CIS-TR-80-6

A Hybrid Solution
for Concurrent Operations on B-trees

Carla Schlatter Ellis

March 1980

Department of Computer and Information Science
University of Oregon
Eugene, Oregon 97403

Abstract

B-trees are useful for supporting large ordered indexes in data base systems. Several solutions have recently been proposed to deal with the problem of allowing concurrent operations in data structures related to B-trees. In this paper, the strengths and weaknesses of two of these solutions are described. A combined approach is presented that can be tuned to satisfy specific performance requirements. Informal arguments for the correctness of this new solution are given.

1. Introduction

B-trees are useful data structures for supporting large ordered indexes in data base systems. Providing concurrent access to these structures is necessary in order to give multiple users acceptable response times to their requests. However, allowing concurrent operations complicates the problem of ensuring the integrity of the data and introduces the possibility of deadlock.

A number of solutions have been proposed to deal with this problem of concurrency in variants of B-trees [Bayer & Schkolnick 77, Miller & Snyder 78, Ellis 78, Lehman & Yao 79, Kwong & Wood 79]. These solutions share certain characteristics: First, they each use a locking mechanism to restrict parallel access to critical portions of the tree. In order to achieve a high degree of concurrency, recent approaches attempt to limit the number of nodes that must be locked at the same time by one process and the amount of time each node remains locked. Another similarity among solutions is that modifications of the original B-tree structure are introduced to provide alternate paths by which a process can reach information being moved by another process.

In this paper, we focus on two of these previous solutions, namely [Lehman and Yao 79] and [Ellis 78]. We examine the strengths and weaknesses of each approach and present a solution which combines techniques from both. The objective of the proposed system is to allow some degree of tuning to satisfy specific performance requirements. Informal arguments are given for the correctness of this new solution.

2. Two Previous Approaches

The data structures used in the two solutions considered here are based on the B*-tree. A B*-tree of degree k is defined to be a finite set of nodes (often referred to as pages) which is either a single leaf node or consists of a root with from 2 to k sons. Each non-leaf node except the root has from $\lceil k/2 \rceil$ to k sons. All of the actual data is stored in the leaf nodes and every path from root to leaf is the same length. A non-leaf node contains pointers to its sons and label fields that serve to direct searching processes down the appropriate pointer. Figure 1 gives an example of a B*-tree. Algorithms to insert or delete information must restructure the tree whenever the operation would produce a node with more than k or fewer than $\lceil k/2 \rceil$ sons. Thus, attempting to insert a new entry into an already full node causes that node to split in two with the new entry

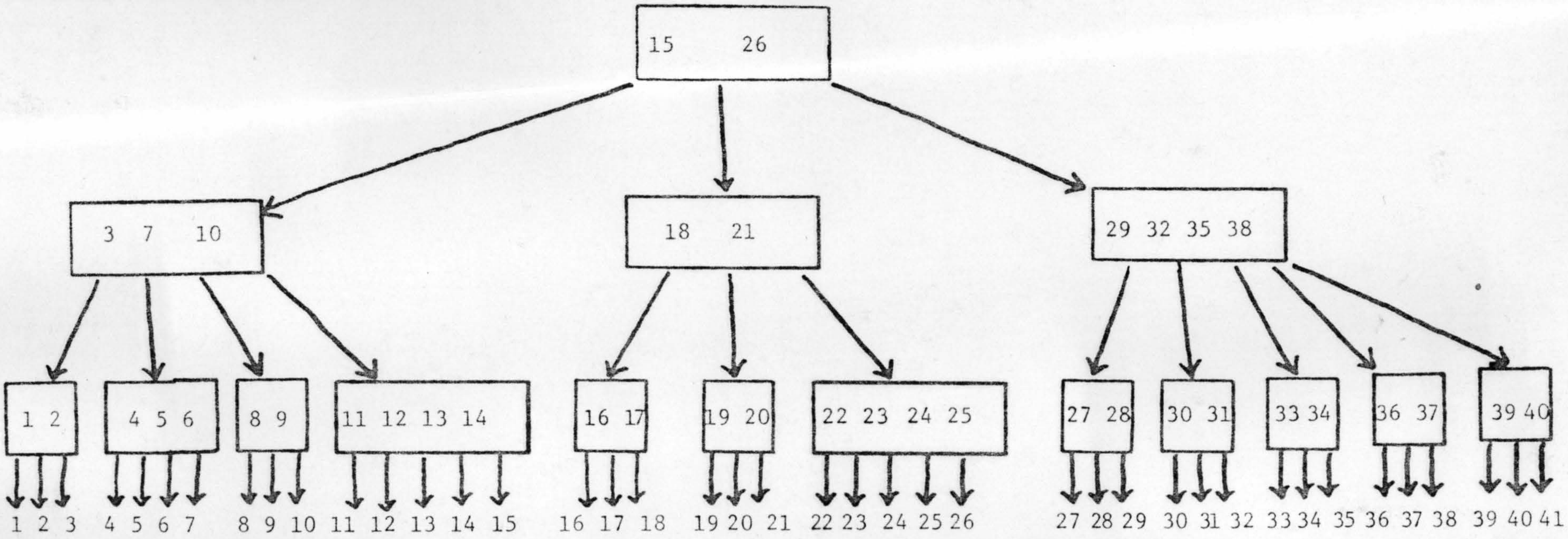

Figure 1 B^*-Tree of degree k=5

inserted into the appropriate half and an entry for its newly created brother inserted into the father node. A node is referred to as safe for insertion if it has fewer than k sons and safe for deletion if it has more than $\lceil k/2 \rceil$ sons.

