Efficient Locking for Concurrent Operations on B-Trees

PHILIP L. LEHMAN
Carnegie-Mellon University
and
S. BING YAO
Purdue University

The B-tree and its variants have been found to be highly useful (both theoretically and in practice) for storing large amounts of information, especially on secondary storage devices. We examine the problem of overcoming the inherent difficulty of concurrent operations on such structures, using a practical storage model. A single additional "link" pointer in each node allows a process to easily recover from tree modifications performed by other concurrent processes. Our solution compares favorably with earlier solutions in that the locking scheme is simpler (no read-locks are used) and only a (small) constant number of nodes are locked by any update process at any given time. An informal correctness proof for our system is given.

Key Words and Phrases: database, data structures, B-tree, index organizations, concurrent algorithms, concurrency controls, locking protocols, correctness, consistency, multiway search trees
CR Categories: 3.73, 3.74, 4.32, 4.33, 4.34, 5.24

1. INTRODUCTION

The B-tree [2] and its variants have been widely used in recent years as a data structure for storing large files of information, especially on secondary storage devices [7]. The guaranteed small (average) search, insertion, and deletion time for these structures makes them quite appealing for database applications.

A topic of current interest in database design is the construction of databases that can be manipulated concurrently and correctly by several processes. In this paper, we consider a simple variant of the B-tree (actually of the B*-tree, proposed by Wedekind [15]) especially well suited for use in a concurrent database system.

Methods for concurrent operations on B*-trees have been discussed by Bayer and Schkolnick [3] and others [6, 12, 13]. The solution given in the current paper

This research was supported by the National Science Foundation under Grant MCS76-16604.
Authors' present addresses: P. L. Lehman, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213; S. B. Yao, Department of Computer Science and College of Business and Management, University of Maryland, College Park, MD 20742.

has the advantage that any process for manipulating the tree uses only a small (constant) number of locks at any time. Also, no search through the tree is ever prevented from reading any node (locks only prevent multiple update access). These characteristics do not apply to the previous solution.

A discussion of a similar problem (concurrent binary search trees) has been given by Kung and Lehman [8]. The present paper expands some of the ideas in that paper and applies them to a model in which the concurrent data structure is stored on secondary storage. In addition, a solution for B-trees has the appeal of demonstrated practicality (see, e.g., [1]).

While the analysis is performed for B-trees used as primary indexes, the extension to secondary indexes is straightforward.

2. THE STORAGE MODEL

We consider the database to be stored on some secondary storage device (hereinafter referred to as the "disk"). Many processes are allowed to operate on these data simultaneously. Each process can examine or modify data only by reading those data from the disk into its private primary store (the "memory"). To alter data on the disk, the process must write the data to the disk from its memory.

The disk is partitioned into sections of a fixed size (physical pages; in this paper, these will correspond to logical nodes of the tree). These are the only units that can be read or written by a process. Further, a process is considered to have a fixed amount of primary memory at its disposal, and can therefore only examine a fixed number of pages simultaneously. This primary memory is not shared with other processes.

Finally, a process is allowed to lock and unlock a disk page. This lock gives that process exclusive modification rights to that page; also, a process *must* have a page locked in order to modify that page. Only one process may hold the lock for a given page at any time. Locks *do not* prevent other processes from reading the locked page. (This does not hold for the solutions given, e.g., in [3].)

We assume that some locking discipline is imposed on lock requests, for example, a FIFO discipline or locking administration by a supervisory process.

In the protocol and in the algorithms and proofs given below, we use the following notation. Lowercase symbols (x, t, current, etc.) are used to refer to variables (including pointers) in the primary storage of a process. Uppercase symbols (A, B, C) are used to refer to blocks of primary storage. It is these blocks which are used by the process for reading and writing pages on the disk.

lock(x) denotes the operation of locking the disk page to which x points. If the page is already locked by another process, the operation waits until it is possible to obtain the lock.

unlock(x) similarly denotes the operation of releasing a held lock.

$A \leftarrow get(x)$ denotes the operation of reading into memory block A, the contents of the disk page to which x points.

put(A, x) similarly denotes the operation of writing onto the page to which x points, the contents of memory block A. The procedures must enforce the restriction that a process must hold the lock for that page before performing this operation.

