familiar structure, but possess several variants. The particular organization we use has two basic features: (a) all user key-value pairs are stored in the leaves, with keys and values in interior nodes serving only an indexing function (this is called a B^+-tree), and (b) every node has a pointer to its right sibling (this is called a B^{link} tree). Property (a) facilitates insertion and deletion in that one is not presented with the case of deleting a separator key in an interior node, etc. Property (b) helps with sequential access in the leaves; we apply it to good effect at all levels, to assist navigation in the face of asynchronous updates to different levels of the tree.

We consider a B-tree used as a general index, which may permit storing multiple values associated with the same key. (One can easily restrict any given tree to permit only single values.) This leads to the following set of basic operations:

Create(), which creates a new, empty B-tree

Fetch(key, value), which searches for the key-value pair; it returns the pair if it is present, or *null* otherwise; *value* may be $-\infty$, which requests the lowest pair for that key, or $+\infty$, which requests the highest; *key* may be $-\infty$, which requests the lowest key in the tree, or $+\infty$, which requests the highest;

FetchNext(key, value), which finds the next higher key-value pair and returns it, or *null* if there isn't a higher pair; *value* may be $-\infty$ to find the lowest pair with a key at least as high as the argument key, and may be $+\infty$ to find the lowest pair for the next higher key;

Insert(key, value), which inserts the key-value pair; and

Delete(key, value), which deletes the key-value pair; *value* may be $-\infty$ to delete the lowest pair for the given key

Obviously one can develop a number of minor variations concerning what happens if one attempts to insert a key that is present, to delete a key that is absent, etc. One can also implement reverse scanning (*FetchPrevious*), though it is not strictly symmetrical to implement because we link nodes in only one direction. Linking in one direction reduces the number of nodes affected by structural changes (node insertion and deletion), so is generally preferable to bi-directional linking.

In our scheme each B-tree node includes the following data:

n – the number of key-value pairs in the node

$pair_i$ – the ith key-value pair, for $1 \le i \le n$; the pairs are in increasing lexicographic order; note that since we support multiple values associated with the same key, we need both keys and values in interior nodes, so that we can split long runs of values associated with the same key.

min, *max* – pairs that define the range of pairs allowed in the node; for each pair p_i, $min \le p_i < max$. Also, a node's *max* equals its right sibling's *min*. The *min* of the leftmost node of a level is $\langle -\infty, -\infty \rangle$; the *max* of the rightmost node is $\langle +\infty, +\infty \rangle$. Neither *min* nor *max* need be a pair present in the tree. We interpret $pair_0$ as meaning *min* and $pair_{n+1}$ as *max*.

$child_i$ – present in interior nodes only, the ith child of this node, for $0 \le i \le n$; normally every pair p in the child's subtree obeys $pair_i \le p < pair_{i+1}$; we later relax that property to allow deferred updates of interior nodes.

next – pointer to the right sibling; *null* if at right end

level – the level of the node, 0 for leaves, 1 for their parents, etc.; while not strictly necessary, this field is convenient

state – the state of node; any of several values (*beingAdded*, *present*, *beingDeleted*, *deleted*, *unlinked*, and *derooted*), which indicate the stage of adding or deleting node

Later we will add more fields related to handling concurrency and supporting cursors. Notice that there are no back pointers from child nodes to their parents. Figure 1 illustrates the primary fields of a node.

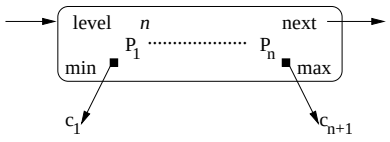

Fig. 1. A B^{link} tree node

The B-tree itself consists only of a pointer cell referring to the root node of the tree. We update this cell only when the tree changes in height. For a tree of order k, normally each node except for the root will contain at least $\lceil k/2 \rceil$ pairs, and can never contain more than k pairs. The root node can contain as few as 0 pairs (for an empty tree).

B. Atomic Access

We desire to specify atomic access according to *regions of code*, similar to synchronized methods or synchronized blocks in Java, or atomic blocks as proposed elsewhere [21], [23], [8]. Our intention is to describe our algorithm's use of these regions so that one can easily generate an implementation based either on locks or, in the future, on transactional memory. To facilitate this, a region specifies an object and a locking mode. The modes are S (shared) and X (exclusive).

Locking semantics: When entering an atomic region, one *acquires* the designated object in the designated mode. Only one thread at a time may acquire a given object in X mode; multiple threads may acquire an object in S mode, but not at the same time as any thread in X mode. When a thread has acquired an object in X mode, it may read and write fields of the object; in S mode it is limited to reading fields of the object. To avoid deadlock, we do not allow a thread to *upgrade* its lock on a given object by nesting an X mode region within an S mode region for that object. One may nest an S region in an X region, but it does not downgrade the lock. Exiting a region (by completing execution, returning from within it, throwing an exception, etc.) *releases* the corresponding lock on the acquired object, reverting the acquisition state to what it was before entering the region (which may be no different). For a given nest of regions, all writes must occur logically after the first acquisition in X mode and before the last X mode release, and all reads must occur logically after the first acquisition (in any mode) and before the last release. Further, execution must be consistent with a single total order of execution of regions, and with the order of execution of regions by each thread.

Our coarse- and medium-grained schemes lock *one* object to gain S or X mode access to any or all of a clearly specified *set* of objects. In particular they lock the B-tree object (not the root node, but the cell referring to it, since the root node may change) to gain S or X mode access to levels of the tree above a statically determined threshold. One uses fine-grained access for levels below the threshold.

IV. The B-Tree Design

Section III-A introduced the operations and basic structure of our B-tree design. We now begin to address making them highly concurrent. As taught in a data structures course or presented in a textbook, B-tree operations are usually implemented so as to recurse down the tree. This is simple, natural, and elegant, but becomes problematic in the face of concurrency. Either you effectively lock all nodes from the root to the leaf, sometimes drastically reducing concurrency, or you end up with a very complex recursion that somehow

resynchronizes if things change under its feet. One property of the textbook approach is that it wraps all the changes triggered by a given operation (e.g., all the possible splits caused by an insertion into a full leaf) into one large operation. This operation is supposedly atomic, maintaining strong structural invariants on the tree. Following others [22], we break the large operation down into smaller steps, weakening the structural invariants of the tree in order to improve concurrency.

Invariants still maintained: We maintain a strong invariant for each level of the tree. The nodes of a level exactly partition the key-value space from $\langle -\infty, -\infty \rangle$ to $\langle +\infty, +\infty \rangle$, with the *max* of each node equaling the *min* of its right neighbor. The pairs of each node lie within the node's range and are stored in order. We always maintain the *next* links. Any update to a node preserves these invariants.