In [Ellis 78], we describe two solutions for concurrent search and insertion in 2-3 trees. A 2-3 tree is the smallest special case of B*-trees and is usually used for relatively small amounts of data (i.e. the entire tree can reside in main memory). The second of the two solutions (the 2-3 Pipeline solution) is the one of interest here. This solution uses the three types of locks proposed by [Bayer & Schkolnick 77]: a read lock or ρ-lock, an exclusive lock or ξ-lock, and a writer exclusion lock or α-lock. Figure 2 shows the compatibility relation for different processes that these locks satisfy. An edge between any two nodes in this graph means that two different processes may simultaneously hold these locks on the same node of the tree. An additional type of lock, a special case of α-lock, is used to sequence leaf insertions and prevent deadlock. It is compatible with the other form of α-lock.

The 2-3 Tree structure is modified by providing each non-leaf node with an overflow area that makes inserted information accessible to searching processes while the restructuring operations necessitated by the insertion are still in progress. A single additional pair of pointer and label fields is sufficient above the maximum non-leaf level (referred to as the father of leaf level). At the father of leaf level, nodes head linked lists of leaves rather than having a fixed number of sons. This change prevents deadlock between a restructuring process and a process attempting to insert a new leaf by allowing the inserter to first put the new leaf in place and then to release all locks it holds. The final modification in the data structure deals with the possibility that the path followed by a writer process during its search phase may not be the correct path for restructuring because of intervening restructuring operations. Thus each non-leaf node includes a pointer to its current father node.

The basic strategy of this solution is to allow inserting processes to search down the tree for the appropriate place of insertion using ρ-locks as in readers, insert a new leaf, and then restructure bottom up using α-locks to enforce following the leader. Thus multiple writers may be searching and restructuring along the same path. In addition, a technique suggested in [Lamport 77] permits a node to be shared by multiple readers and a writer concurrently adding a new pointer-label pair. The basic idea is that when a reader process and a writer process

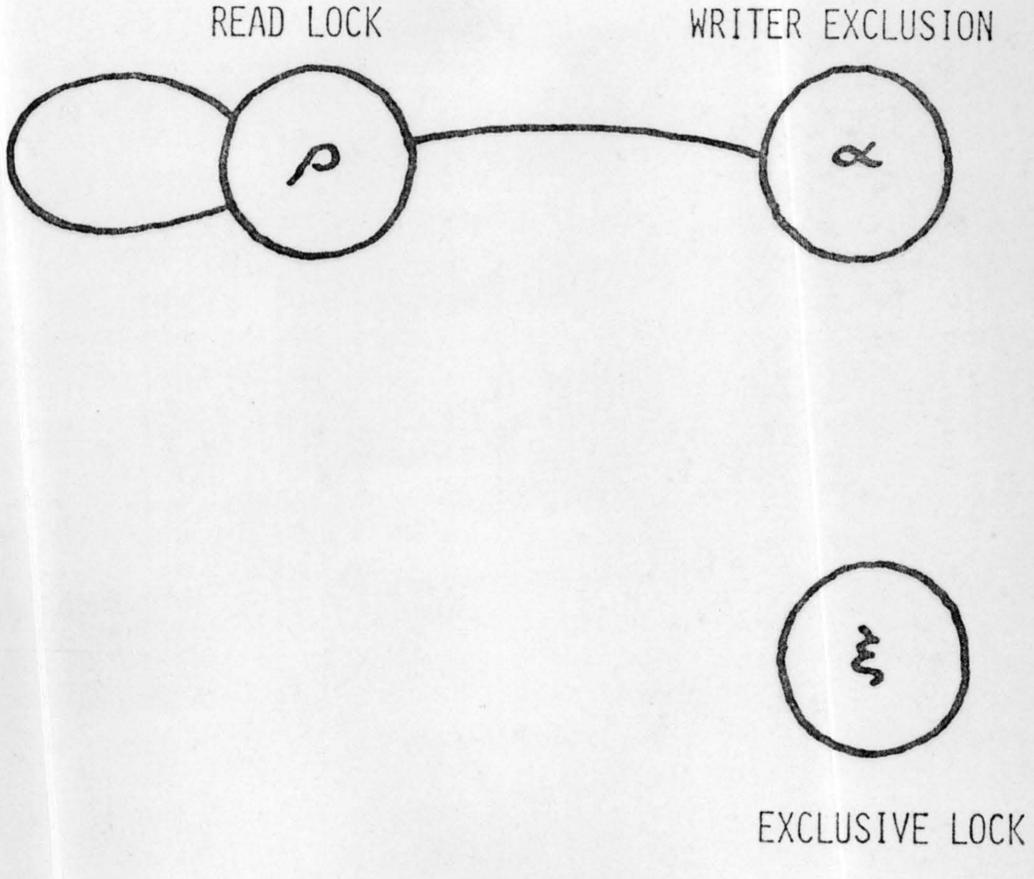

Figure 2 Compatibility graph for locks.

access the same node, they move in opposite directions. ξ-locks are needed only when entries are deleted from a node (e.g. when splitting causes entries to be transferred to a newly created brother).