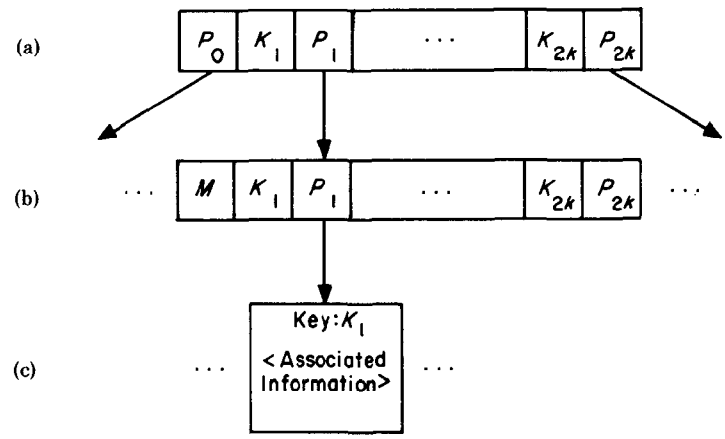

Fig. 1. B*-tree nodes (with no "high key").

To summarize, then, in order to modify a page x, a process must perform essentially the following operations.

```
lock(x);
A ← get(x);                          /* read x into memory from disk*/
modify data in A;
put(A, x);                           /* rewrite memory to disk */
unlock(x);
```

3. THE DATA STRUCTURE

3.1 B*-Trees

In this section we develop the data structure to be used by the concurrent processes. The data structure is a simple variation of the B*-tree described by Wedekind [15] (based on the B-tree defined by Bayer and McCreight [2]).

The definition for a B*-tree is as follows.

3.1.1 *Structure*

(a) Each path from the root to any leaf has the same length, h.
(b) Each node except the root and the leaves has at least $k + 1$ sons. (k is a tree parameter; $2k$ is the maximum number of elements in a node, neglecting the "high key," which is explained below.)
(c) The root is a leaf or has at least two sons.
(d) Each node has at most $2k + 1$ sons.
(e) The keys for all of the data in the B*-tree are stored in the leaf nodes, which also contain pointers to the records of the database. (Each record is associated with a key.) Nonleaf nodes contain pointers and the key values to be used in following those pointers.

B*-trees have nodes that look like those shown in Figure 1. The K_i are instances of the key domain, and the P_i are pointers. The P_i point to other nodes, or—in the case of the P_i in leaf nodes—they may point to records associated with the key values stored in the leaf. This arrangement gives leaf and nonleaf nodes

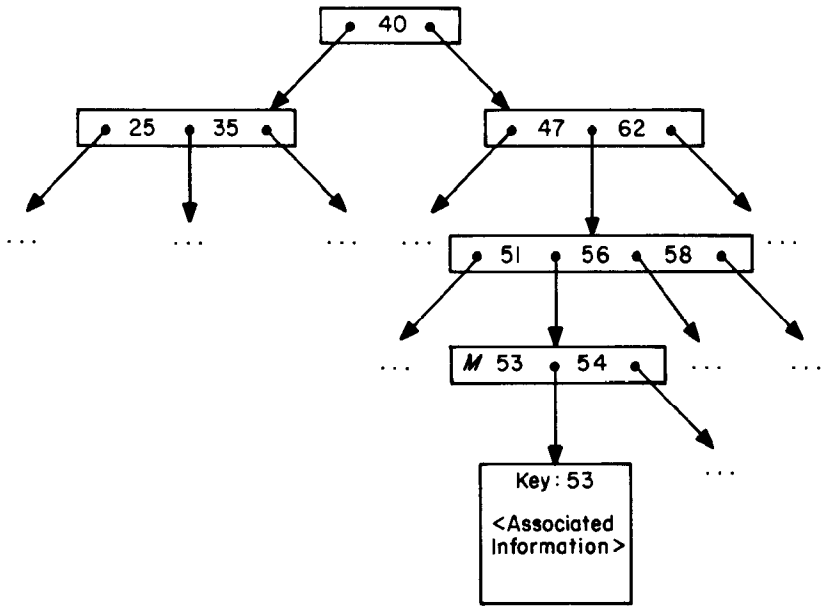

Fig. 2. An example B*-tree (with parameter $k = 2$).

essentially the same structure in our model. M is a marker that indicates a leaf node and occupies the same position as the first pointer in a nonleaf node. An example B*-tree is shown in Figure 2.

