Readings in Database Systems

Edited by
Michael Stonebraker
University of California, Berkeley

MORGAN KAUFMANN PUBLISHERS, INC.
SAN MATEO, CALIFORNIA

Concurrency of Operations on *B*-Trees

R. Bayer* and M. Schkolnick

IBM Research Laboratory, San José, CA 95193, USA

Summary. Concurrent operations on *B*-trees pose the problem of insuring that each operation can be carried out without interfering with other operations being performed simultaneously by other users. This problem can become critical if these structures are being used to support access paths, like indexes, to data base systems. In this case, serializing access to one of these indexes can create an unacceptable bottleneck for the entire system. Thus, there is a need for locking protocols that can assure integrity for each access while at the same time providing a maximum possible degree of concurrency. Another feature required from these protocols is that they be deadlock free, since the cost to resolve a deadlock may be high.

Recently, there has been some questioning on whether *B*-tree structures can support concurrent operations. In this paper, we examine the problem of concurrent access to *B*-trees. We present a deadlock free solution which can be tuned to specific requirements. An analysis is presented which allows the selection of parameters so as to satisfy these requirements.

The solution presented here uses simple locking protocols. Thus, we conclude that B-trees can be used advantageously in a multi-user environment.

1. Introduction

In this paper, we examine the problem of concurrent access to indexes which are maintained as *B*-trees. This type of organization was introduced by Bayer and McCreight [2] and some variants of it appear in Knuth [10] and Wedekind [13]. Performance studies of it were restricted to the single user environment. Recently, these structures have been examined for possible use in a multi-user (concurrent) environment. Some initial studies have been made about the feasibility of their use in this type of situation [1, 6], and [11].

An accessing schema which achieves a high degree of concurrency in using the index will be presented. The schema allows dynamic tuning to adapt its performance to the profile of the current set of users. Another property of the schema is that it is deadlock free. This is achieved by providing a set of strict locking protocols which must be followed by each process accessing an index. The properties of the locks and the protocols together guarantee that deadlocks cannot arise. Furthermore, the schema is shown to be a generalization of several methods used in earlier attempts to achieve concurrent operations in *B*-trees.

In Section 2, we define terms which will be used in the paper. Then in Section 3, we introduce the problem and some basic solutions to it. In Section 4, we present the general schema to be studied. In Section 5, the schema is shown to be deadlock free. Section 6 contains a quantitative analysis of the schema. Using the results of this analysis, tuning parameters can be selected to optimize the performance of the schema. Finally, Section 7 briefly discusses some extensions to the schema.

2. Definitions

We assume the reader is familiar with *B*-trees ([2, 10]). In the sequel we will be using the variant known as *B**-tree [13]. A *B**-tree with parameter k is a tree structure for storing entries. An *entry* is a pair (entry key, associated information). The keys are linearly ordered, the associated information is of no interest in this paper. *B**-trees have the following properties:

1) All entries are stored on leaf nodes. Each leaf node contains a number μ of entries.

2) All paths from the root to a leaf node have the same length.

3) All nonleaf nodes contain a number of elements: $p_0, r_1, p_1, r_2, p_2, \ldots, r_\mu, p_\mu$, where the p_i's are pointers to immediate descendants of this node and the r_i's are elements which can be compared with the keys in the entries. They are called *reference keys*. All keys in the subtree pointed to by p_{i-1} are less than the reference key r_i and all keys in the subtree pointed to by p_i are greater than or equal to the reference key r_i.

4) The number μ referred to in 1 and 3, may vary from node to node but satisfies $k \leqq \mu \leqq 2k$ for all nodes except for the root, where $1 \leqq \mu \leqq 2k$.

We will say that a node in a *B**-tree is at a *level i* if the path from that node to a leaf contains i nodes. Because of property 2, this number is well defined for all nodes. The level of the root is said to be the height of the tree. We will use $\ell(n)$ to denote the level of node n and h to denote the height of the tree.

An example of a *B**-tree with $h=3$ and $k=2$ is shown in Figure 1. Entries on the leaf nodes are shown as parenthesized objects. The value in parentheses is the entry key. Nodes have been given labels in order to refer to them.

The operations to be performed on these structures will be of three kinds: A *search* for a given key, an *insertion* of a given entry, and a *deletion* of an entry with a given key. A process executing the first operation is said to be a *reader*. A process executing the second or third operation is said to be an *updater*. Note that a search does not result in a modification of the tree. If an insertion (respectively a deletion) is attempted and the key to be inserted (deleted) is (is not) found

* *Permanent address:* Institut für Informatik der Technischen Universität München, Arcisstr. 21, D-8000 München 2, Germany (Fed. Rep.)

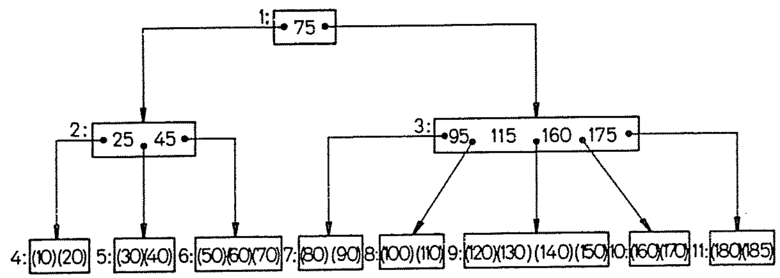

Fig. 1. A B^*-tree with $h=3$, $k=2$

in a leaf node then the insertion (deletion) is said to be *unsuccessful*. A successful operation done by an updater results in a modification of the tree. We assume the reader is familiar with the way these modifications affect the tree structure. We will use the following important fact: When an updater attempts an insertion (deletion) and scans node n, it can easily check a sufficient condition on n that any ancestors will not be affected by the insertion (deletion). If the condition is satisfied, n will be known as *safe*; otherwise, n is said to be *unsafe*.

In a B^*-tree, the criterion for determining safeness of a node is very simple: on insertions, a node is safe if the number μ is less than $2k$. On deletions, a node is safe if the number μ is greater than k. For example, a process trying to insert the key (186) in the tree shown in Figure 1 would first scan the root (node 1) and it would determine it is safe. It would then branch to node 3. Since for this node $\mu = 2k$, it would determine this node is unsafe, i.e., it cannot tell whether its ancestors, in this case node 1, would be affected by the insertion. Finally, it would move on to node 11 which it would determine to be safe, i.e., neither node 3 nor node 1 will be affected by the insertion. (In what follows, we will not worry about changing the unsafeness status of node 3 to being safe.)

The fact that we can determine safeness of a node is the feature of a B^*-tree that is used in all solutions described in this paper. In fact, any other tree structure for which the same property can be established can be accessed concurrently using the protocols described here. Examples of these are prefix B-trees [4] and enciphered B-trees [3].