The 2-3 Pipeline solution achieves a high degree of concurrency. The major drawback for database applications is the underlying assumption that the entire data structure is resident in main memory. Maintenance of the father pointers and scanning of the linked list of leaves would involve a considerable number of secondary storage accesses if this approach were applied to larger paged trees.

The second solution which serves as a basis for this work is presented in [Lehman and Yao 79]. The data structure is a modified B*-tree called a B+-tree. Each non-leaf node is augmented by an additional label field which holds an upper bound on the values that may be stored in the subtree rooted at that node and by an additional pointer to the brother on its right. Thus all nodes at the same level form a linked list. An example of a B+-tree is given in figure 3. When a restructuring process splits a node, the newly created brother is inserted directly to the right of the original node so that the entries which have been moved are reachable through the brother pointer. With this convention, the nodes belonging to a writer's restructuring path are accessible either immediately or via brother links from nodes on its search path. Therefore a stack recording the rightmost node scanned at each level during the search phase supplies the leftmost candidate for restructuring at each level. Father pointers are not necessary with this approach.

Only one type of lock is needed in this solution. The lock serves the same purpose as the α-lock in the 2-3 Pipeline solution (i.e. excluding other updating writers but not readers or searching writers). Readers do not use locks of any kind. The simple locking scheme is possible because each process operates on its own private copy of the disk page. Several copies of a disk page may reside in main memory simultaneously with at most one process holding a lock on the page and modifying its copy. Writing the modified copy back onto the disk page appears to other processes as an instantaneous change in the data structure.

The strong point of this solution lies in the choice of the B+-tree as the data structure. The brother links provide a natural method of recovery from the restructuring actions of other processes. The simple locking scheme demands relatively little overhead for requesting and releasing locks. The major weakness is the requirement for private copies.

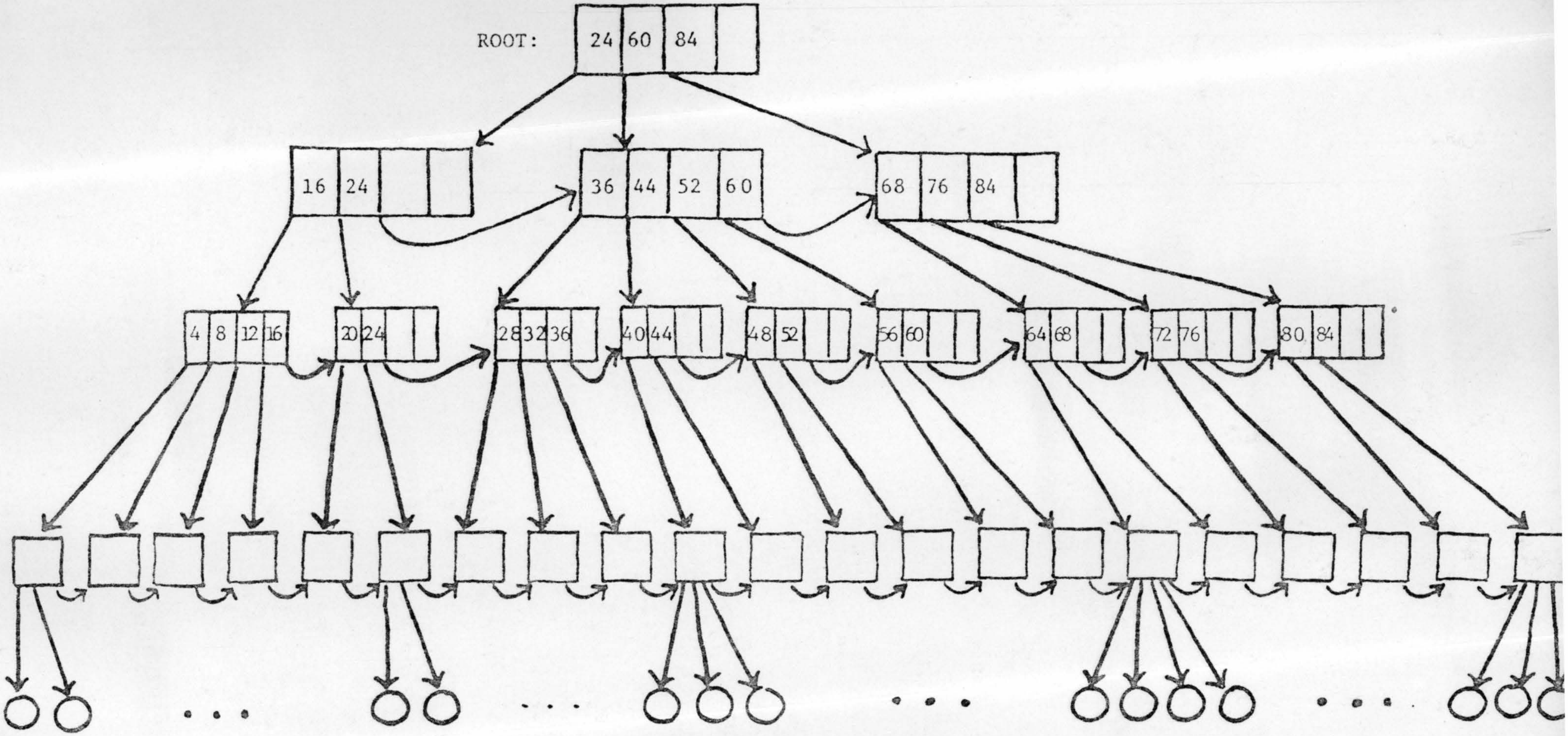

Figure 3 B^+-Tree

Compared to a solution in which sharing of main memory pages is allowed this restriction should lead to a higher number of secondary storage transfers and a poorer main storage utilization. Consider a B+-tree of level 4 accessed concurrently by n reading processes. With this approach, we can expect n/4 copies of the root page to exist simultaneously in main memory. In addition to the space consumed by versions of the same page, each copy involves a transfer from disk. The advantages in terms of disk traffic of a policy in which the root node and nodes at level 1 remain in main memory and are shared have been discussed in [Knuth 73] for sequential access and in [Baer & Anstead 78] for concurrent operations. In the next section, we propose a system that can implement this policy in a highly concurrent environment.