3.1.2 *Sequencing*

(a) Within each node, the keys are in ascending order.

(b) In the B*-tree an additional value, called the "high key," is sometimes appended to nonleaf nodes (Figure 3).

(c) In any node, N, each pointer, say P_i, points to a subtree (T_i) (whose root is the node to which P_i points). The values stored in T_i are bounded by the two key values, K_i and K_{i+1}, to the "left" and "right" of P_i in node N. This gives us a set of (pointer, value) pairs in nonleaf nodes, such that the set of values v, stored in subtree T_i, are bounded by

$$K_{i-1} < v \leq K_i,$$

where $k_0 = -\infty$ (or may be considered to be the last k in the node to the left; in any case, k_0 does not *physically* exist in node N), and K_{2k+1} is the high key, if it exists. The high key, then, serves to provide an upper bound on the values that may be stored in the subtree to which P_{2k} points and therefore is an upper bound on values stored in the subtree with root N. Leaf nodes have a similar definition (see Figure 3), with the stipulations that the K_i are the keys stored in the tree, and the P_i are pointers to their associated records.

3.1.3 *Insertion Rule*

(a) If a leaf node has fewer than $2k$ entries, then a new entry and the pointer to the associated record are simply inserted into the node.

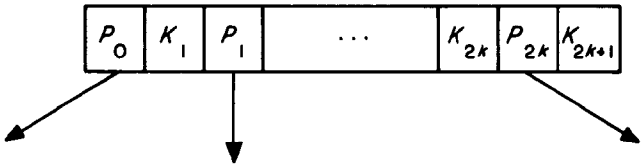

Fig. 3. A B*-tree node with a "high key" (K_{2k+1}).

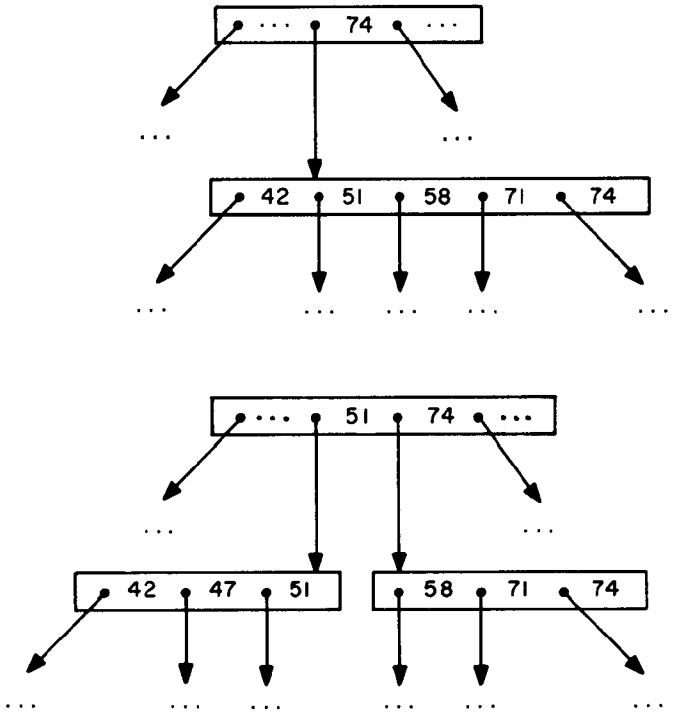

Fig. 4. Splitting a node after adding "47" ($k = 2$).

(b) If a leaf has $2k$ entries, then the new entry is inserted by splitting the node into two nodes, each with half of the entries from the old node. The new entry is inserted into one of these two nodes (in the appropriate position). Since one of the nodes is new, a new pointer must be inserted into the father of the old single node. The new pointer points to the new node; the new key is the key corresponding to the old half-node. In addition, the high key of each of the two new nodes is set.[1] Figure 4 shows an example of the splitting of a node.

(c) Insertion into nonleaf nodes proceeds identically, except that the pointers point to son nodes, rather than to data records.

[1] More specifically, when splitting node a into a' and b', the (new) high key for node a' is inserted into the parent node. The high key for node b' is the same as the old high key for a. A new pointer to b' is also inserted.