3. Basic Solutions

In a multiuser environment, concurrent access of processes to an index structure must be supported. The problem of concurrent access is that of allowing a maximum number of processes to operate on the tree without impairing the correctness of their operations.

A simple-minded solution for the problem of concurrent access would be to strictly serialize all updaters, by requiring each updater to gain exclusive control of the tree – e.g., by placing an exclusive lock on the whole tree – before it begins accessing it, thus, preventing all other updaters and readers from altering or reading the index while the specific update takes place. Readers, on the other hand, could access the structure concurrently with other readers. Clearly this simple mechanism can only be used if the level of activity is rather low.

Fig. 2. Compatibility graph for locks: Solutions 1 and 2

We will now present three solutions to the problem of concurrent access in a B^*-tree. For each solution we will give a protocol for both readers and updaters. All solutions use locks on the nodes of the tree. These locks are granted by a scheduler upon request by a process. We will assume that, except as noted in Solution 3, the scheduler services these requests in a FIFO order. This order is maintained by having one service queue for every node in the tree. A process requesting a lock on a node will be placed at the end of the queue for that node and the scheduler will service processes that are at the beginning of the queues.

Solution 1. This solution is essentially the one presented by Metzger [11]. It is derived from the simple-minded solution when the fact that safeness of a node can be established is used. The solution uses two types of locks: a *read* lock, or ρ-lock, and an *exclusive* lock, or ξ-lock. A node cannot simultaneously be locked with a ρ-lock and a ξ-lock. In fact, these locks satisfy the compatibility relation shown in Figure 2.

An edge between any two nodes in a compatibility graph means that two *different* processes may simultaneously hold these locks on the same node. The absence of an edge indicates that two different processes cannot hold these locks simultaneously on a node. These constraints are enforced by the lock scheduler.

The protocol for readers is as follows:

0) Place ρ-lock on root;
1) Get root and make it the current node;
main loop: 2) **While** current node is not a leaf node **do**
{Exactly one ρ-lock is held by process}
begin
3) Place ρ-lock on appropriate son of current node;
4) Release ρ-lock on current node;
5) Get son of current node and make it current;
end mainloop

By executing this protocol, a reader would scan the B^*-tree, starting at the root and moving down towards a leaf node.

The protocol for an updater is as follows:

0) Place ξ-lock on root;
1) Get root and make it the current node;
main loop: 2) **While** current node is not a leaf node **do**
{number of ξ-locks held $\geqq 1$}

```
begin
   3) Place ξ-lock on appropriate son of current node;
   4) Get son and make it the current node;
   5) If current node is safe
      then release all locks held on ancestors of current node
end
```

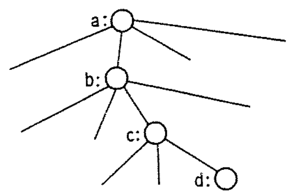

Fig. 3. Skeletal B^*-tree

As an example of how this last protocol would work, consider an update on the node d of the (skeletal) B^*-tree shown in Figure 3. Assume that, for this update, nodes a, b, and d are safe and c is not. Before execution of the mainloop, a ξ-lock would be placed on node a and this node would be scanned.

Then the following sequence of events would take place.

i) [Step 3] A ξ-lock is requested on node b.
ii) [Step 4] After ξ-lock is granted, node b is retrieved.
iii) [Step 5] Since node b is safe, the ξ-lock on node a is released, thereby, allowing other updaters or readers to access node a.
iv) [Step 3] A ξ-lock is requested on node c.
v) [Step 4] After ξ-lock is granted, node c is retrieved.
vi) [Step 5] Since node c is unsafe, the ξ-lock on node b is kept.
vii) [Step 3] A ξ-lock is requested on node d.
viii) [Step 4] After ξ-lock is granted, node d is retrieved.
ix) [Step 5] Since node d is safe, the ξ-locks on nodes b and c can be released.

This solution has the advantage of requiring only a simple protocol to achieve a reasonable gain in concurrency over the simple minded solution described at the beginning of this section. However, it suffers from the fact that updaters first ξ-lock the root of a subtree when updating this subtree (and thus preventing all other accesses to this subtree) even when, as it happens most of the time, the update will have no effect on this root. In the above example, the entire tree remained ξ-locked while node b was being retrieved and examined (a slow process). In turn, the subtree rooted at b had a ξ-lock on its root while both nodes c and d were retrieved and examined to find that node b would not be modified as a result of the update.

To achieve higher concurrency one may let updaters behave like readers in the upper part of the tree. This leads to the next solution.

Solution 2. This solution is a variant of one used by one of the authors in the design of an interactive data base system [12]. It uses the same locks as in Solution 1. The protocol for a reader is also as in Solution 1. Updaters however, have a different protocol, as follows:

```
            0) Place ρ-lock on root;
            1) Get root and make it the current node;
main loop:  2) While current node is not a leaf node do
               begin
                  3) If son is not a leaf node
                        then place ρ-lock on appropriate son
                        else place ξ-lock on appropriate son;
                  4) Release lock on current node;
                  5) Get son and make it the current node
               end mainloop;
            6) {A leaf node has been reached}
               If current node is unsafe
               then release all locks and repeat access to the tree, this time
                  using the protocol for an updater as in Solution 1;
```

By following this protocol, updaters will proceed down the tree as if they were readers until they are about to go to a leaf node. At this point, they ξ-lock the leaf node in order to make the update. However, if it is found that the update would affect nodes higher in the tree, all the analysis done so far would be lost and the update must be retried (note that this only involves the release of one lock and the time lost in scanning the tree). There is no actual modification done to any node which would have to be restored.

The protocol shown works for trees of height greater than one but it is a simple matter to accommodate for this case, so from now one we assume all of our trees have heights greater than one.

Solution 2 achieves high concurrency by allowing both readers and updaters to share all higher levels of the tree. It is only when the update is to be performed on an unsafe leaf that the updater adopts the protocol of the previous solution and thus prevents concurrency as occurred in that solution. Since this only happens roughly once every k updates done to the tree it is a very infrequent action, for typical uses of B^*-trees is with a large k [2].

If the time spent in the unsuccessful analysis or the interference with readers becomes critical, as would be for example, on a very deep tree, then Solution 3 is more attractive.

Solution 3. This solution uses three types of locks, a ρ-lock, an α-lock, and a ξ-lock. The compatibility graph is shown in Figure 4. In this diagram, a new type of edge is shown by a directed broken line from the α node to the ξ node. This means that the α-lock can be converted into a ξ-lock.

Conversion from one type of lock to another type is taken to be a basic (or atomic) operation which happens in one step. To request a conversion from one type of lock to another, a process has to hold a lock of the first type on a node. When making a request for a conversion, a process will be placed *at the beginning* of the queue for the node in question (thus, these conversion requests are serviced before any other requests). If the conversion is granted the process now holds a lock of the second type on this node. If the conversion requests a lock incompatible with a lock placed by another process then the conversion is not granted

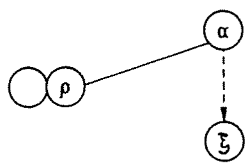

Fig. 4. Compatibility and convertibility graph for locks: Solution 3

and the requesting process is placed on a wait status at the beginning of the queue for that node.