3. A Hybrid Solution

The basic idea is to combine the two previous approaches so that either solution may be emphasized depending on the setting of a parameter. This parameter specifies the level of the tree at which a transition is made between shared pages and private copies. The concept is illustrated in Figure 4. Processes operate on the shared nodes using a locking scheme related to the one in [Ellis 78]. The compatibility graph of figure 2 is pertinent to this design. The technique of reading and writing in opposite directions is applied to facilitate sharing. Below the designated level, processes use the technique of [Lehman & Yao 79]. The two previous solutions are similar with respect to the bottom-up locking protocol for inserters and the α-lock is used throughout the tree. By contrast, ξ-locks and ρ-locks are needed only for shared pages. This arrangement is particularly convenient since most of the splitting operations (which require ξ-locks on shared pages) occur in the bottom levels which typically will not be shared. The B+-tree is adopted as the data structure.

The major problem to be solved is to develop the procedure for making the transition between the two approaches. Without concurrent restructuring operations, a process can simply keep a counter of levels through which it has passed in order to detect when it reaches the transition point. The difficulty arises with the possibility that the root node may split and a new root be created. The level of each node then increases by one and the set of nodes previously at the transition level change from sharable to private access. Thus a process may have obsolete information indicating that a node resides in shared memory and must be able to recover. An important question is how to consistently handle α-locks for both shared and private pages so that a change in the level of a page