Invariants not maintained: We relax those invariants that *connect* levels. In particular, a child pointer may refer to a node whose *min* is lower than what the parent has recorded as the child's *min* (and likewise for *max*). Thus, search must sometimes proceed to the *right* at the same level, as opposed to *down* towards the leaves. This relaxed parent-child invariant matches with our uni-directional links at each level, which allow immediate access from each node to the same-level node with the next higher set of keys/values. Figure 2 shows one example situation: here, searching for key 7 will proceed *down* from node **P** to node **A**, and then *right* from **A** to **B**. The search is perceiving the state between the split of **A** at the leaf level and the insertion of **B** at the parent level.

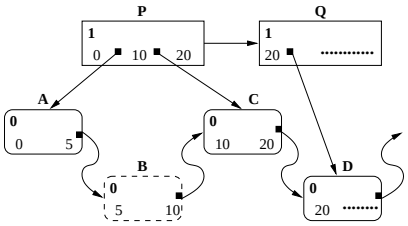

Fig. 2. A B^{link} tree with a split in progress

New invariants: A new invariant we introduce is that once a node becomes empty, it remains empty. (This does not apply to a leaf node whose range includes all keys, i.e., the node representing a one-level tree.) The purpose of this invariant will become clear when we discuss our deletion procedure.

A. Procedure for Searching from the Root

We now offer pseudo-code for searching for a given key-value pair at a given level of the tree. The special keys/values $-\infty$ and $+\infty$ are readily handled by the within-node search procedures, which we need not detail here.

```
Node Search (int level, Pair pr): // assumes level >= 0
  Node curr;
  Node next = root;
  Node result = null;
  while (result == null)
    curr = next;
    atomic (curr, S)
      if (pr does not lie in range of curr)
        // by invariant, must be to the right
        next = curr.next;
      else if (curr.level > level+1)
        // need to descend to next level
        next = child whose range we think contains pr;
      else if (curr.level == level+1)
        // we are one level above desired level
        result = child whose range we think contains pr;
      else
        // we are the appropriate node (short tree)
        result = curr;
  return result;
```

B. Procedure for Applying an Operation

The *Search* procedure essentially uses higher levels of a tree to find, relatively quickly, a good starting point for obtaining the node on which one desires to operate. Because of asynchrony, the desired node may actually be to the right of the node returned by *Search*, so a fundamental operation at a given level proceeds using this pattern:

```
Result ApplyOp (Node start, Pair pr, Mode m):
  Node curr;
  Node next = start;
  Result res;
  while (next != null)
    curr = next;
    atomic (curr, m) // m is the locking mode Op needs
      if (pr lies in range of curr)
        res = perform Op;
        next = null;
      else
        next = curr.next;
  return res;
```

If we abandon the discipline of strictly matching acquire/release pairs, we could write a simpler searching routine that would find the desired node, lock it in the requested mode, and return it. Another way to avoid writing the pattern repeatedly (once for each operation) is to write a single search routine that takes a handle on an operation (e.g., a function pointer in C) and its arguments, i.e., essentially a closure. The routine does the search and applies the operation, so the *ApplyOp* pattern appears only once. Notice that in between executions of the atomic block, node ranges can change. However, the proper node always lies to the right.

C. Complexity of Searching

At this point a few remarks on the algorithmic complexity of our search procedure may be helpful. It should be clear that if the B-tree is balanced (as it should be by definition), and its levels are up to date with respect to each other, then a search from the root visits $O(\log n)$ nodes for a tree containing n key-value pairs. If there are k deferred structure-modifying operations (SMOs), then a parent node may omit up to k

child pointers. Thus a traversing thread may need to make up to k moves to the *right* in the tree, without moving *down*. To maintain a bound of $O(\log n)$ time for traversing a tree, we need to bound k so that it also is $O(\log n)$. This requires semaphore-style synchronization for performing SMOs, i.e., a semaphore that, when idle, has a value N that is $O(\log n)$, and on which a thread must perform a P before requesting an SMO, and a V after updating the parent level.

We did not feel it was worth putting this extra logic into our implementation, for the following reason. In a system with high concurrency, any given thread can be continually overtaken as it traverses toward a target leaf node. The overtaking threads can perform inserts in leaves and force continual splits. The semaphore synchronization we just described controls only the number of *simultaneously outstanding* SMOs, not the *total number* of SMOs that can occur in between times that our unlucky thread makes progress. This issue exists in lock coupling implementations as well, if they support overtaking: the parent has out-of-date boundary key information, so the unlucky thread goes to the "wrong" child. It may need to traverse to the right. But other threads can insert rapidly and push the unlucky thread's target key farther and farther right.

The only solution is to impose a fairly strong global fairness-of-progress guarantee, namely that operations started against the tree earlier will eventually beat all new operations in making progress. One mechanism that would work is to distribute sequentially numbered tickets as operations begin, and to hold back new operations if the oldest incomplete operation is more than $O(\log n)$ tickets ago. There are undoubtedly more clever ways to do this that reduce the frequency of synchronization. We chose not to pursue a comprehensive solution, but we do insure that the degree of concurrency in our experiments does not exceed the number of hardware threads available, thus reducing the likelihood of long periods of thread inactivity because of descheduling, which would make threads vulnerable to this kind of starvation.

D. Fetch and FetchNext

Implementing *Fetch* is straightforward given *Search* and *ApplyOp*: either the desired pair is present or it is not, and we return the appropriate result, as shown in *FetchCore* below (which would be invoked where *Op* is called in *ApplyOp*).

```
Pair FetchCore (Node curr, Pair pr):
  // by design pr is in curr's range
  // and curr is acquired in S mode
  if (pr in curr)
    return pr;
  else
    return null;
```

FetchNext is more complex, and requires a custom version of *ApplyOp*, shown as *FetchNextOp* below. The custom version is required because the input pair for *FetchNext* may be the last pair in a node (or beyond it), forcing *FetchNext* to examine nodes to the right. In this case *FetchNextOp* must lock both nodes to insure that no pair is inserted between *pr* and *res*. It must skip any intervening empty nodes (we guarantee that such nodes will remain empty).

```
Pair FetchNextOp (Node start, Pair pr):
  Node curr;
  Node next = start;
  Pair res = null;
  while (next != null)
    curr = next;
    atomic (curr, S)
      next = curr.next;
      if (pr is not in range of curr)
        continue;
      else if (curr has a pair > pr)
        res = the smallest such pair;
        next = null;
      else // find next non−empty node
        Node succ;
        while (next != null)
          succ = next;
          atomic (succ, S)
            next = succ.next;
            if (succ has pairs)
              res = smallest pair of succ;
              next = null;
  return res;
```

Fetching the first and last pairs of the entire tree are slightly special cases, but offer no difficulty.