For the three types of locks as defined for Solution 3, the only time a conversion (from α-lock to ξ-lock) fails is when another process holds a ρ lock on the resource on which a conversion is being attempted. (Note that a ρ-lock is the only type of lock which is compatible with an α-lock.) An attempt to convert this α-lock into a ξ-lock will be delayed until the ρ-lock on the resource is released since the ξ-lock is incompatible with the ρ-lock.

Readers use the same protocol as in Solution 1. Updaters now use the following protocol:

```
            0) Place an α-lock on the root;
            1) Get the root and make it the current node;
main loop:  2) While current node is not a leaf node do
               {number of α-locks held ≧1}
               begin
                 3) Place an α-lock on appropriate son of current node;
                 4) Get son and make it the current node;
                 5) If current node is safe
                    then release all locks held on ancestors of current node
               end mainloop;
            6) {A leaf node has been reached. At this time we can determine
               if update can be successfully completed.}
               If the update will be successful
               then convert, top-down, all α-locks into ξ-locks;
```

Using this protocol, an updater descends the tree as in Solution 1 but using α-locks instead of ξ-locks. This has the advantage of allowing readers to share the nodes on which an updater has placed its α-locks, thus increasing concurrency. On the other hand, all nodes that need to be modified as a result of the update, are locked exclusively after Step 6. Thus, the analysis phase need not be repeated, as occurred in Solution 2. Moreover, ξ-locks are placed only on those nodes that will be modified, thus readers are prevented from examining only the minimal possible set of nodes.

The main disadvantage of this solution is that, as in Solution 1, one updater may temporarily block other updaters from scanning a node, even if this node will not be affected by the update. Also, there is overhead time spent in doing lock conversions.

Although the α-lock on the leaf node needs to be converted to a ξ-lock everytime the update can be successfully completed, conversion of locks in higher nodes occurs as infrequently as the repetition of analysis in Solution 2, a very small proportion of time. Note that the required α to ξ conversion on the leaf node could be eliminated by changing the protocol to set up a ξ-lock directly on a leaf node, instead of an α-lock. If the update affects higher nodes still held with α-locks, then as will be seen in the generalized solution, the ξ-lock on the leaf must first be converted to an α-lock and then the α- to ξ-lock conversion can be made. Thus, the very frequent operation of converting the α-lock on the leaf to a ξ-lock can be replaced by a slightly more complicated protocol and the infrequent conversion of a ξ-lock on a leaf to an α-lock. A new conversion property among locks is needed, namely ξ-locks must be convertible into α-locks.

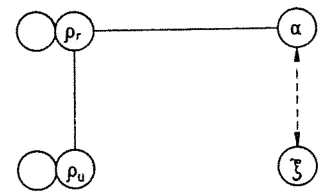

Fig. 5. Compatibility and convertibility graph for locks: Generalized solution

4. A Generalized Solution

In the previous section, we presented three solutions to the problem of deadlock free concurrent operations on *B*-trees. We observed that these solutions were complimentary in the sense that each had advantages over the other in certain situations. Thus, none of them was a best solution in all possible cases. This suggests that a combined solution may be more suitable. In this section, we present such a generalized solution.

There are 4 locks needed in this approach. A ρ_r-lock, a ρ_u-lock, an α-lock, and a ξ-lock. Their compatibility-convertibility diagram is shown in Figure 5. Note that, as mentioned already in the remarks following Solution 3, there is a need for conversion from α to ξ and from ξ to α.

The protocol for a reader is as in Solution 1, with ρ_r replacing ρ. The protocol for an updater is given below. As can be observed, there are two parameters P and Ξ. Intuitively, these stand for the maximum number of levels in the tree on which an updater may place ρ_u-locks and ξ-locks respectively. A variable *H* is introduced which has as its value, the current value of the height $\hbar$ of the tree. In an implementation of a *B**-tree the value $\hbar$ can be stored in a directory entry together with a pointer to the root of the tree. The variable *H* then refers to this entry and the root of the tree is considered a descendant of *H*.

Protocol for an updater:

```
begin
  {Let variable H always contain the height ℏ of the tree, ℏ≧0}
  procedure process son of current;
```

```
begin       get son of current;
            current:=son of current;
            if current is safe
            then remove locks on all ancestors of current;
end;
0) If P≠0 then place ρ_u-lock on H else place α-lock on H;
   current:=H; {root=son of H};
1) Ξ̄:=min {h, Ξ};
   P̄:=min {P, h−Ξ̄};
   ᾱ:=h−Ξ̄−P̄;
2) for L:=1 step 1 until P̄ do
     begin place a ρ_u-lock on son of current;
           release ρ_u-lock on current;
           get son of current;
           current:=son of current
     end;
3) for L:=1 step 1 until ᾱ do
     begin place an α-lock on son of current;
           process son of current;
     end;
4) for L:=1 step 1 until Ξ̄ do
     begin place a ξ-lock on son of current;
           process son of current;
     end;
5) if ρ_u-lock still held
   then begin release all locks;
              P=0; Ξ=0;
              repeat protocol and exit.
   end;
6) if α-locks still held
   then begin 6a): convert top-down all ξ to α;
              6b): convert top-down all α to ξ;
   end;
7) MODIFY: modify all nodes with ξ-locks, requesting additional ξ-locks
           for overflows, underflows, splits and merges as necessary;
8) release all locks;
end;
```

After Step 6 of this protocol is executed, the updater can proceed with the actual change. This is done in Step 7. All nodes in the locked subpath are locked with ξ-locks. The rules for insertion and deletion on B^*-trees determines whether additional ξ-locks will be acquired. This happens when attempting an overflow (or underflow) into a brother. This situation is shown in Figure 6. Assume that, at the end of Step 6, nodes b and c are held with ξ-locks and an update operation is to be performed on node c. Nodes d and e are immediate brothers of c. Since b has a ξ-lock we know that the update on c will propagate up to b. An overflow (or underflow) operation is then attempted into one of d or e. To do this, a ξ-lock is requested on d and when granted, an attempt is made to combine c and d

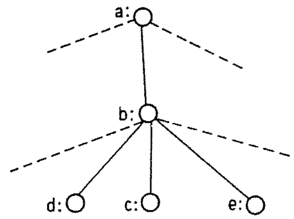

Fig. 6. Example of update

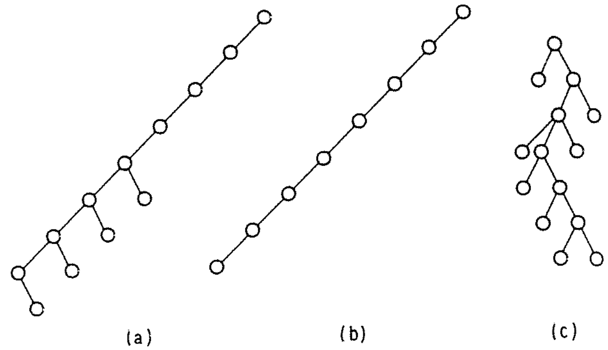

Fig. 7. Combs

together. (If this is not possible, an attempt to combine c and e is made.) After the required modifications at this level have been completed, node b is updated. Note that node b is safe (since there is no lock on node a) so after it is modified, the update terminates.