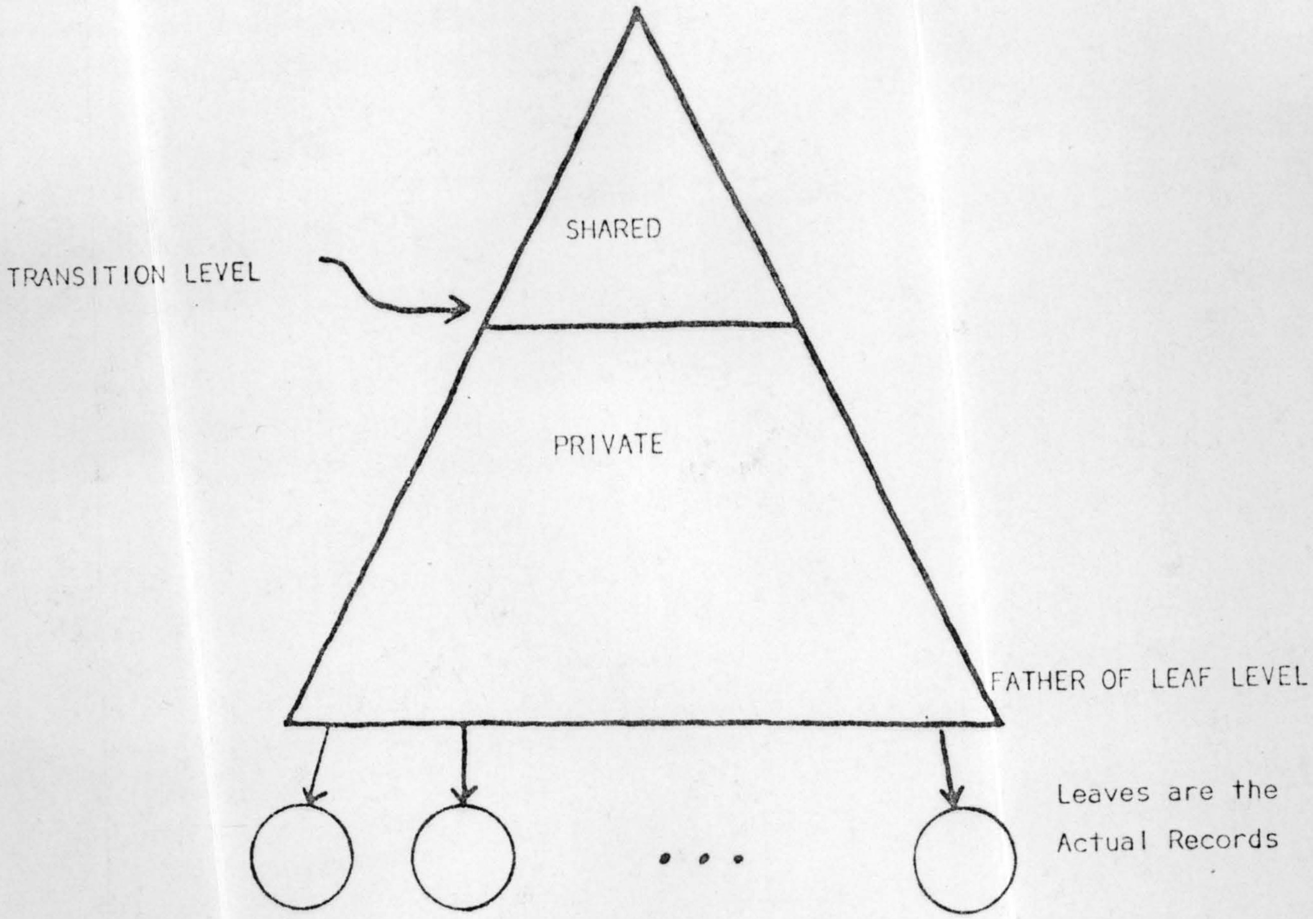

Figure 4 Tree Configuration for Hybrid Solution

has no effect on its lock status. A method is also needed for determining when it is possible to replace a formally sharable page that some active processes gained access to before the root split. In the following section, we describe a monitor system for memory management and concurrency control that deals with these issues.

3.1 The System Model

The official version of the entire B+-tree is stored on some secondary memory device. Copies of selected pages (not necessarily of the up to date version) reside in either shared or private areas of main storage. In order to operate upon a node of the tree, a process must either find it occupying a block of shared memory or read it from secondary storage into a private block. Thus we need a set of memory management functions to locate sharable pages, supervise the replacement of pages, and initiate transfers between disk and main memory.

This model utilizes an associative table of sharable pages to determine which memory block contains a specified node. In addition to fields for the page and block addresses, each entry in the table contains a field recording the present level of the node and a use bit that help decide when a page may be removed.

The following functions are provided by the system:

FIND (node, index) searches the associative table for the specified node. If successful, the use bit is set and the index of the table entry is returned. ADDR[index] locates the main memory block desired.

DEPART (index, node) writes the contents of the block starting at ADDR[index] onto the disk page indicated by node and resets the use bit.

CREATE (node, block#, level) creates an entry in the table for a new node and makes the specified block sharable. If the level parameter indicates that this is a new root, the header is reassigned to point to the new root and the level of every other entry is incremented thereby generating a set of candidates for page replacement.

block# ← GET (node) reads the contents of the disk page pointed to by node into the memory block.

PUT (block#, node) writes the contents of the given memory block onto the disk page indicated by node.

The concurrency control functions of the system allow processes to request and release locks. Since ρ-locks and ξ-locks are used only for shared pages, the associative table appears to be the appropriate place to store the status of a node with regard to these locks. The field giving the number of ρ-locks held also influences the page replacement decision. Finally, each sharable page is associated with a waiting queue for processes attempting lock requests which are incompatible with the current lock status. The situation with regard to α-locks is more interesting. It is desirable that the data structures which represent α-locking status be in main memory and not distinguish between sharable and private nodes. Since the number of pages in the B+-tree should be very large compared to the number of α-locks held by concurrently active writers, a list recording only the α-locked nodes is a suitable implementation. A waiting queue is associated with each entry in the list for processes blocked on attempting to α-lock that node.

The following indivisible locking operations are defined:

ALPHA-LOCK (node) searches the list of locked nodes. If the node is found, the requesting process is blocked; otherwise an entry is inserted for the node.

RELEASE-ALPHA (node) finds the entry in the list for the node. If no processes are waiting in the associated queue, the entry is deleted; otherwise the first process is awakened.

RHO-LOCK (node, index) serves the dual purpose of finding the page and locking it. The associative table is searched and the index of the table entry is returned. In addition the lock status is checked and updated (ρ-lock count is incremented) after a possible wait.

RELEASE-RHO (index) decrements the ρ-lock count. If the count goes to zero and a process is waiting to ξ-lock, it is awakened.

XI-LOCK (index) and

RELEASE-RHO (index) similarly manipulate the lock status and wait queues of an entry in the associative table in compliance with the compatibility graph of figure 2.

3.2 The Algorithms for Readers and Inserters

The procedures executed by searching and inserting processes are given in Appendix 1 for trees containing at least one non-leaf node. In order to simplify the presentation of the algorithms in this section, we further assume the tree contains both sharable and private levels (i.e. for the specified transition level, l, and the father of leaf level, h, $0<l<h$).

Readers begin at the root and ρ-lock each sharable node on their path before scanning it for the desired value. Upon leaving a sharable node, the process releases its ρ-lock. Scanning a node is accomplished by reading the pointer and label fields sequentially from left to right and comparing the desired value, v, with each label field, label[i], until $v \le$label[i] or all of the labels in this node have been read (i=# sons) and found to be less than v. In the first case, the next node to be searched is the son pointed to by son[i]. In the second case, this node must have recently split and the brother link should be followed. Figure 5 shows the fields of a non-leaf node. For each node on the path below the transition level, the node must be copied from disk before it is scanned. Eventually the searching process reaches the father of leaf node that should point to the record associated with v.