E. Insert: Handling Splits

Except for the case of a node that is full, *Insert* is also straightforward. However, it is important to obey the range of pairs allowed in a node. If the pair being inserted comes after all pairs currently in the node and the node has room, one might think it is all right to insert the pair in the node. But if the pair does not lie in the node's assigned range, we *must* proceed to the right. This situation would not arise in a non-concurrent B-tree, or in one that maintained strict consistency of boundary key information across levels. But using deferred SMOs implies that some insertions may arrive at the "wrong" node, to the left of where they should be. If we fail to obey the nodes' assigned range information, we can end up inserting pairs out of order. In fact, it was through considering cases such as these that we determined the wisdom and clarity of recording the ranges in the nodes, as opposed to keeping only separator keys as a non-concurrent B-tree would.

What if the proper node to receive the new pair is full? As expected, we split the node, inserting a new right sibling that receives the higher half of the pairs. We determine a boundary pair value b that separates the two groups of pairs, and set the left (original) node's range to end at b, and the right node's range to start at b and end where the original node previously ended.

```
boolean InsertCore (Node curr, Pair pr):
  // by design pr is in curr's range,
  // and curr is acquired in X mode
```

```
if (pr in curr) // no change required
  return false;
else if (pr fits in curr)
  insert pr;
  return true;
// overflow: rebalance or split
// this is the place to attempt rebalancing if desired
Node fresh = a new node; fresh.state = beingAdded;
balance old pairs plus pr between curr and fresh,
  with fresh getting the higher pairs;
// insert fresh on same−level linked list of nodes
fresh.next = curr.next; curr.next = fresh;
// adjust node ranges
fresh.max = curr.max;
Pair bnd = smallest pair of fresh;
curr.max = fresh.min = bnd;
// request a deferred SMO
request InsertSMO(curr.level+1, curr.min, curr, bnd, fresh);
return true;
```

At this point, the insertion is complete at the leaf level. We then perform, as a *separate operation* on the tree, an SMO to insert the boundary pair *bnd* and the new node into the parent level. Note that the node at the parent level that we traversed to get to the leaf that we split may not have been, or may no longer be, the parent of the split node. That is why we speak of inserting the pair and node into the parent *level*. That insertion proceeds analogously to inserting a pair into a leaf. If the insertion at the parent level causes a split, we perform yet *another* separate SMO to insert the new parent-level node into the grandparent level, etc. SMOs do have ordering requirements, which we discuss further below (Section IV-H).

Figure 3 illustrates the overall sequence of handling a split. (A) shows the initial situation; (B) shows the state after splitting node **A** into **A** and **C** (but before running *InsertSMO*), and (C) shows the final state after running *InsertSMO*. As discussed with Figure 2, the situation in (B) is visible to concurrent B-tree operations.

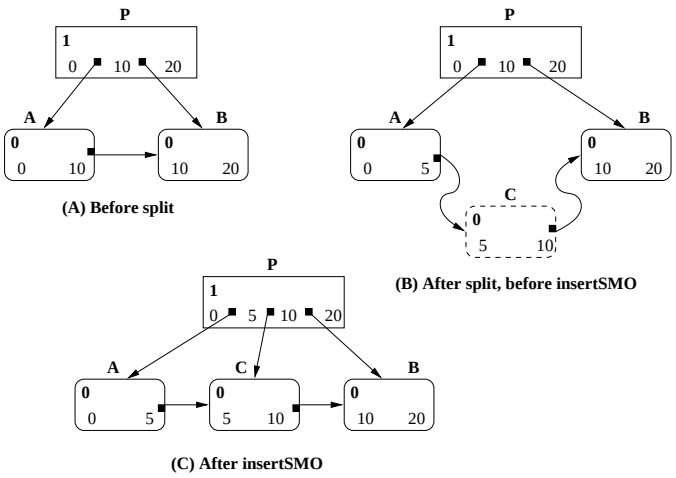

Fig. 3. The Insertion Operation

If the root node splits, we create a new root node, referring to the split node and its right sibling, and update the root pointer to refer to the new root node. Note that operations that come to the split root node before we add the new root node still proceed correctly, though they may have to take an immediate move to the right.

F. Insert: Rebalancing

An option that sometimes avoids allocating a new node when inserting into a full node is to perform local *rebalancing*. Recall our rule that pairs (and the key range associated with a node) can move only to the right. If an inserted pair overflows its target node, and the target's right sibling has free space, we can move some pairs from the target to the sibling. This requires locking both nodes exclusively, of course. Note that not only do some *pairs* move, but also some range of *key space* gets moved from the target to the right sibling as well. This requires a separate SMO to the parent level, to record that adjustment of key space (updating the boundary between the siblings). This is similar to inserting into the parent level after a split, except that it modifies existing information as opposed to adding a new pair and child pointer. It was not obvious to us that the greater space efficiency sometimes possible with rebalancing instead of splitting was worth the extra implementation effort (in either case, space efficiency remains $O(n)$), so we omitted this rebalancing capability in our prototype, and likewise omit design details here.

G. Delete: Handling Underflow

As with insertion, deletion most commonly involves a search followed by updating one leaf node. For leaves that become underfull, there are two strategies employed in the past. One strategy rebalances the under-populated leaf by drawing pairs from one (or both) of the leaf's siblings. This applies in a B-tree of order k if there are still $2 \times \lceil k/2 \rceil$ pairs left between two adjacent nodes, i.e., enough to properly populate both nodes. If there are not enough, then one shifts all the pairs into one of the nodes and deletes the other one. This strategy maintains $O(n)$ space use for the B-tree, necessary for obtaining $O(\log n)$ levels and thus $O(\log n)$ time for operations.

An alternative strategy, employed in a number of commercial systems, presumably because it is simpler and usually works adequately well in practice, is to tolerate underfull nodes, deleting them only when they become entirely empty. Our prototype takes this approach, to avoid the complex algorithmics of rebalancing. Note that one can also simply rebuild a tree if its space efficiency is too low, though doing so concurrently would require additional coding effort.