The following observations follow directly from the protocols:

Observation 1. All nodes which are locked by a process form a comb. (A comb of a tree is a subtree with the following restriction: if a node has more than one subtree as descendants, only one of them can have more than one node.) Examples of combs are shown if Figure 7.

Observation 2. If a process holds a ρ_r, ρ_u, or α-locks, then its comb is reduced to a path (as in Fig. 7b). Let this path be $(p_1, p_2, \ldots, p_n)$. Then

a) The process has not been granted any α to ξ conversions. In this case, there are integers j, k, with $0 \leqq j \leqq k \leqq n$ such that all nodes $p_1, p_2, \ldots, p_j$ have ρ_r-locks (if a reader, in this case $j=k=n$) or ρ_u-locks (if an updater), all nodes $p_{j+1}, \ldots, p_k$ have α-locks and all nodes $p_{k+1}, \ldots, p_n$ have ξ-locks.

b) The process has been granted α to ξ lock conversions. Then it no longer holds ρ_u-locks, p_n is a leaf node and there is an integer k such that $1 \leqq k \leqq n$, $p_1, \ldots, p_k$ have ξ-locks and $p_{k+1}, \ldots, p_n$ have α-locks.

In the following sections we will examine these protocols more closely. We will show that they are deadlock free and will analyze the amount of concurrency they provide.

5. Deadlock Freeness of the Generalized Solution

In this section we will show that the generalized solution is deadlock free. As can be observed, the protocols for this solution are combinations of protocols for Solutions 1, 2, and 3. In fact, the main loop of each of these solutions can be obtained from the generalized solution by appropriate choices of the parameters P and Ξ (this will be done in Section 6) and by identifying both ρ_r and ρ_u locks with

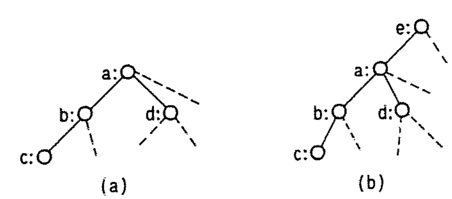

Fig. 8. Example of deadlock

the ρ lock. Since each one of Solutions 1, 2, and 3 can be shown to be deadlock free, it would appear that a solution as presented in Section 4 but with just 3 locks, a ρ-lock (instead of a ρ_r and a ρ_u-lock), an α-lock and a ξ-lock could be also shown to be deadlock free (the ρ-lock would be compatible with itself and an α-lock but not with a ξ-lock). Thus a simpler generalized solution would be obtained. This turns out not to be the case as shown by the following example:

Example. Consider a tree as in Figure 8a, where node c is unsafe and all other nodes shown are safe.

Assume $P=2$, $\Xi=1$. An updater, U_1 on node c would: set up a ρ-lock on a and get node a;
set up a ρ-lock on b; release ρ-lock on a; get b;

Now, other updaters can enter the tree through node a and go down towards node d. As a result of successive updates along this path, node a may split and a new root e be created as in Figure 8b.

A second updater U_2 may now want to update node c.
He would then: set up a ρ-lock on node e; get node e;
set up a ρ-lock on node a; release ρ-lock on e; get a;
set up an α-lock on b (this would be granted since the only other lock on b, which is held by U_1, is a ρ-lock, compatible with α);
get b;
since b is safe, release ρ-lock on a;
get ξ-lock on node c; get node c since node c is unsafe, no locks are released;
At this point, U_2 would begin converting its ξ-locks into α-locks:
Convert ξ-locks in node c into an α-lock.
Then, the α-locks would be converted into ξ-locks:
Convert α-lock in node b into a ξ-lock;

This last conversion would be blocked since there is another process, U_1, holding a ρ-lock on b which is incompatible with a ξ-lock. Thus U_2 could not proceed. But U_1 is now also blocked since it would try to set up a ξ-lock on node c next. This request cannot be granted since another process, U_2, holds an α-lock which is incompatible with a ξ-lock. Thus, U_1 and U_2 are in a deadlock situation.

Forcing a read lock requested by an updater to be incompatible with an α-lock, as is done in the generalized solution by distinguishing between the ρ_r and ρ_u locks, resolves the above problem. In fact, as we now proceed to show, it makes this solution deadlock free. We first introduce some definitions:

Definition 1. A lock request on a node is said to be *pending* until granted by the scheduler.

Definition 2. Given two processes U and V, we say that U *waits on* V, denoted $U \vdash V$, if U has a pending request to lock a node on which V has a lock incompatible with the one U is requesting.

Definition 3. A process is called a c-process if it has a pending request for a lock conversion.

Definition 4. If a process U has a pending lock request on a node n then n is said to be the *critical node* of U. If U is not requesting a lock or the request was granted then U does not have a critical node.

Definition 5. The *critical level* of a process U, denoted by $\lambda(U)$ is the level of its critical node, if U has one, or 0 otherwise.

Our intent is to show that the given locking protocol is deadlock free. Intuitively, this means that any given process using these protocols to request locks will not be forever prevented from completing its task because of the existence of other locks placed by other processes. We must, however, be careful not to include the lock scheduler as an interfering process. In fact, a lock scheduler can arbitrarily produce deadlock situations by consistently failing to service a given process request, thus preventing it from completing its task. We then request that the scheduler have the following properties:

a) It shouldn't grant incompatible lock requests (or conversions) since this would create a situation inconsistent with the attributes of the locks.

b) It should grant lock conversion requests on a node before any other requests on that node. Note that since both α and ξ locks are incompatible among themselves, there can be at most one lock conversion request on any given node. Granting the unique lock conversion possible on a node before other requests eliminates the possibilities of trivial deadlock situations.

c) It should be fair in servicing requests, i.e., there should be a finite number of requests granted by the scheduler before a given request is finally granted. Servicing a request which doesn't result in an incompatible lock to be placed in a node should result in granting the requested lock.

There are many lock schedulers satisfying these restrictions and we have chosen one using a FIFO model to illustrate our protocols. Any other scheduler satisfying a), b), and c) will also result in a deadlock free operation.

Now we present a series of lemmas, all of which follow from the observations on the protocols made at the end of the previous section.

Lemma 1. If a process V holds a ρ_r or ρ_u-lock on a node m then

$$\lambda(V) < \ell(m).$$

(Recall that $\ell(m)$ is the level of node m.)