Inserting a new leaf into the tree first requires a search for the proper place to insert. An inserter process performs essentially the same steps as a reader during this phase except that it also records on a stack the rightmost node scanned at each level. When it reaches the father of leaf level, the process α-locks the node, reads it again from the disk in case a restructuring operation occurred before the lock request was granted, and if necessary follows brother pointers (α-locking as it moves right) to the appropriate father of leaf node. If the resulting node is safe, the new entry is inserted, the disk page updated, and the α-lock released; otherwise, a splitting operation is performed, the stack popped, and the newly created non-leaf node inserted into the father of the node that split. After the next node in the restructuring path is correctly identified and α-locked, the α-lock on the split node is released. The restructuring operations proceed bottom up until a safe node is encountered. The details depend on whether the node being modified is private or sharable. The inserting process maintains a level counter to help it determine which actions are appropriate. While the value of the level counter is greater than the transition level, the process is guaranteed to be operating upon private nodes. After that, an attempt

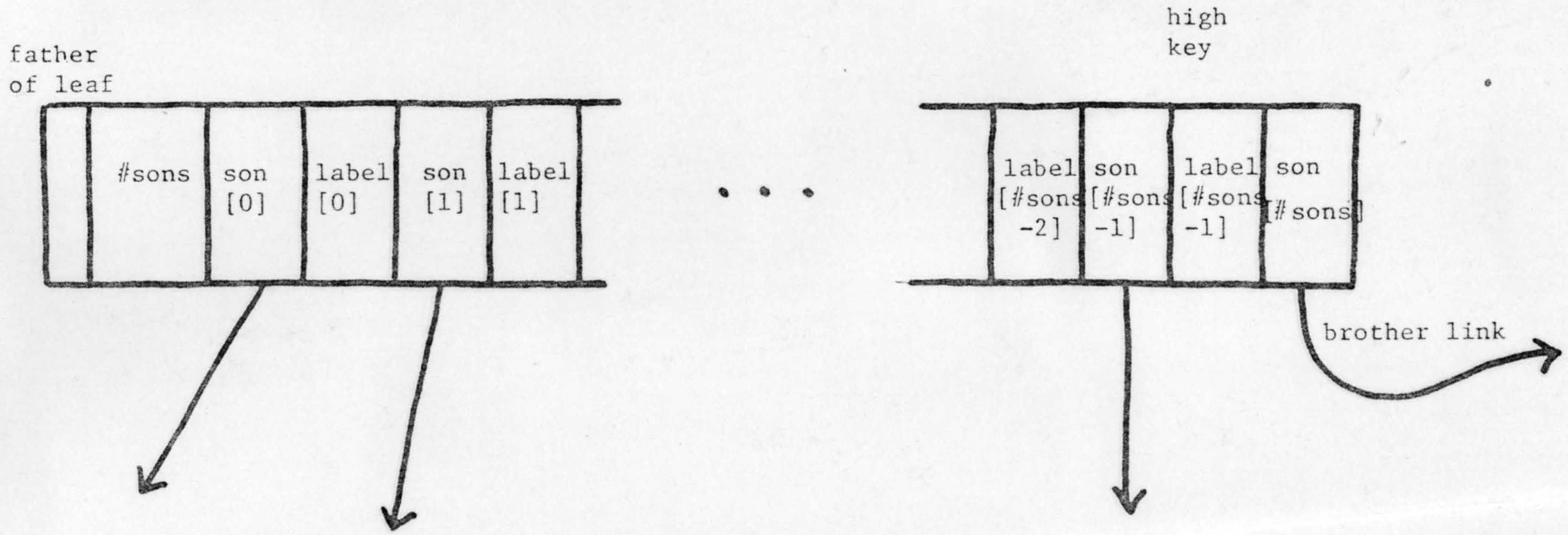

Figure 5 Individual node of the B$^+$-Tree.

is made to FIND each desired node in shared storage.

To split a node, the following steps are taken:

1) Construct a new non-leaf page, B, with entries copied from the right half of the original unsafe node, A. If A is sharable, it must be ξ-locked while the brother pointer and #sons field are updated. Insert the new entry into the appropriate node of the pair. If A is sharable, an entry for B must be made in the associative page table by calling CREATE.

2) Put B onto disk.

3) Write A onto disk by calling DEPART if A is sharable or PUT if it is private.

4) Pop the stack and α-lock that node. Using FIND for sharable nodes and GET for private nodes, move right until the actual father of A is identified and α-locked for the insertion of A's new brother.

The steps necessary to insert an entry into a node depend on whether the entry is for a leaf or a non-leaf node. However the essential property is that pointers and labels are moved right to make room for the new entry by starting with the brother pointer and proceeding right to left, moving label[i] before son[i], until the proper place to insert is reached.

3.3 Correctness Arguments

In this section we sketch a correctness proof of these algorithms. There are four parts to the discussion: proving freedom from deadlock, appealing to the proof in [Lehman & Yao 79] for the correctness of operations on the private portion of the tree, proving correctness for the shared nodes, and finally showing that the transition is made properly.

α-locks are placed by the inserter according to the well-ordering on nodes of a B+-tree. An inserter holds only one ξ-lock at a time and other inserters are already excluded from that node by the processes's single α-lock. The process releases the ξ-lock before making any further requests. Thus the deadlock situation in which two processes each request an incompatible lock on a node already locked by the other does not arise here.

If the designated transition level is set at 0, our solution reduces to that of [Lehman & Yao 79] where private copies are made at all levels. The arguments for the integrity of the tree structure on disk and for the non-interference among processes given there hold for the private portion of our tree.

Modifications of the disk version of sharable nodes are performed in essentially the same way as for private nodes. Thus, if we can be assured that the contents of the shared page are valid at the time of the DEPART operation, the arguments for structural integrity still hold. Since inserters must be granted an α-lock on a node before FINDing it during the restructuring phase and DEPART is done prior to releasing the α-lock, only one process may be modifying the node.