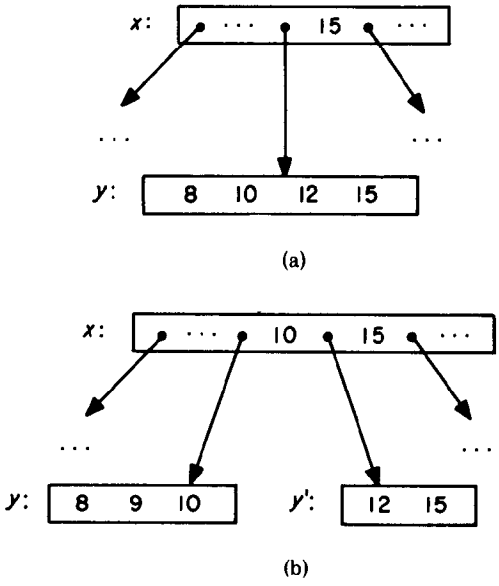

Fig. 5. Counterexample to naive approach.

A node (as in the rules given above) with less than $2k$ entries is said to be "safe" (with respect to insertion), since insertion can be done by a simple operation on the node. Similarly, a node with $2k$ entries is "unsafe," since splitting must occur. A similar definition holds for deletion from a node: a node is "safe" (respectively, "unsafe") if deletion can (cannot) occur in a node without its effects spreading to other nodes, that is, if the node has more than $k + 1$ (exactly $k + 1$) entries.

A simple example suffices to show that the naive approach to concurrent operation on B*-trees is erroneous.

Consider the B*-tree segment shown in Figure 5a. Suppose we have two processes: a search for the value 15 and an insertion of the value 9. The insertion should cause the modification to the tree structure shown in Figure 5b.

Now consider the following sequence of operations:

	search(15)	insert(9)
1.	$C \leftarrow$ read(x)	
2.		$A \leftarrow$ read(x)
3.	examine C; get ptr to y	
4.		examine A; get ptr to y
5.		$A \leftarrow$ read(y)
6.		insert 9 into A; must split into A, B
7.		put(B, y')
8.		put(A, y)
9.		Add to node x a pointer to node y'.
10.	$C \leftarrow$ read(y)	
11.	*error*: 15 not found!	

The problem is that the search first returns a pointer to y (from x) and then reads the page containing y. Between these two operations, the insertion process has altered the tree.

3.2 Previous Approaches

The previous example demonstrates that the naive approach to the concurrent B-tree problem fails: taking no precautions against the pitfalls of concurrency leads to incorrect results due to the operations of several processes. To put the problem in perspective, we briefly outline here some other approaches and solutions that have been proposed.

The first solution to the concurrent B-tree problem was offered by Samadi [13]. His approach is the most straightforward one that considers concurrency at all. The scheme simply uses semaphores (which themselves were first discussed in [5]) to exclusively lock the entire path along which modifications might take place for any given modification to the tree. This effectively locks the entire subtree of the highest affected node.

The algorithm proposed by Bayer and Schkolnick [3] is a substantial improvement to Samadi's method. They propose a scheme for concurrent manipulation of B*-trees; the scheme includes parameters which may be set depending on the degree and type of concurrency desired. First modifiers lock upper sections of the tree with writer-exclusion locks (which only lock out other writers, *not readers*). When the actual modifications must be performed, exclusive locks are applied, mostly in lower sections of the tree. This sparse use of exclusive locks enhances the concurrency of the algorithm.

Miller and Snyder [12] are investigating a scheme which locks a region of the tree of bounded size. The algorithm employs *pioneer* and *follower locks*, to prevent other processes from invading the region of the tree in which a particular process is performing modifications. The locked region moves up the tree, performing appropriate modifications. With the help of a locking discipline that uses a queue, readers moving down the tree "flow over" locked regions, avoiding deadlock. The trade-off between this algorithm and the one presented in the present paper is that the latter locks a substantially smaller section of the tree, but requires a slight modification to the usual B-tree or B*-tree structure, to facilitate concurrency.

Ellis [6] presents a concurrency solution for 2-3 trees. Several methods are used to increase the concurrency possible, and (it is claimed) these are easily extendible to B-trees. The paper includes an application of the idea of reading and writing a set of data in opposite directions (introduced by Lamport [11]), and that of allowing a slight degradation to temporarily occur in the data structure. Also, Ellis uses the idea of relaxing the responsibility of a process to finish its own work: postponing work to a more convenient time.