Proof. Follows directly from observation 2a. □

Lemma 2. If a process V holds a ξ-lock on a node m then either
2a) $\lambda(V) < \ell(m)$ or
2b) $\lambda(V) \geqq \ell(m)$ and V also holds a ξ-lock on the father of m.

Proof. We use observation 2. If V has not requested α to ξ conversions then Case 2a applies (Steps 0 through 5 of the protocol). In Step 6a) since a ξ to α conversion is always granted then $\lambda(V)=0$. In Step 6b), α to ξ conversion is top down, and $\lambda(V)<$level of any node on which V holds a ξ-lock. Thus, $\lambda(V)<\ell(m)$. If, on the other hand, V has been granted all α to ξ conversions then V holds a comb made up of nodes all of which are ξ-locked and V is performing the actual update on some node in Step 7. If $\lambda(V)\geqq\ell(m)$ it means that V is acquiring a ξ-lock on a node q to perform an overflow (or underflow) or split (or merge) operation. But in this case, V holds a ξ-lock on the father r of q. Since all nodes held with ξ-locks by V form a comb, if $\lambda(V)\geqq\ell(m)$ it means that m is a descendant of r, and V holds a ξ-lock on the father of m. This completes the proof of Lemma 2. □

Lemma 3. If a process V holds an α-lock on a node m then either

3a) $\lambda(V)<\ell(m)$ or

3b) $\lambda(V)=\ell(m)$ and V is attempting an α to ξ conversion on node m or

3c) $\lambda(V)>\ell(m)$ and V is attempting an α to ξ conversion and has an α-lock on the father of m.

Proof. Assume $\lambda(V)\geqq\ell(m)$. Since V holds an α-lock, observation 2 applies. Thus, if $\lambda(V)=\ell(m)$, V must be attempting an α to ξ conversion on m while if $\lambda(V)>\ell(m)$, V must be attempting an α to ξ conversion on an ancestor of m, and since all nodes locked by V form a path, holds an α-lock on a father of m.

This proves the lemma. □

Lemmas 1, 2, and 3 are now used to prove Lemma 4. This is the key lemma in proving deadlock freeness of the generalized solution.

Lemma 4. If U, V are processes and $U\vdash V$ then either

4a) $\lambda(U)>\lambda(V)$ or

4b) $\lambda(U)=\lambda(V)$, V is a c-process and U is not a c-process.

Proof. Let $U\vdash V$ and let m be the critical node for U. Thus $\lambda(U)=\ell(m)$. We consider 3 cases.

Case 1. If V holds a ρ_u or ρ_r lock on m, then by Lemma 1,

$$\lambda(U)=\ell(m)>\lambda(V)$$

and Case 4a) of Lemma 4 holds.

Case 2. If V holds a ξ-lock on m, Lemma 2 applies. Thus, either $\lambda(V)<\ell(m)$ and the lemma holds or $\lambda(V)\geqq\ell(m)$. But this last case is not possible for then V would hold a ξ-lock on the father of m also, which means that U could not have any locks set up on the father of m and so it could not be attempting to acquire a lock on m. Clearly, U could not be attempting a conversion either since V has a ξ-lock on m.

Case 3. If V holds an α-lock on m, Lemma 3 applies. Thus, either $\lambda(V)<\ell(m)$ (and we are done) or $\lambda(V)\geqq\ell(m)$. If $\lambda(V)=\ell(m)$, we know V is attempting an α- to ξ-conversion on node m. But since no two processes can be attempting a conversion simultaneously on the same node (they would both have to hold α-locks on the node which is not possible) we get that U cannot be a c-process. Finally, if $\lambda(V)>\ell(m)$ then V holds α-locks on both m and its parent. We already saw that U cannot be attempting a conversion on node m since V has an α-lock on it. Thus, U must be attempting to acquire a new lock on m. But to do this it must have a lock on a parent of m. If U were a reader then U would not be waiting to get a ρ_r-lock on m. Thus U has to be an updater. But this cannot happen since any lock U holds on a parent of m is incompatible with α. This concludes the proof of the lemma. □

Lemma 4 allows us to show:

Theorem. The generalized solution is deadlock free.

Proof. Assume to the contrary that a deadlock exists. Then, there exist processes $U_1, U_2, \ldots, U_k$ such that

$$U_1\vdash U_2\vdash U_3\vdash\cdots\vdash U_{k-1}\vdash U_k\vdash U_1 \qquad (*)$$

and there is no way to grant any pending locks in the chain. Note that since $U_1\vdash U_1$ cannot happen, $k\geqq 2$ and so, there are at least three processes U_1, U_2, U_3 with $U_1\vdash U_2\vdash U_3$ and $U_2\neq U_1$, $U_2\neq U_3$.

By Lemma 4, $\lambda(U_1)\geqq\lambda(U_2)$ with $\lambda(U_1)=\lambda(U_2)$ if U_2 is a c-process. On the other hand, $\lambda(U_2)\geqq\lambda(U_3)$ with $\lambda(U_2)=\lambda(U_3)$ only if U_2 is not a c-process. Thus $\lambda(U_1)>\lambda(U_3)$.

The above result implies that in $(*)$ we have $\lambda(U_1)>\lambda(U_1)$ a contradiction. Thus, the theorem holds. □

6. Selection of P and Ξ

As presented in Section 4 the generalized solution depends on the parameters P and Ξ.

By varying these parameters, many different concurrency patterns can be obtained.

For example, if $P=0$, $\Xi=h$ Steps 2 and 3 of the protocol would not be executed and Step 4 essentially reduces this solution to Solution 1. Note that setting up Ξ to h is not possible since one does not know in advance how high the tree is. But it is easy to define a way of simulating the protocol for the generalized solution to work as if one knew what h was before accessing the tree. Also, in order to get Solution 1, one has to change ρ_r to ρ, but clearly this is no problem either since α-locks are not present so that the locks ρ_r and ξ in the generalized solution can be mapped to the ρ and ξ-lock respectively of Solution 1. In the sequel, when saying that the generalized solution reduces to one of Solution 1, 2, or 3 we mean it modulo these types of changes.

If we let $P=h-1$ and $\Xi=1$ then we get Solution 2 (in this case, the retry would have to have $P=0$, $\Xi=h$). Finally, setting $P=0$, $\Xi=0$ one gets Solution 3. As was mentioned in Section 3 each one of these solutions has advantages and disadvantages over the other. A proper choice of P and Ξ will *tune* the generalized solution to yield the best performance for a given application. In what follows we show a model of access with an analysis of the relevant components of the cost of a solution. Expressions for these components will be obtained from which

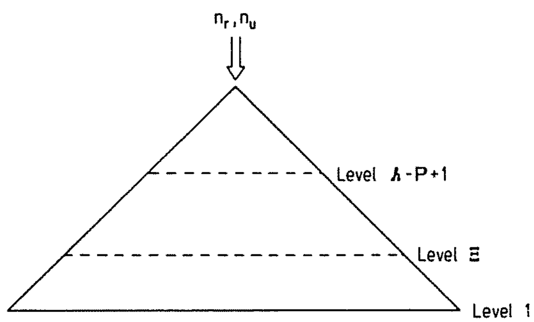

Fig. 9. Model for concurrency and overhead analysis

suitable values of P and Ξ can be chosen. We will assume that all readers and updaters access the structure using the same P and Ξ (as will be explained below, this may not be the case, but helps to evaluate a strategy).