We offer pseudo-code for *DeleteOp* below. Its subtle aspect is the adjustment of the ranges of nodes when a node becomes empty and we wish to delete it. We desire to make the range of the empty node also empty (*max* equal to *min*), so that the node can be unlinked and freed later. To do this, we "push" the node's range to the right (following our rule that range and pairs move only to the right). There are two special cases of concern. One is if the empty node is the last one on its

level. In that case, we simply leave it. This can result in at most $O(\log n)$ space waste, not significant asymptotically (or in practice for trees of any size). The other interesting case is when the node's right sibling is also in the process of being deleted. In this case, we search for a node farther to the right, until we reach a node not being deleted. (Such a node must exist, since we do not delete the rightmost node.) It might seem that threads can race in deleting nodes and pushing range to the right, but since we retain an *X* mode lock on the node being deleted until we successfully push its range right, threads cannot pass each other in the pushing process. Rebalancing underfull nodes (if implemented) needs to proceed similarly, skipping over nodes being deleted to find a suitable node into which to shift pairs (and range). That search should be done within the first atomic block, as noted in the code.

```
boolean DeleteOp (Node start, Pair pr):
  Node here, first, second;
  Node next = start;
  boolean more = false;
  while (true) // loop to find pr's node
    here = next;
    atomic (here, X)
      if (pr not in range of here)
        next = here.next;
        continue;
      else if (pr not in here)
        // no change required;
        return false;
      delete pr;
      if (here has pairs but is underfull)
        // rebalancing goes here; should search to find
        // the first node to the right that is not being
        // deleted, then atomically shift pairs and range
        // to that node and start a rebalancing SMO at
        // the parent level
      if (here has any pairs || here.next == null)
        return true;
      // starting to delete node here
      here.state = beingDeleted;
      first = here;
      // repeatedly push key range to the right
      boolean more = true;
      while (more)
        atomic (first, X)
          first.max = here.min;
          second = first.next;
          atomic (second, X)
            second.min = here.min;
            more = (second.state == beingDeleted);
          first = second;
    break;
  request DeleteSMO(here.level+1, here.min, here, first);
  return true;
```

In the worst case, the code above acquires three nodes at once, the node being deleted (*here*) and a pair of adjacent nodes (*first* and *second*) between which we are shifting key range. (In the first iteration of the loop, *here* and *first* are the same node.) It may be possible to reduce this to two nodes, but it would require relaxing the invariant that key range is always exactly partitioned across a level.

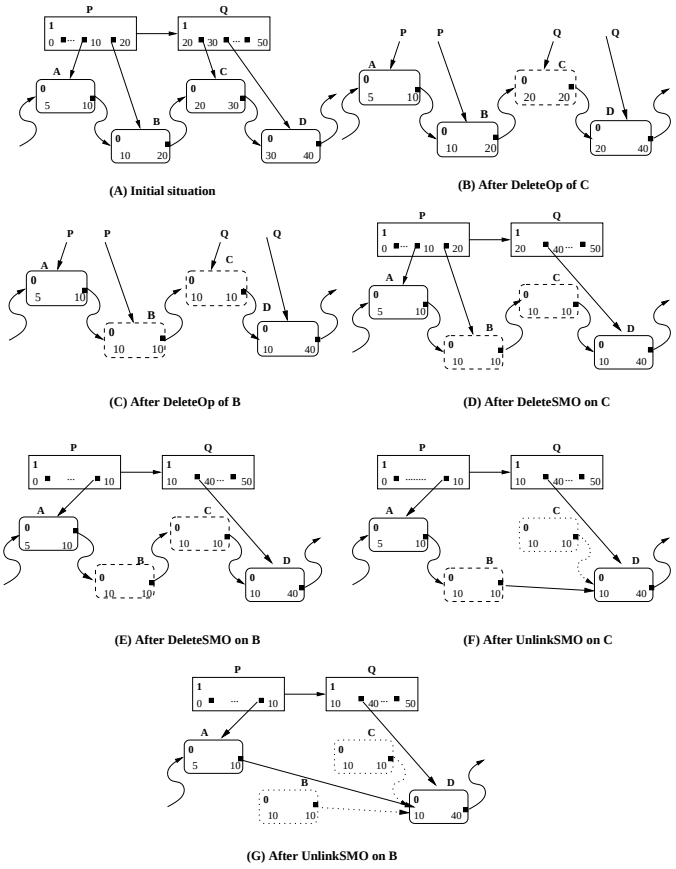

Fig. 4. The Deletion Operation

Figure 4 (A), (B), and (C) illustrate this "pushing" procedure. (A) shows an initial state with nodes **P** and **Q** at the parent level, and leaves **A**, **B**, **C**, and **D**. (B) shows the situation after *DeleteOp* makes **C** empty: **C**'s range has moved to **D**; **Q** will be updated by the deferred *DeleteSMO*. Similarly, (C) shows the situation after *DeleteOp* makes **B** empty. Notice how its range is "pushed" through **C** to **D**, but updating **C**'s (empty) range on the way. We discuss the remainder of the figure after presenting the relevant SMOs.

Concerning rebalancing, if the underfull node's right sibling is too full to receive all of the underfull node's pairs, one can imagine "pulling" pairs from the left sibling. This is complex because the linked list goes in one direction only. Double linking requires more locking and updates, so is not a great option. Notice that *together* an underfull node and a full right sibling still maintain $O(n)$ space usage all together, so in fact "pulling" is not necessary.

H. Deferred Structure-Modifying Operations

First we take up how each SMO is implemented, and then consider how we insure that we execute SMOs in a safe

order. In the pseudo-code above, there are occasions where we *request* an SMO. By that we mean that we either (a) make a note of the desired SMO and execute it at the end of the current operation, just before returning to the user, or (b) enqueue the SMO for some helper thread to execute on our behalf. Option (a) we call *synchronous* and (b) *asynchronous*. Both are *deferred*, meaning they are executed after the current operation and start in a situation where the thread is not in an atomic block. Asynchronous SMO execution requires designing a suitable work queuing mechanism, which we found to be tricky, but has the advantage that the number of helper threads constrains the number of concurrent SMOs, and in particular using a single helper thread guarantees atomicity of SMOs with each other, simplifying implementation. Asynchronous SMO execution reduces user-thread operation times (and variance in those times), while synchronous SMO execution may increase concurrency of SMO execution.