The interaction of processes within the shared nodes is more interesting. We must show that a reader or searching writer eventually reaches the proper father of leaf node in spite of modifications concurrently being made to the fields of the node being scanned. This demands that the pointer followed as a result of scanning lead either to the correct node or to one of its left brothers. The technique of reading and writing in opposite directions [Lamport 77] is used for inserting an entry into a node. Lamport proved that when the fields of a data item are written from right to left, a read performed left to right obtains values such that for each field, the value immediately to the right is from the same or later version of the data item. The i + 1st version of a label field is less than or equal to the ith version and the i + 1st version of a pointer field leads either to the node pointed to in the ith version or its (possibly new) left brother. Suppose that the correct next node to be scanned is pointed to by the ith version of son[j] and that the reader process meets the writer between reading son[j] and label[j] (i.e. it sees the ith version of the pointer and the i + 1st version of the associated label). Then the following cases should be considered:

a. The i + 1st version of label[j] equals the ith version of label[j-1]. The reader has already encountered this value and passed it because the desired key is greater. Therefore, the jth entry is rejected and the reader follows the i + 1st version of son[j+1] based on the value of label[j+1].

b. The i + 1st version of label[j] is less than the ith version but greater than label[j-1]. If the node being scanned is a father of leaf node, the writer is attempting to insert a new leaf to the left of son[j]. With regard to a search for the new leaf value, this pointer-label combination is inconsistent, but acceptable. The new label[j] causes our reader to reject the

the jth entry in item a. If the node being scanned is not a father of leaf node, the old son[j] has recently split and the writer is inserting the newly created brother as son[j+1].

A decision based on the ith version of label[j] is equivalent to scanning a private copy read from disk prior to the write of the i + 1st version. Brother links provide recovery if needed as shown by Lehman and Yao. Finally, ξ-locks are required when a writer is removing entries from a node so that no inconsistent information is visible to readers.

The correctness of the transition from one approach to the other depends on an important characteristic of B-tree variants; namely that once a node is created at a particular level in the tree, its level can never decrease. Thus a node that originally resides in shared storage can become a private node, but not vice versa. A process counting levels during either the top-down search or the bottom-up restructuring might make the mistake of assuming that a certain node is sharable. If the node is no longer sharable, memory management will fail to find the page and we proceed with a private copy. On the downward path, detecting the first private node signifies that the transition level has been passed.

4. Summary and Conclusions

In this paper we have combined two previous solutions in an attempt to overcome some of their shortcomings. Our solution depends upon a parameter that specifies the level at which a transition is made from a locking approach on shared pages to operating upon private copies. The performance measures that should be affected by the selection of this parameter value are the number of secondary disk transfers, the main memory requirements with a fixed number of concurrent processes, and the amount of locking overhead. Characteristics of the processing environment that must be taken into account are the degree of the tree, k, and the numbers of concurrent readers and inserters. Simulation experiments investigating the tradeoffs will be reported in a subsequent paper. Probablistic analyses will also be presented.

The subject of deletion has been omitted because in many applications it is performed relatively infrequently and the strategy of allowing deletion from unsafe father of leaf nodes without restructuring (i.e. allowing nodes with less than $\lceil k/2 \rceil$ sons) is acceptable. It is possible to extend this solution to allow concurrent restructuring.

5. Bibliography

Baer, J-L. and S. Anstead, "Concurrent Accesses of B*-trees in a Paging Environment", Proc. of International Conf. on Parallel Processing, 1978.

Bayer, R. and M. Schkolnick, "Concurrency of Operations on B-trees", ACTA INFORMATICA 9, 1977, pp. 1-22.

Ellis, Carla, "Concurrent Search and Insertion in 2-3 Trees", TR-78-05-01, University of Washington, 1978, to appear in ACTA INFORMATICA.

Knuth, D.E., *The Art of Computer Programming, Vol. 3, Sorting and Searching and Searching*, Addison-Wesley, 1973, pp. 471-479.

Kwong, Y.S. and D. Wood, "Concurrency in B-trees, S-trees, and T-trees", TR-79-CS-17, McMaster University, 1979.

Lamport, L., "Concurrent Reading and Writing", CACM, Vol. 20, No. 11, Nov. 1977, pp. 806-811.

Lehman, P. and S. B. Yao, "Efficient Locking for Concurrent Operations on B-trees", preliminary report 1979.

Miller, R. and L. Snyder, "Multiple Access to B-trees", Proc. Conf. Information Science and Systems, Mar 1978.

READER

```
RHO-LOCK (header,i)
current ← ADDR[i]
RHO-LOCK (current,j)
while j ≠ λ  /*current is sharable*/ do begin
     RELEASE-RHO(i)
     i ← j
     previous ← current
     current ← SCANNODE (value, ADDR[i], brotherlink)
     if previous is father of leaf and not brotherlink
     then begin /*current is leaf*/
          Compare value with labels in ADDR[i]
          to determine success
          RELEASE RHO(i)
          return
     end
     RHO-LOCK (current,j)
end
A ← GET (current)
RELEASE-RHO(i)
while true do begin
     previous ← current
     current ← SCANNODE (value, A, brotherlink)
     if previous is father of leaf and not brotherlink
     then begin
          Compare value with labels in A to
          determine success
          return
     end
     A ← GET (current)
end
```