There are two main components in the cost of a given solution. One is the time spent by processes waiting for locks to be removed before they can proceed. The second one is the time overhead due to placing locks on the nodes, converting locks, or repeating part of an analysis.

Assume a tree as in Figure 9, and a given number n_r of readers and n_u of updaters. The updaters will place ρ_u-locks from level h down to level $h-\mathrm{P}+1$. From level $h-\mathrm{P}$ down to $\Xi+1$ they will place α-locks and finally, for the last Ξ levels, they will place ξ-locks.

Now, in the upper P levels there are no conflicts, since updaters and readers use compatible ρ_u and ρ_r locks. But in level $h-\mathrm{P}$, the updaters set up α-locks which are incompatible among themselves. Thus, at this level, some updaters will have to wait for other updaters.

Let v_i denote the number of nodes of the tree at level i. We will assume that each updater has equal probability $\frac{1}{v_i}$ of scanning each node at level i when traversing the tree from the root to a leaf node. Then, since there are n_u updaters, the expected number of nodes which are visited by the n_u updaters at level $h-\mathrm{P}$ is given by

$$\Phi(h-\mathrm{P}, n_u) = v_{h-\mathrm{P}}\left(1-\left(1-\frac{1}{v_{h-\mathrm{P}}}\right)^{n_u}\right).$$

To obtain this expression, note that $\left(1-\frac{1}{v_{h-\mathrm{P}}}\right)^{n_u}$ is the probability that a given node will not be scanned by any of the n_u updaters. Thus, $1-\left(1-\frac{1}{v_{h-\mathrm{P}}}\right)^{n_u}$ gives the probability that a given node will be scanned by at least one updater. The expected number of nodes which will be scanned by at least one updater is then $\Phi(h-\mathrm{P}, n_u)$.

This expression also gives the expected number of updaters that will proceed down the tree (from level $h-\mathrm{P}$) without waiting for an α-lock to be granted. Thus, the expected number of updaters that will wait is given by:

$$W_u = n_u - \Phi(v_{h-\mathrm{P}}, n_u).$$

The $\Phi(h-\mathrm{P}, n_u)$ updaters that can proceed downwards from level $h-\mathrm{P}$ will do so without interfering with each other. When they get to level Ξ they may interact with readers. Assuming the updaters acquire the ξ-locks before the readers request ρ_r-locks (this will give a worst case value for the quantity being computed), the expected number of readers that wait is:

$$W_r = \begin{cases} n_r \dfrac{\Phi(h-\mathrm{P}, n_u)}{v_\Xi} & (\text{if } \Xi \neq 0) \\ 0 & (\text{if } \Xi = 0). \end{cases}$$

W_u and W_r together give a measure of the number of processes that will have to wait when accessing the tree. Besides this component of the cost of a (P, Ξ) solution there is the overhead cost involved. This cost has three subcomponents. One is given by the fact that, after scanning the tree from the root to the leaf, a process may find that all his processing has to be repeated (this happens if in Step 5 of the protocol an updater finds that it still holds a ρ_u-lock). We will measure this component by computing Q, the expected number of nodes per updater that are scanned again to repeat an analysis. The second subcomponent is C_ξ, the expected number of ξ-locks that an updater will convert into α-locks. Finally, the third subcomponent is C_α, the expected number of α-locks that an updater will convert into ξ-locks.

To compute Q, we will assume that all updaters are performing insertions. In this case, one out of k updaters will, on the average, cause a split of a node at the leaf level which propagates up the tree; one out of k^2 updaters will, on the average, cause a split of a node at level 2 which will propagate up the tree, and so on. Thus, we consider the probability that an updater will *modify* a node at level i or above to be $\left(\frac{1}{k}\right)^{i-1}$.

An updater will repeat his analysis if it causes a node at level $h-\mathrm{P}+1$ or above to be modified. Since when this happens, h nodes will be scanned again, we have that

$$Q = \begin{cases} h \cdot \left(\dfrac{1}{k}\right)^{h-\mathrm{P}} & \text{if } \mathrm{P} \neq 0 \\ 0 & \text{if } \mathrm{P} = 0. \end{cases}$$

(Note that if P = 0, there is no retry involved.)

To compute C_ξ we note that an updater will convert ξ-locks into α-locks whenever it reaches Step 5 of the protocol and discovers it holds an α-lock, but not a ρ_u-lock. This, in turn happens only if the update will modify nodes at a level $\Xi+1$ or higher, but lower than level $h-\mathrm{P}+1$. The number of locks to be modified in this case is always Ξ. Thus,

$$C_\xi = \begin{cases} \Xi \cdot \left[\left(\dfrac{1}{k}\right)^{\Xi} - \left(\dfrac{1}{k}\right)^{h-\mathrm{P}}\right] & \text{if } h > \mathrm{P}+\Xi \\ 0 & \text{otherwise.} \end{cases}$$

Finally, to compute C_α, we note that if the update will modify nodes exactly up to a level i, $\Xi+1 \leq i \leq h-\mathrm{P}$ then i α-locks are converted into ξ-locks. Thus, if $h > \mathrm{P}+\Xi$

$$C_\alpha = \sum_{i=\Xi+1}^{h-\mathrm{P}} i \cdot \left[\left(\frac{1}{k}\right)^{i-1} - \left(\frac{1}{k}\right)^{i}\right] = \frac{k-1}{k} \sum_{i=\Xi+1}^{h-\mathrm{P}} i \left(\frac{1}{k}\right)^{i-1}.$$

Clearly, if $h \leq \mathrm{P}+\Xi$ no α-locks are placed in the first place so $C_\alpha = 0$.

In Table 1, we give values for these 5 components for various choices of h, k, n_r, and n_u. Notice that W_u and W_r are shown in two columns, a high and a low. This is because the actual number of nodes at level i, v_i can fluctuate according to:

$$v_h = 1$$
$$2 \leq v_{h-1} \leq 2k+1$$
$$2(k+1)^{h-i-1} \leq v_i \leq (2k+1)^{h-i} \qquad i \leq h-2.$$

The high value is obtained when using the upper bound for v_i and the low value is obtained when using the lower bound for v_i.