In any case, each SMO must search, to the appropriate level. We provide a target pair and also the relevant child node(s) and other information. The code below sketches *InsertSMOCore*, as would appear inside the *ApplyOp* pattern. However, before calling *Search*, the *InsertSMO* code needs to check if the level of the requested insert is higher than the root node of the tree (i.e., the root just split). In that case, it must create a new node with the two children and given boundary key, and update the root pointer to refer to that new node.

```
void InsertSMOCore (Node parent,
                    Pair start, Node child,
                    Pair split, Node fresh):
  // here we hold parent in X mode,
  // and start is in parent's range
  insert split and fresh just after start and child;
  fresh.state = present;
  if (parent overflows)
    do split analogously to leaf level;
    // the next level SMO request looks like this:
    request InsertSMO(parent.level+1, parent.min, parent,
                      newbnd, fresh);
```

In pseudo-code, *DeleteSMO* is quite similar to *Delete* (see below). However, after removing the child from the parent's level (which may request an SMO at the grandparent level, etc.), it requests an *UnlinkSMO*, to remove the *child* from the linked list at its level and attempt to reclaim it. If the *child* is the first node of its level, then we can skip unlinking. Otherwise, *UnlinkSMO* searches to find the predecessor of the child being removed. It first proceeds similarly to *Search* and *ApplyOp*, except that it is trying to find a node whose *min* is *less than* the child's *min* (*start* in the code below), and in its *ApplyOp* loop it is trying to find a node whose *next* is *child*. Call this node *pred*. The pseudo-code for *UnlinkCore* shows the rest. Note, however, that it is possible for *child* to end up as the first node of its level, even *after* we request the *UnlinkSMO*. In this case *pred* will be *null*.

```
void DeleteSMOCore (Node parent, Pair start,
                    Node child, Node rcvr):
  // here we hold parent in X mode and
  // start is in parent's range;
  // rcvr, which receives the deleted range,
  // is used for synchronization
  delete start and child from the node;
  child.state = deleted; // will no longer be offered
  if (rcvr is not a child of parent)
    push range from start to the right of parent
  if (parent is underfull or empty)
    proceed similarly as for leaf level
  // may request DeleteSMO to remove
  // parent at grandparent level
  request UnlinkSMO(child.level, start, child, rcvr);

void UnlinkCore (Node pred, Node child, Node rcvr):
  // if pred != null we hold it in
  // X mode, and child == pred.next
  if (pred != null) pred.next = rcvr;
  child.state = unlinked;
```

Returning to Figure 4, (D) shows the situation after the *DeleteSMO* for **C**. At that time, **Q** no longer points to **C**, but **C** is still linked at the leaf level. (E) shows the situation after the *DeleteSMO* for **B**. Similarly, **P** no longer points to **B**, but **B** is still linked. Finally (F) and (G) show the situations after the *UnlinkSMO* for **C** and **B**, respectively.

I. Ordering of Deferred SMOs

It is fairly easy to see that we need *some* ordering restrictions on executing SMOs. For example, if a node splits, and then all of the fresh node's pairs are deleted, we might have concurrent deferred SMOs for the split that introduces the new node and for the node's deletion. A drastic solution would be to execute SMOs one at a time in the order they were requested. However, this would be a concurrency bottleneck and is overly restrictive, since many SMOs can proceed at the same time safely. The important insight to solving this problem is that we need to execute two SMOs in the order in which they were requested only if their affected key ranges overlap: SMOs for non-overlapping ranges can proceed concurrently (modulo atomicity of updates to parent-level nodes).

At first, it may not be clear how we can enforce ordering for each bit of key range without introducing another complex concurrent data structure. What we do is that we use the child-level nodes as surrogates for their ranges. We say that an SMO *pertains to* one or more affected nodes if their range is involved. Thus an *InsertSMO* pertains to the node that was split (*child*) *and* to the node introduced by the split (*fresh*), and a *DeleteSMO* pertains to the deleted node (*child*) and the node to which it ultimately shifted its key range (*rcvr*). Likewise an *UnlinkSMO* pertains to *child* and *rcvr*.

When we *request* an SMO, we obtain a *ticket* for each node to which the SMO pertains. Here is how tickets work. Each node has two *ticket counters*, *nextTicket* and *nextServed*, each starting at 0. To get a ticket, while holding the node in *X* mode we read the value of *nextTicket*, and then increment it. An

SMO is *ready* if the *nextServed* values in each node to which the SMO pertains match the tickets that we obtained when we requested the SMO. We begin to execute an SMO only after it is ready. When an SMO completes, if increments *nextServed* in the nodes to which it pertains. Checking whether an SMO is ready should properly be done having acquired the relevant objects in at least *S* mode, though an implementation may be able to use some form of "volatile" memory read instead. Likewise, if an SMO does not need to acquire a pertinent object in *X* mode, when done it may be able to use an atomic increment instruction on *nextServed*, avoiding locking. These are just refinements to the obvious safe strategy of accessing and updating the ticket counters only under the proper lock.

Figure	Node B		Node C		Node D	
	tkt	srvd	tkt	srvd	tkt	srvd
(A)	3	3	6	6	10	10
request DeleteSMO(1, 20, C, D, 6, 10)						
(B)	3	3	7	6	11	10
request DeleteSMO(1, 10, B, D, 3, 11)						
(C)	4	3	7	6	12	10
run DeleteSMO(1, 20, C, D, 6, 10) request UnlinkSMO(0, 20, C, D, 7, 12)						
(D)	4	3	8	7	13	11
run DeleteSMO(1, 10, B, D, 3, 11) request UnlinkSMO(0, 10, B, D, 4, 13)						
(E)	5	4	8	7	14	12
run UnlinkSMO(0, 20, C, D, 7, 12)						
(F)	5	4	8	8	14	13
run UnlinkSMO(0, 10, B, D, 4, 13)						
(G)	5	5	8	8	14	14

Fig. 5. Ticket sequencing for Figure 4

Figure 4 implicitly shows our ordering constraints. We assumed that a *DeleteOp* call requested a *DeleteSMO* for **C**, and then another *DeleteOp* call requested a *DeleteSMO* for **B**. Ticket ordering on **D** forces the two *DeleteSMO*s to run in that order at the parent level. They take new tickets on **D** for the two *UnlinkSMO*s, so those also run in the order **C** then **B**. Figure 5 shows details, assuming some starting ticket numbers, and adding ticket numbers to SMO requests. We abbreviate *nextTicket* as tkt and *nextServed* as srvd, and show the ticket situation before/after each SMO-related operation. Running a *DeleteSMO* involves requesting an *UnlinkSMO* as part of it, so those two appear together.

J. Cursors

A *cursor* caches information about a node, and the path from the root to that node, in the hope of speeding up later operations, such as *FetchNext* during a sequential scan. If a sequence of operations has some degree of locality, then cursors will likely speed up the sequence, at the cost of additional bookkeeping. A cursor may be used for accessing leaf nodes, or for accessing interior nodes when performing SMOs or needing to search further.