Guibas and Sedgewick [6a] have proposed a uniform "dichromatic framework" for balanced trees. This is a simplified view for studying balanced trees in general: it reduces all balanced tree schemes to special cases of "colored" binary trees and has the advantage of conceptual clarity. Those authors are using their framework to investigate a *top-down* locking scheme for concurrent operations, which includes splitting "almost-full" nodes on the way down the tree. This contrasts with the *bottom-up* scheme we present below. We project that their scheme will lock somewhat more nodes than ours (decreasing concurrency) and will require slightly more storage.

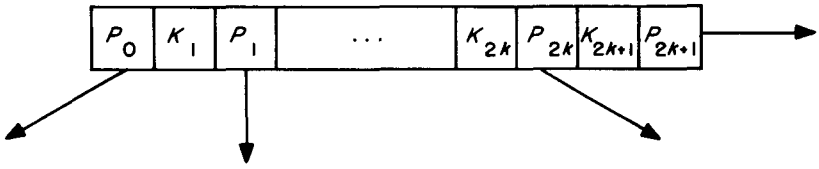

Fig. 6. A B^{link}-tree node.

Another approach to concurrent operations on B-trees is currently under investigation by Kwong and Wood [10].

3.3 B^{link}-Tree for Concurrency

The B^{link}-tree is a B*-tree modified by adding a single "link" pointer field to each node (P_{2k+1}—see Figure 6). (We pronounce "B^{link}-tree" as "B-link-tree.")

This link field points to the next node at the same level of the tree as the current node, except that the link pointer of the rightmost node on a level is a null pointer. This definition for link pointers is consistent, since all leaf nodes lie at the same level of the tree. The B^{link}-tree has all of the nodes at a particular level chained together into a linked list, as illustrated in Figure 7.

The purpose of the link pointer is to provide an additional method for reaching a node. When a node is split because of data overflow, a single node is replaced by two new nodes. The link pointer of the first new node points to the second node; the link pointer of the second node contains the old contents of the link pointer field of the first node. Usually, the first new node occupies the same physical page on the disk as the old single node. The intent of this scheme is that the two nodes, since they are joined by a link pointer, are functionally essentially the same as a single node until the proper pointer from their father can be added. The precise search and insertion algorithms for B^{link}-trees are given in the next two sections.

For any given node in the tree (except the first node on any level) there are (usually) two pointers in the tree that point to that node (a "son" pointer from the father of the node and a link pointer from the left twin of the node). One of these pointers must be created first when a node is inserted into the tree. We specify that of these two, the link pointer must exist first; that is, it is legal to have a node in the tree that has no parent, but has a left twin. This is still defined to be a valid tree structure, since the new "right twin" is reachable from the "left twin." (These two twins might still be thought of as a single node.) Of course, the pointer from the father must be added quickly for good search time.

Link pointers have the advantage that they are introduced simultaneously with the splitting of the node. Therefore, the link pointer serves as a "temporary fix" that allows correct concurrent operation, even before all of the usual tree pointers are changed for a new (split) node. If the search key exceeds the highest value in a node (as indicated by the high key), it indicates that the tree structure has been changed, and that the twin node should be accessed using the link pointer. While this is slightly less efficient (we need to do an extra disk read to follow a link pointer), it is still a correct method of reaching a leaf node. The link pointers should be used relatively infrequently, since the splitting of a node is an exceptional case.