From Table 1, we can choose values of P and Ξ which will guarantee an average performance prescribed in advance. For example, a concurrency of more than 50% of the updaters and more than 99% of the readers would be possible, in the case $h=5$, $k=10$, $n_u=30$, $n_r=70$ by choosing $\Xi=1$, $\mathrm{P}=2$. The average number of nodes that are scanned again after a retry by an updater is 0.005 and the number of lock conversions per updater is on the average, 0.099 for ξ to α conversion and 0.207 for α to ζ conversion.

For $h=3$, $k=100$ good concurrency levels are achieved with $\Xi=1$ and $\mathrm{P}=1$ or 2. In the latter case, there is an increase in the number of nodes that are accessed before a retry which is compensated by reducing to 0 the number of lock conversions.

We have shown how to select the parameters P and Ξ to obtain a given level of concurrency. This assumes that all updaters use the update protocol with the same values of P and Ξ. But there is nothing that prevents an updater from using its own P and Ξ. By doing this, an updater may use information he has gathered on previous accesses to further contribute to an increase in concurrency. For example, an updater that accesses an index to perform an insertion on a leaf node he has visited recently and found to be very far from full could choose $\mathrm{P}=h-1$ and $\Xi=1$ and be almost guaranteed not to perform a retry while at the same time allowing for maximum concurrency with other processes. (If the updater had found the node almost full on a previous access, it may use $\mathrm{P}=h-2$, $\Xi=1$ to insure that even a split to level 2 would not cause a retry and still allow for high concurrency!)

The fact that each updater may use its own parameters P and Ξ gives the generalized solution an added flexibility while at the same time preserving deadlock freeness of the schema. In fact, in proving deadlock freeness in Section 5, no assumptions where made as to the values for P and Ξ each updater might choose.

Table 1

			$n_u=5$	$n_r=95$	$h=5$	$k=10$		
Ξ	P	W_u low	W_u high	W_r low	W_r high	Q	C_ξ	C_α
0	0	4.00	4.00	0.00	0.00	0.0000	0.0000	1.1110
0	1	3.06	0.45	0.00	0.00	0.0005	0.0000	1.1106
0	2	0.43	0.02	0.00	0.00	0.0050	0.0000	1.1070
0	3	0.04	0.00	0.00	0.00	0.0500	0.0000	1.0800
0	4	0.00	0.00	0.00	0.00	0.5000	0.0000	0.9000
1	0	4.00	4.00	0.04	0.00	0.0000	0.1000	0.2110
1	1	3.06	0.45	0.07	0.00	0.0005	0.0999	0.2106
1	2	0.43	0.02	0.16	0.00	0.0050	0.0990	0.2070
1	3	0.04	0.00	0.18	0.00	0.0500	0.0900	0.1800
1	4	0.00	0.00	0.18	0.00	0.5000	0.0000	0.0000
2	0	4.00	4.00	0.39	0.01	0.0000	0.0200	0.0310
2	1	3.06	0.45	0.76	0.05	0.0005	0.0198	0.0306
2	2	0.43	0.02	1.79	0.05	0.0050	0.0180	0.0270
2	3	0.04	0.00	1.95	0.05	0.0500	0.0000	0.0000
3	0	4.00	4.00	4.32	0.22	0.0000	0.0030	0.0040
3	1	3.06	0.45	8.37	0.98	0.0005	0.0027	0.0036
3	2	0.43	0.02	19.72	1.07	0.0050	0.0000	0.0000
4	0	4.00	4.00	47.50	4.52	0.0000	0.0004	0.0004
4	1	3.06	0.45	92.03	20.57	0.0005	0.0000	0.0000
5	0	4.00	4.00	95.00	95.00	0.0000	0.0000	0.0000

			$n_u=5$	$n_r=95$	$h=3$	$k=100$		
Ξ	P	W_u low	W_u high	W_r low	W_r high	Q	C_ξ	C_α
0	0	4.00	4.00	0.00	0.00	0.0000	0.0000	1.0101
0	1	3.06	0.05	0.00	0.00	0.0003	0.0000	1.0098
0	2	0.05	0.00	0.00	0.00	0.0300	0.0000	0.9900
1	0	4.00	4.00	0.47	0.00	0.0000	0.0100	0.0201
1	1	3.06	0.05	0.91	0.01	0.0003	0.0099	0.0198
1	2	0.05	0.00	2.33	0.01	0.0300	0.0000	0.0000
2	0	4.00	4.00	47.50	0.47	0.0000	0.0002	0.0003
2	1	3.06	0.05	92.03	2.34	0.0003	0.0000	0.0000
3	0	4.00	4.00	95.00	95.00	0.0000	0.0000	0.0000

			$n_u=5$	$n_r=95$	$h=2$	$k=1000$		
Ξ	P	W_u low	W_u high	W_r low	W_r high	Q	C_ξ	C_α
0	0	4.00	4.00	0.00	0.00	0.0000	0.0000	1.0010
0	1	3.06	0.00	0.00	0.00	0.0020	0.0000	0.9990
1	0	4.00	4.00	47.50	0.05	0.0000	0.0010	0.0020
1	1	3.06	0.00	92.03	0.24	0.0020	0.0000	0.0000
2	0	4.00	4.00	95.00	95.00	0.0000	0.0000	0.0000

7. Extensions to Sequential Readers

In some uses of B^*-trees to support indexes, the nodes of the tree can also belong to sequential data structures. A common situation would be that of readers performing sequential scans through a sequence of nodes at the same level in

Table 1 (continued)

		$n_u=30$		$n_r=70$	$h=5$	$k=10$		
Ξ	P	W_u low	W_u high	W_r low	W_r high	Q	C_ξ	C_α
0	0	29.00	29.00	0.00	0.00	0.0000	0.0000	1.1110
0	1	28.00	13.86	0.00	0.00	0.0005	0.0000	1.1106
0	2	13.45	0.97	0.00	0.00	0.0050	0.0000	1.1070
0	3	1.73	0.05	0.00	0.00	0.0500	0.0000	1.0800
0	4	0.16	0.00	0.00	0.00	0.5000	0.0000	0.9000
1	0	29.00	29.00	0.03	0.00	0.0000	0.1000	0.2110
1	1	28.00	13.86	0.05	0.01	0.0005	0.0999	0.2106
1	2	13.45	0.97	0.44	0.01	0.0050	0.0990	0.2070
1	3	1.73	0.05	0.74	0.01	0.0500	0.0900	0.1800
1	4	0.16	0.00	0.78	0.01	0.5000	0.0000	0.0000
2	0	29.00	29.00	0.29	0.01	0.0000	0.0200	0.0310
2	1	28.00	13.86	0.58	0.12	0.0005	0.0198	0.0306
2	2	13.45	0.97	4.79	0.22	0.0050	0.0180	0.0270
2	3	1.73	0.05	8.18	0.23	0.0500	0.0000	0.0000
3	0	29.00	29.00	3.18	0.16	0.0000	0.0030	0.0040
3	1	28.00	13.86	6.36	2.56	0.0005	0.0027	0.0036
3	2	13.45	0.97	52.66	4.61	0.0050	0.0000	0.0000
4	0	29.00	29.00	35.00	3.33	0.0000	0.0004	0.0004
4	1	28.00	13.86	70.00	53.80	0.0005	0.0000	0.0000
5	0	29.00	29.00	70.00	70.00	0.0000	0.0000	0.0000