A cursor contains, for each level of the tree, the node most recently traversed at that level. New cursors start in an uninitialized state. Using an uninitialized cursor requires traversing from the root of the tree, but will initialize the cursor for each level of tree accessed during the search. Considering our *Search* routine, the node that should be entered into a cursor when starting from the root is the node whose key range includes the key for which we are searching.

When using an *initialized* cursor to search for a given pair *p* at a given level (e.g., leaf level), one examines the node that the cursor records for that level. If the cursor has no node for that level, or if the pair *p* lies outside the range of the node, one examines the node remembered by the cursor for the next higher level. If one finds a node whose range contains *p*, one proceeds from that point as in *Search*. It is possible that none of the nodes includes *p* (the tree may have grown in height), in which case one treats the cursor as uninitialized and starts from the root. As one finds nodes that include *p*, one records them in the cursor for future use.

A *FetchNext* proceeds only slightly differently from ordinary search: it looks for a node that includes *p* but whose *max* is strictly $> p$. It may still have to move right from that node, but cannot distinguish that case without inspecting the pairs in the node. In such cases of sequential access requiring a move to the right, one typically moves just one node to the right, and obviously it is sometimes necessary.

Optionally one may cache additional information in cursors to speed operations more. For example, one may record the *min* and *max* of each node recorded in the cursor, to reduce probing nodes that are not likely to help. The utility of this depends on locking costs, etc. Another possible improvement is to record in the cursor with each node the pair most recently accessed at that node, and the index of that pair within the node's array of pairs. To determine whether the index is valid, we add a field to each node called *modCount*, which we increment whenever we change the set of pairs in a node. The cursor samples and saves *modCount* when saving the pair and index information. Thus, if the saved *modCount* matches, then we can use the index and avoid a binary search for the pair. In trees with large nodes under heavy sequential access loads this may be a good idea, but deserves some study as to the benefits realizable in practice. The primary benefit of cursors is likely to be reducing the *number* of nodes accessed, the *number* of binary searches performed, etc.

K. Reclaiming Deleted Nodes

Consider a node that becomes empty and is unlinked from its level's linked list. One might think that we could reclaim that node immediately. In fact, it is possible that there are active threads that will still traverse it, and cursors may also refer to it. We propose to reclaim nodes using *reference counting*. The obvious reference counting strategy is to maintain in each node the current number of references to the node, both from other nodes and from threads and cursors. However, that strategy requires incrementing and decrementing reference counts even for read-only search operations. So the actual reference counting scheme we propose is *deferred* reference counting [24]. In this scheme we maintain in the node a count only of the number of references *from other nodes*, a quantity that does not change frequently. A node is certainly not eligible

from reclamation until this reference count is 0. (Notice that unlinked nodes can refer to each other and to linked nodes, so it is not obvious when the count will become 0.) But we need to prevent reclaiming the node if there are thread or cursor references to it. We now present one way to do that.

For each thread, and for each level of a cursor, assign an id that is unique among all the ids currently assigned. It is perhaps easiest to imagine the id as a direct index into a single flat array, which we call the *dynamic reference table*. In practice, one needs a scheme that is relatively fast at assigning currently unused ids and at getting and setting the node associated with an id.

Whenever a thread holds a reference to a node that it has *not currently acquired* (in either *S* or *X* mode), the thread stores a copy of the node reference into the thread's unique slot in the dynamic reference table. Likewise, when the thread is done using that reference, it clears the slot. Similarly, when caching a node reference in a cursor, one stores a copy of the node reference into the unique table slot assigned to that level of that cursor. We clear slots associated with cursors only when reinitializing or destroying the cursor. If it were important, it *might* be possible for an asynchronous thread to clear information in a cursor, but we view cursors as private to threads and thus not requiring synchronization for them to access. Allowing other threads to access cursors would impose additional synchronization requirements, which seems unwise.

When a node's reference count becomes 0, we scan the dynamic reference table. (This scan may be done only periodically by a background thread, if desired.) If no slots refer to the node, we can reclaim it. This works because once we unlink the node, no additional threads or cursors can obtain references to the node. If some table slot refers to the node, we cannot reclaim it. We manage the deferred reclamation of these nodes by entering them into a *watched node table*, along with a record of the table slot(s) that refer to the node. Periodically we check to see if any watched node's slots no longer refer to the node. Eventually there will be no such slots and we reclaim the node. We additionally need some means for indicating that a thread or cursor will no longer use its dynamic reference table slots, so that the table space can be reused or compacted. We handle that by requiring threads to *connect* to a tree (in order to obtain a table slot) and later to *disconnect* from it (freeing the slot). Since threads do not retain node references *between* B-tree operations, the connect/disconnect protocol can be associated with accessing *any* B-tree, not with each one individually. Likewise, we require cursor finalization, which will release the associated table slots. Our approach is essentially a specific application of Michael's *hazard pointers* technique [25].

L. Lock Granularity

In our design as presented we assumed atomic blocks (locks) that protect one B-tree node at a time. We call this *fine-grained* locking. For those actions that require atomic access to two or three nodes at once, we employ nested atomic blocks. In principle, the fine-grained approach will yield the highest concurrency (among approaches that lock only whole nodes, as opposed to fields of nodes, etc.). However, the highest *concurrency* may not give the highest *throughput*, because locking overhead can be significant. In fact, there are reasons to believe that in multi-core systems locking overhead will be relatively higher than in previous systems.

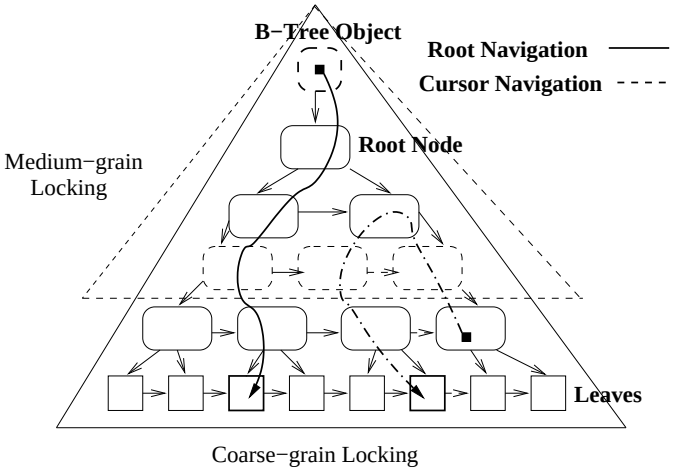

Fig. 6. Granularities of Atomic Blocks

We observe that in B-trees SMOs occur only a fraction of the time that leaf modifications do, and become progressively rarer at higher levels of the tree. Hence we propose to use fine-grained locking only for levels at or below a chosen *threshold level* of a given tree. Our implementation currently offers only these choices for fine-grained locking: none (one lock for the whole tree), leaves only, or all nodes fine-grained.