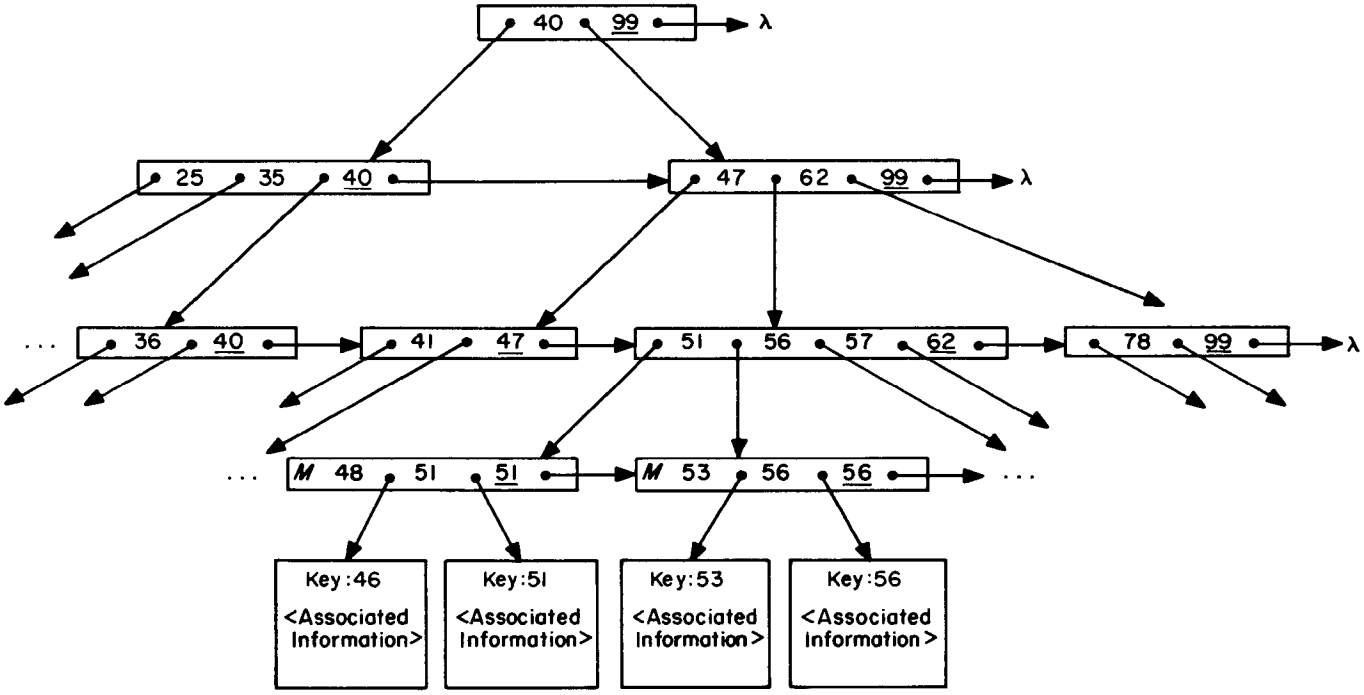

Fig. 7. A B^{link}-tree.

An additional advantage of the B^{link}-tree structure is that when the tree is searched serially, the link pointer is useful for quickly retrieving all of the nodes in the tree in "level-major" order, or, for example, retrieving only leaves.

4. THE SEARCH ALGORITHM

4.1 Algorithm Sketch

To *search* for a value, v, in the tree, the search process begins at the root and proceeds by comparing v with the values in each node in a path down the tree. In each node, the comparisons produce a pointer to follow from that node, whether to the next level, or to a leaf (record) node. If the search process examines a node and finds that the maximum value given by that node is *less* than v, then it infers some change has taken place in the current node that had not been indicated in the father at the time the father was examined by the search. The current node must have been split into two (or more) new nodes. The search must then rectify the error in its position in the tree by following the link pointer of the newly split node instead of by following a son pointer as it would ordinarily do.

The search process eventually reaches the leaf node in which v must reside if it exists. Either this node contains v, or it does not contain v and the maximum value of the node exceeds v. Therefore, the algorithm correctly determines whether v exists in the tree.

4.2 The Algorithm

Search. This procedure searches for a value, v, in the tree. If v exists in the tree, the procedure terminates with the node containing v in A and with t containing a pointer to the record associated with v. Otherwise, A contains the node where v would be if it existed. The notation used in the following algorithm is defined in Section 2. In this procedure, we use an auxiliary operation called *scannode*, defined as follows:

$x \leftarrow$ *scannode*(v, A) denotes the operation of examining the tree node in memory block A for value v and returning the appropriate pointer from A (into x).

```
procedure search(v)
current ← root;                                   /* Get ptr to root node */
A ← get(current);                                 /* Read node into memory */
while current is not a leaf do
begin                                             /* Scan through tree */
  current ← scannode(v, A);                       /* Find correct (maybe link) ptr */
  A ← get(current)                                /* Read node into memory */
end;
                                                  /* Now we have reached leaves. */
while t ← scannode(v, A) = link ptr of A do
                                                  /* Keep moving right if necessary */
begin
  current ← t;
  A ← get(current)                                /* Get node */
end;
                                                  /* Now we have the leaf node in which v should exist. */
if v is in A then done "success" else done "failure"
```

Note the simplicity of the search, which behaves just as a nonconcurrent search, treating link pointers in exactly the same manner as any other pointer.