		$n_u=30$		$n_r=70$	$h=3$	$k=100$		
Ξ	P	W_u low	W_u high	W_r low	W_r high	Q	C_ξ	C_α
0	0	29.00	29.00	0.00	0.00	0.0000	0.0000	1.0101
0	1	28.00	2.07	0.00	0.00	0.0003	0.0000	1.0098
0	2	2.06	0.01	0.00	0.00	0.0300	0.0000	0.9900
1	0	29.00	29.00	0.35	0.00	0.0000	0.0100	0.0201
1	1	28.00	2.07	0.69	0.05	0.0003	0.0099	0.0198
1	2	2.06	0.01	9.68	0.05	0.0300	0.0000	0.0000
2	0	29.00	29.00	35.00	0.35	0.0000	0.0002	0.0003
2	1	28.00	2.07	70.00	9.73	0.0003	0.0000	0.0000
3	0	29.00	29.00	70.00	70.00	0.0000	0.0000	0.0000

		$n_u=30$		$n_r=70$	$h=2$	$k=1000$		
Ξ	P	W_u low	W_u high	W_r low	W_r high	Q	C_ξ	C_α
0	0	29.00	29.00	0.00	0.00	0.0000	0.0000	1.0010
0	1	28.00	0.22	0.00	0.00	0.0020	0.0000	0.9990
1	0	29.00	29.00	35.00	0.03	0.0000	0.0010	0.0020
1	1	28.00	0.22	70.00	1.04	0.0020	0.0000	0.0000
2	0	29.00	29.00	70.00	70.00	0.0000	0.0000	0.0000

the tree. The ideas developed in the protocols presented in Section 4 can also be adapted to allow sequential read accesses concurrently with random read and update accesses. We will briefly discuss how this can be done for one case. The reader will then quickly see how these concepts could be used in other situations.

Consider the very frequent case of readers accessing the nodes of the tree following a path from the root down, with the additional freedom of allowing a left to right scan of some adjacent nodes to be made at the same level of the tree. Thus, a reader could, for example, start at the root, follow a downward path to a node, move right on adjacent nodes at the same level to another node then continue down the tree.

A similar analysis to that done in Lemmas 1, 2, and 3 would show that no deadlocks could occur by allowing such traversals, except for one case. This occurs when an updater, having locked with ξ-locks all nodes that will be modified, begins to perform the actual modifications and discovers that an overflow or underflow exists and wishes to scan the left brother q of a node p (on which it holds a ξ-lock) to see if there is room to accommodate for this overflow or underflow. Instead of acquiring a ξ-lock on q, as indicated in Step 7 of the protocol for the generalized solution, the updating process would then have to:

1) convert the ξ-lock on p to an α-lock;
2) place a ξ-lock on q;
3) Examine node q. If it can be used to accommodate the overflow or underflow, convert the α-lock on p to a ξ-lock and perform the operation; otherwise, release the ξ-lock on q and convert the α-lock on p to a ξ-lock; {Examination of the right brother would not require any changes to our original protocol}.

It can be shown that with this modification, the new protocols are still deadlock free.

Other modes of traversals by readers can also be accommodated using the ideas presented here.

8. Conclusions and Implementation Considerations

In this paper we have examined several solutions to the problem of concurrency in indexes implemented as B^*-trees. A generalized solution has been presented which allows tuning the access to these structures to optimize concurrency of operations. Furthermore, this solution has been shown to be deadlock free. This shows that with proper locking techniques, B^*-trees support easy and highly concurrent access to indexes.

When implementing B^*-trees with the provision for parallel operations some structuring concept should be chosen, which allows to consider a B^*-tree together with the operations and their lockprotocols as one conceptual unit. Several such concepts, most of them related to the Simula classes, have been offered in the literature, but they usually ignore the problem of parallel operations, assuming that these units will be used by one sequential process only. This automatically results in a serial application of operations.

To deal with parallel (from an external point of view) operations the concept of a Monitor has been introduced [5, 7]. Monitors, however, deal with external parallelism essentially by enforcing internally a serialization of operations. Unfortunately this means that a Monitor would cancel exactly the effect we are trying to achieve.

In the Operating Systems Project BSM at the Technical University in Munich a structuring concept called *Manager* was developed [8, 9]. *Managers* deal with parallel external operations but allow in a carefully controlled way also internal parallelism of operations. It seems that Managers would be a suitable structuring concept for implementing the solutions presented in this paper.

Acknowledgements. Mike Blasgen participated in the initial discussions that lead to the generalized solution presented here. His ideas about locking in B^*-trees as implemented in System R [1] and his cooperation with the authors are sincerely appreciated.

References

1. Astrahan, M.M., et al.: System R: Relational approach to database management. ACM Transactions on Database Systems **1**, 97–137 (1976)
2. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indexes. Acta Informat. **1**, 173–189 (1972)
3. Bayer, R., Metzger, J.: On the Encypherment of Search Trees and Random Access Files. ACM Transactions on Database Systems **1**, 37–52 (1976)
4. Bayer, R., Unterauer, K.: Prefix *B*-trees. IBM Research Report RJ 1796, San Jose, Calif., 1976. ACM Transactions on Database Systems **2**, 11–26 (1977)
5. Brinch Hansen, P.: A programming methodology for operating system design. IFIP Congress 1974, Stockholm, pp. 394–397. Amsterdam: North Holland 1974
6. Held, G., Stonebraker, M.: *B*-trees reexamined. ERL, College of Engineering, Univ. of California, Berkeley, Calif., Memo. #ERL-M528, July 2, 1975
7. Hoare, C.A.R.: Monitors: An operating system structuring concept. Comm. ACM **17**, 549–557 (1974)
8. Jammel, A., Stiegler, H.: Verwalter, eine Methode der rekursiven Prozeßzerlegung. Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften, LRZ Internschrift 7604/1, München, 1976
9. Jammel, A., Stiegler, H.: Managers versus Monitors. Submitted to: IFIP Congress (1977)
10. Knuth, D.E.: The art of computer programming, Vol. 3. Sorting and searching. Reading, Mass.: Addison-Wesley 1972
11. Metzger, J.K.: Managing simultaneous operations in large ordered indexes. Technische Universität München, Institut für Informatik, TUM-Math. Report, 1975
12. Schkolnick, M.: Initial specifications for DFMAS. Unpublished document, May 1975
13. Wedekind, H.: On the selection of access paths in a data base system (J.W. Klimbie, K.L. Koffeman, eds.), Data base management, pp. 385–397. Amsterdam: North-Holland 1974

Received June 10, 1976