V. Informal Correctness Claims

We first consider the issue of which nodes constitute a particular level of the tree. For each level, there is a *first* node at that level. Consider the nodes at level *n* referred to by level $n+1$ (their parents), and all the nodes reachable from these level *n* nodes by following *next* pointers. We first claim that these are exactly the nodes of level *n* whose state is not *unlinked* or *unrooted*; we call these the *linked nodes*. It is easy to verify that nodes in states *present* and *beingDeleted* have a parent referring to them. Nodes in state *beingAdded* do not have a referring parent, but a referring left sibling that is reachable from the parent level, and will remain so (because of SMO ordering) at least until the new node is added at the parent level. Nodes in state *deleted* likewise have a reachable left sibling (because they were not the first child referred to from the parent level). Finally, *unlinked* nodes are not reachable from the parent level or from any linked node. The *unrooted* state is used for an old root node when the tree shrinks in height. It is obviously not reachable from the root, but continues to refer to the first node of the next lower level. Unlinked nodes continue to refer to their former sibling.

Second, we claim that the nodes of a level exactly partition the key-value pair space from $\langle -\infty, -\infty \rangle$ to $\langle +\infty, +\infty \rangle$. The initial node starts that way, and each split maintains the

property. When deleting a node, we make its range empty (but in sequence), "pushing" its range to the right. Thus, when we unlink a node in the middle of a level, we maintain the invariant. Likewise, if we delete a node at the left end of a level, since its range is empty and starts with $\langle -\infty, -\infty \rangle$, its right sibling also starts with $\langle -\infty, -\infty \rangle$.

Third, we claim that the min and max of a node never increase. A split decreases the max of the splitting node. A delete also decreases the max of the node being deleted, and the "pushing" right of its range decreases its sibling's min. Further pushing decreases one node's max and the next node's min. No node will ever have max $<$ min; the leftmost node's min will always be $\langle -\infty, -\infty \rangle$ and the rightmost node's max will always be $\langle +\infty, +\infty \rangle$ (it never decreases).

Fourth, because of the non-increasing min and max of nodes, "out of date" references from parents or non-linked nodes can refer only to nodes whose range begins no higher than what the referring node might assume. Thus, when search proceeds to level *n* searching for a particular pair, it will always arrive at or to the left of the proper node. If it is to the left, it can search to the right. It need not lock nodes (in lock coupling style) while doing this, since the referent node's range also obeys the non-increasing property.

Fifth, we make the following claims concerning SMOs and their ordering. (1) The left sibling *O* of a new node *N* must be inserted in the parent level before *N* is. This is because the ticket from *O* for adding *O* is strictly less than the ticket from *O* for adding *N*. (2) The right sibling *N* must be inserted in the parent level before *O* is deleted. Again, this is because the ticket from *O* for adding *N* must be strictly less than the ticket from *O* for deleting *O*. (3) A being-deleted node *D* must be deleted before the node *R* receiving its range is deleted. This is because the ticket from *R* for deleting *O* will be strictly less than the ticket from *R* for deleting *R*. (4) When a node *U* becomes unlinked, its range receiving node *R* must be linked. This is because when the *UnlinkSMO* is requested, *R* is not yet deleted (by (3)), so the ticket from *R* for unlinking *U* must be less than the ticket from *R* for unlinking *R*. (5) When *U* becomes unlinked, its range receiving node *R* will be its right sibling. *U* is deleted after all the nodes between *U* and *R*, and thus it will be unlinked after those nodes are unlinked. Since they're unlinked, there is no node between *U* and *R*, hence *R* is *U*'s right sibling. (6) If in *UnlinkSMO* the predecessor *pred* is not null, then *pred* is linked and is the left sibling of *child*. The fact that *pred* is linked follows from the fact that a search found it (search can find only linked nodes). It is the left sibling because it refers to *child*. These facts do not come from ticket ordering of *UnlinkSMO* but from the particular search it does. If *child* is first on its level, or becomes so during the search, *pred* will be null.

Why search works: At a given level when searching for pair *p*, in a parent node whose range includes *p* we will choose the child *c* that apparently contains *p*. When we arrive at *c*, it may have a different range from what we saw in the parent, but it can only be lower. If *c*'s *max* is $\leq p$, we proceed to the right. In between releasing *c* (in *S* or *X* mode) and acquiring its right sibling *r*, *r*'s range could become lower, but again, we will simply keep searching. It is not possible to prove termination of search, but it is not hard to see that if it terminates it gives the correct answer, and that termination is a problem only if other threads insert new pairs more rapidly than the searcher can skip over them. It is also possible that *c* is unlinked by the time we examine it. However, it maintains a *next* pointer to a node whose *min* is $\leq$ the *min* of *c* (which equals the *max* of *c*). Again, we can find *p* by proceeding to the right, even though *c* was unlinked. The same holds going to the only child of a unrooted former root node.

Why insert works: The non-split case follows from correctness of search. The split case maintains the invariants that allow search to work. In a sense the *InsertSMO* is not necessary for correctness, but it *is* necessary for the tree to stabilize to $O(\log n)$ cost. The SMO ordering constraints insure that operations pertaining to ranges containing any specific key *k* (splitting, balancing, deleting (merge of ranges), etc.) execute in the same order at the parent level as they do at the child level. Therefore, since *k* is always in the range of some node at the child level, not only will it be in the range of some node at parent level, in an idle tree the parent level will indicate the exact child whose range contains *k*.

Why delete works: The non-SMO case again follows from correctness of search. Otherwise, let *D* be the node to be deleted and *R* the node receiving its key range. The *X* mode lock we keep on *D* prevents any pushing of range from the left of *D*. The nodes between *D* and *R*, which are being deleted, have already pushed their range right (ultimately to *R*). They will be deleted from the parent level by the time this *DeleteSMO* executes, and they will be unlinked from the child level by the time the *UnlinkSMO* for *D* runs. So when the SMOs run, *R* is *D*'s right sibling and is still linked (even if *D* is the first node of its level, etc.).

VI. Implementation Details

We have implemented the algorithm in C using pthreads as the threading library. We tested the algorithm on Linux/x86 and AIX/PowerPC machines. The implementation has three different components: a multi-threaded workload generator that emulates a concurrent user workload consisting of concurrent insert, delete, and search operations; a workload manager that converts user requests into tree operations and also manages auxiliary functions such as cursors and asynchronous SMO threads (when supported); and the core B^{link}-tree component that implements a non-unique key-value pair index.