Note also that this procedure does no locking of any kind. This contrasts with conventional database search algorithms (e.g., Bayer and Schkolnick [3]), in which all searches read-lock the nodes they examine.

5. THE INSERTION ALGORITHM

5.1 Algorithm Sketch

To *insert* a value, v, in the tree, we perform operations similar to that for *search* above. Beginning at the root, we scan down the tree to the leaf node that should contain the value v. We also keep track of the rightmost node that we examined at each level during the descent through the tree. This descent through the tree constitutes a search for the proper place to insert v (which is, say, node a).

The insertion of the value v into the leaf node may necessitate splitting the node (in the case where it was unsafe). In this case, we split the node (as shown in Figure 8), replacing node a by nodes a' (a new version of a which is written on the same disk page) and b'. The nodes a' and b' have the same contents as a, with the addition of the value v. We then proceed back up the tree (using our "remembered" list of nodes through which we searched) to insert entries for the new node (b') and for the new high key of a' in the parent of the leaf node. This node, too, may need to be split. If so, we backtrack up the tree, splitting nodes and inserting new pointers into their parents, stopping when we reach a safe node—one that does not need to be split. In all cases, we lock a node before modifying it.

Deadlock freedom is guaranteed by the well-ordering of the locking scheme, as shown below. Note the possibility that—as we backtrack up the tree—due to node splitting the node into which we must insert the new pointer may not be the same as that through which we passed on the way to the leaf. Rather, the old node we used during the descent through the tree may have been split; the correct insertion position is now in some node to the right of the one where we expected to insert the pointer. We use link pointers to find this node.

5.2 The Algorithm

In the following algorithms, some procedures are taken as primitives (in the manner of *scannode* above), since they are easily implemented and their operation is not of interest for purposes of this paper. For example,

$A \leftarrow$ *node.insert*(A, w, v) denotes the operation of inserting the pointer w and the value v into the node contained in A.

$u \leftarrow$ *allocate*(*1 new page for* B) denotes the operation of allocating a new page on the disk. The node contained in B will be written onto this page, using the pointer u.

"$A, B \leftarrow$ *rearrange old* A, *adding* ..." denotes the operation of splitting A into two nodes, A and B, in core.

Insert. This algorithm inserts a value, v (and its associated record), into the tree. When it terminates, this procedure will have inserted v into the tree and will