INSERTER

```
initialize stack                     /*as in reader process
RHO-LOCK (header,i)                  except
PUSH (header)                        stack rightmost node in each level
current ← ADDR[i]                    and at father of leaf level α lock*/
RHO-LOCK (current,j)
while j ≠ λ  & current is not father of leaf do begin
     RELEASE-RHO(i)
     i ← j
     previous ← current
     current ← SCANNODE (value, ADDR[i], brotherlink)
     if not brotherlink then
     push (previous)
     RHO-LOCK (current,j)
end
if current is father of leaf then begin
     release RHO(j)
     ALPHA-LOCK (current)
     RELEASE-RHO(i)
     FIND (current,i)
     if i = λ then begin
          A ← GET (current)
          MOVERIGHT
          Compare
     end
     else begin
          SHARABLE_MOVERIGHT
          Compare
     end
end
```

```
else begin  /* NONSHARED */
    A ← GET (current)
    RELEASE-RHO(i)
    while current is not father of leaf do
        previous ← current
        current ← SCANNODE (value,A,brotherlink)
        A ← GET (current)
        if not brotherlink then
        push (previous)
    end
    /*current is father of leaf*/
    ALPHA-LOCK (current)
    A ← GET (current)
    MOVERIGHT
    Compare
end
/*restructuring phase*/
level ← father of leaf level
newnode ← ptr to disk page alloc to record assoc with value
while level > transition + 1 & current is not safe do
begin
    SPLITPRIVATENODE
    A ← GET (current)
    moveright
    release ALPHA (previous)
end
while current is not safe  /*level believed to be =
                                        transition + 1*/
do begin
    SPLITPRIVATENODE
    FIND (current,i)
    while $i \neq \lambda$ do /*within sharable portion*/
    begin
        t ← SCANNODE (value, ADDR[i], brotherlink)
        if brotherlink then begin
            ALPHA-LOCK(t)
            RELEASE-ALPHA (current)
            current ← t
            FIND (current,i)
        end
```

```
            else begin /*actual father*/
                release-ALPHA (previous)
                if current is safe then begin
                    INSERT (ADDR[i], newnode, value)
                    DEPART (i,current)
                    RELEASE ALPHA (current)
                    return
                end
                if current is root then begin
                    SPLITSHAREDNODE
                    root ← alloc new disk page
                    A ← construct page with
                    (previous, value, newnode, B.label[#sons-1],λ )

                    PUT (A,root)
                    CREATE (root,A,newroot)
                    RELEASE ALPHA (previous)
                    RELEASE ALPHA (current)
                    return
                end
                SPLITSHAREDNODE
            end /*actual father*/
        end /* i ≠ λ */
        A ← GET (current)
        moveright
        release ALPHA (previous)
    end /*while current not safe*/
    /*current is safe*/
    INSERT (A,newnode,value)
    PUT (A,current)
    release ALPHA (current)
```

```
procedure SHARABLE MOVERIGHT
    t ← SCANNODE (value, ADDR[i], brotherlink)
    while brotherlink do begin
        ALPHA-LOCK (t)
        RELEASE-ALPHA (current)
        current ← t
        FIND (current,i)
        if $i = \lambda$ then begin
            A ← GET (current)
            moveright
            return
        end
        t ← SCANNODE (value, ADDR[i], brotherlink)
    end

procedure SPLITSHAREDNODE
    begin
        brother ← alloc new disk apge
        B ← copy righthand half of ADDR[i]
        X1-LOCK (i)
            ADDR[i].#sons ← $\lceil k/2 \rceil$ or $\lceil k/2 \rceil -1$
                    depending on where newnode is to be
                    inserted
            ADDR[i].son[#sons] ← brother
        RELEASE-X1 (i)
        if value > ADDR[i].label[#sons-1] then
            INSERT (B,newnode,value)
            else INSERT (ADDR[i],newnode,value)
        CREATE (brother,B,i)
        PUT (B,brother)
        DEPART (i,current)
        newnode ← brother
        previous ← current
        value ← ADDR[i].label[#sons-1]
        current ← pop (stack)
        ALPHA-LOCK (current)
        FIND (current,i)
    end SPLITSHAREDNODE
```

```
function SCANNODE (value: keytype,B: frame#, var brotherlink: boolean): ptr type
     declare local variables: i,link,key
     begin
          i ← 0
          brotherlink ← false
          repeat
               link ← B.son[i]
               if i = B.#sons then begin
                                     brotherlink ← true
                                     return (link)
                                   end
               key ← B.label[i]
               i ← i + 1
          until value ≤ key
          return (link)
     end

procedure INSERT (B: frame#, ptr: ptrtype, value: keytype)
     declare local variable  i
     function PLACETOINSERT: boolean
          if i = 0 then return (true) else
          if value > label [i-1] then return (true)
          else return (false)
     begin
          i ← B.#sons
          B.son[i+1] ← B.son[i]          /*move brotherlink*/
          B.label[i] ← B.label[i-1]      /*move highkey*/
          B.#sons ← B.#sons + 1
          i ← i - 1
          while not PLACETOINSERT do begin
               B.son[i+1] ← B.son[i]
               B.label[i] ← B.label[i-1]
               i ← i-1
          end
          if father of leaf then begin
               B.son[i+1] ← B.son[i]
               B.label[i] ← value
               B.son[i] ← ptr
          end
          else begin
               B.son[i+1] ← ptr
               B.label[i] ← value
          end
     end
```