Upon the first request from any thread, the workload manager registers the thread and assigns it a cursor. The cursor data structure is a stack whose size is bound by the maximum depth of the tree. Initially it contains only one entry, which refers to the B^{link}-tree root. The workload manager then invokes the corresponding B^{link}-tree operation with the cursor top as the starting point of the tree operation. We have also implemented a non-cursor based implementation where each tree operation always starts from the tree root.

For the most part, our prototype implementation follows the design laid out in Section IV. The significant differences or choices we made are as follows. We do not support rebalancing, and thus we delete a node only when it becomes empty. While this can theoretically lead to bad space utilization, it is simpler and has a venerable history in practice. We implemented both synchronous and asynchronous SMO processing; for asynchronous processing we use a single SMO processing thread, which simplified the queuing protocol. We support atomic blocks using CAS-based multiple-reader, single-writer read-write locks. A writer waits for all concurrent readers or any current writer to complete. A new reader waits only if there is a current writer. This policy may starve writers, so we may need to revise it in the future. We implemented the read-write locks using hardware primitives on both architectures.

Our implementation supports three lock granularities: coarse, medium, and fine. In the coarse-grained implementation, an operation locks the entire tree; search operations lock it in *S* (read) mode, whereas insert, delete, and SMO operations lock it in *X* (write) mode. In the medium-grained implementation, we lock the internal (non-leaf) sub-tree separately from the leaves (see Figure 6). While executing search, insert, and delete operations, the navigation phase holds the lock on this region in *S* mode, while SMO operations lock it in *X* mode. We lock leaf nodes individually in either *S* or *X* mode as appropriate. In the fine-grained version, we lock each tree node, leaf and non-leaf, individually in *S* or *X* mode. Note that careful implementation of the core tree operations into three logical phases (search, leaf, SMO) enables us to use the same code for all three granularities.

We are currently evaluating the functionality and performance of the implementation under varying workloads and configuration parameters. Initial experience shows that the medium-grained lock implementation can result in superior concurrency and throughput. While the fine-grained implementation provides the highest concurrency, the cost of locking and unlocking can be significant. In future systems, the relative cost of synchronization is likely to increase, making the situation even worse. The medium-grained implementation executes fewer lock operations than the fine-grained implementation. Moreover, in steady state, the number of SMOs is not significant, which reduces contention on the internal nodes of the tree (i.e., the higher you are in the tree, the lower the frequency of *X* mode locking).

VII. Conclusions and Future Work

We have shown that it is indeed possible to design and implement a highly concurrent B^{link}-tree using atomic blocks. Our design indeed supports multiple lock granularities and cursors. We obtain high concurrency through both finer-grained locking and deferring structure modifying operations. While the design is subtle, it appears necessarily so to obtain concurrency.

In the future we hope to offer detailed performance evaluation of our algorithm, exploring lock granularity, the advantages of cursors, etc., as well as implementing and comparing with a transactional memory version. We also hope to offer a more detailed proof of correctness.

References

[1] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica*, pp. 173–189, 1972.

[2] D. Comer, "The ubiquitous B-tree," *ACM Computing Surveys*, vol. 11, no. 2, pp. 121–137, 1979.

[3] G. Weikum and G. Vossen, *Transactional Information Systems*. Morgan Kaufmann, 2002.

[4] J. K. Metzger, "Managing simultaneous operations in large ordered indexes," Institut Fur Informatik, Technische Universitat Munchen, Tech. Rep., 1975.

[5] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*. Prentice Hall, 1999.

[6] T. Harris and K. Fraser, "Language support for lightweight transactions," in *Proceedings of OOPSLA*, 2003, pp. 14–25.

[7] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural support for lock-free data structures," in *Proceedings of the International Symposium on Computer Architecture*, 1993, pp. 289–300.

[8] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[9] Y. Sagiv, "Concurrent operations on B* trees with overtaking," *Journal of Computer and System Sciences*, vol. 33, no. 2, pp. 275–296, 1986.

[10] D. Lomet, "Simple, robust and highly concurrent B-trees with node deletion," in *Proceedings of the International Conference on Data Engineering*, 2004, pp. 18–28.

[11] D. Shasha and N. Goodman, "Concurrent search structure algorithms," *ACM Transactions on Database Systems*, vol. 13, no. 1, pp. 53–90, 1988.

[12] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, "System R: Relational approach to database management," *ACM Trans. Database Syst.*, vol. 1, no. 2, pp. 97–137, 1976.

[13] R. Bayer and M. Schkolnick, "Concurrency of operations on B-trees," *Acta Informatica*, vol. 9, no. 1, pp. 1–21, 1977.

[14] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems*, vol. 17, no. 1, pp. 94–162, 1992.

[15] Y. Mond and Y. Raz, "Concurrency control in B+-trees databases using preparatory operations," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 1985, pp. 331–334.

[16] G. Graefe, "Hierarchical locking in B-tree indexes," in *BTW 2007*, March 2007, pp. 18–42.

[17] P. Lehman and S. B. Yao, "Efficient locking for concurrent operations on B-trees," *ACM Transactions on Database Systems*, vol. 6, no. 4, pp. 650–670, 1981.

[18] V. Srinivasan and M. Carey, "Performance of B-tree concurrency algorithms," in *Proceedings of the ACM SIGMOD Conference*, 1991, pp. 416–425.

[19] T. Johnson and D. Shasha, "A framework for the performance analysis of concurrent B-tree algorithms," in *Proceedings of ACM Symposium on Principles of Database Systems*, 1990, pp. 273–287.

[20] K. Eswaren, J. Gray, R. Lorie, and I. Traiger, "Granularity of locks and degrees of consistency in a shared data base," IBM Research, Tech. Rep. RJ1654, 1975.

[21] D. Lomet, "Process structuring, synchronization, and recovery using atomic actions," in *Proceedings of the ACM Conference on Language Design for Reliable Software*, 1977, pp. 128–137.

[22] D. Lomet and B. Salzberg, "Concurrency and recovery for index trees," *VLDB Journal*, vol. 6, no. 3, pp. 224–240, 1997.

[23] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi, "Optimizing memory transactions," in *Proceedings of the ACM Conference on Programming Languages, Design, and Implementation*, 2003, pp. 388–402.

[24] L. P. Deutsch and D. G. Bobrow, "An efficient, incremental, automatic garbage collector," *Commun. ACM*, vol. 19, no. 9, pp. 522–526, 1976.

[25] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 491–504, June 2004.