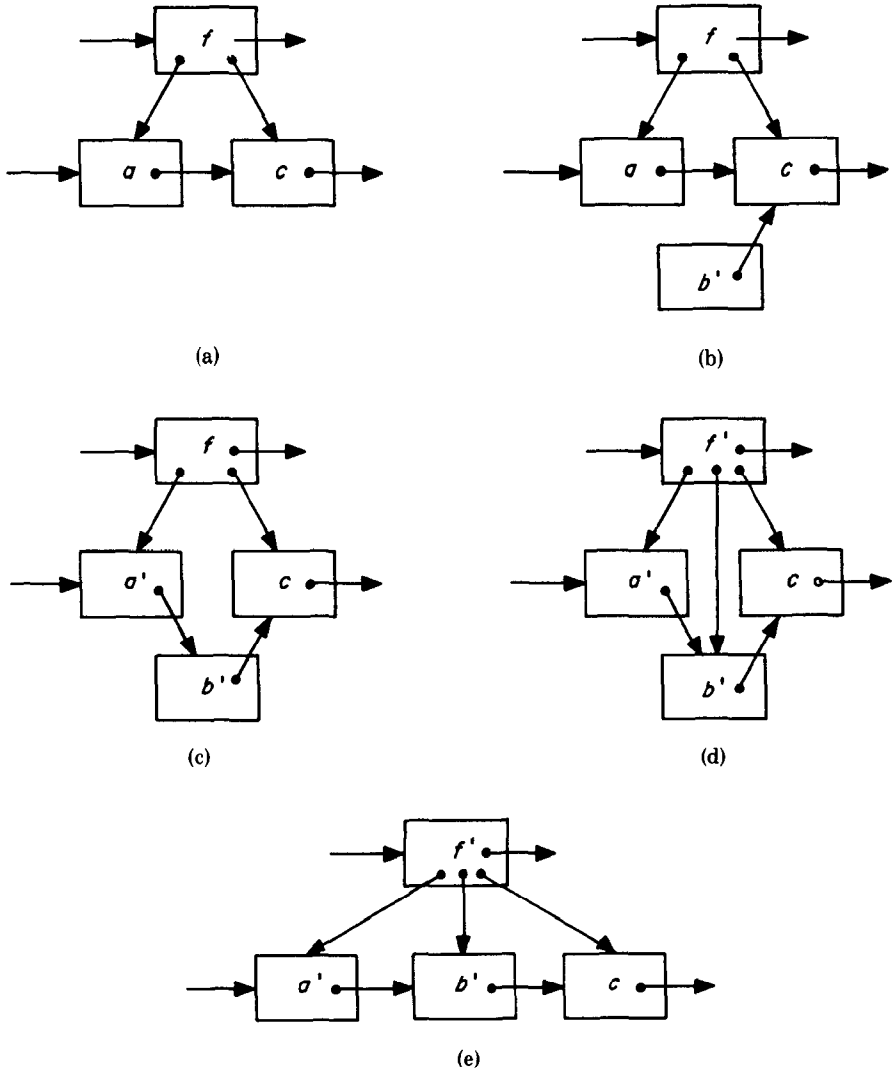

Fig. 8. Splitting node a into nodes a' and b'. (Note that (d) and (e) show identical structures.)

have split nodes, where appropriate, working its way back up the tree.

```
procedure insert(v)
initialize stack;                          /* For remembering ancestors */
current ← root;
A ← get(current);
while current is not a leaf do
begin                                      /* Scan down tree */
  t ← current;
  current ← scannode(v, A);
  if new current was not link pointer in A then
    push(t);                               /* Remember node at that level */
  A ← get(current)
end;
```

```
lock(current);                                    /* We have a candidate leaf */
A ← get(current);
move.right;                                       /* If necessary */
if v is in A then stop "v already exists in tree";  /* And t points to its record */
w ← pointer to pages allocated for record associated with v;
Doinsertion:
if A is safe then
begin
  A ← node.insert(A, w, v);                  /* Exact manner depends if current is a leaf */
  put(A, current);
  unlock(current);                                /* Success—done backtracking */
end else begin                                    /* Must split node */
  u ← allocate(1 new page for B);
  A, B ← rearrange old A, adding v and w, to make 2 nodes,
    where (link ptr of A, link ptr of B) ← (u, link ptr of old A);
  y ← max value stored in new A;                  /* For insertion into parent */
  put(B, u);                                      /* Insert B before A */
  put(A, current);                          /* Instantaneous change of 2 nodes */
  oldnode ← current;                          /* Now insert pointer in parent */
  v ← y;
  w ← u;
  current ← pop(stack);                           /* Backtrack */
  lock(current);                                  /* Well ordered */
  A ← get(current);
  move.right;                                     /* If necessary */
  unlock(oldnode);
  goto Doinsertion                          /* And repeat procedure for parent */
end
```

Move.right. This procedure, which is called by *insert*, follows link pointers at a given level, if necessary.

```
procedure move.right
while t ← scannode(v, A) is a link pointer of A do
begin                                             /* Move right if necessary */
  lock(t);                                        /* Note left-to-right locking */
  unlock(current);
  current ← t;
A ← get(current);
end
```

Note that this procedure works its way back up the tree one level at a time. Further, at most three nodes are ever locked simultaneously, and this occurs relatively infrequently: only when it is necessary to follow a link pointer while inserting a pointer to a split node. In this case, the locked nodes are: the original half of the split node, and two nodes in the level above the split node, while the insertion is moving to the right. This is a substantial improvement upon the solution of only unlocking a node when it is determined that the node is safe.

The correctness of the algorithm relies on the fact that any change in the tree structure (i.e., any splitting of a node) incorporates a link pointer; a split always moves entries to the right in the tree, where they are reachable by following the link pointer.

In particular, we always have some idea of the correct insertion position for an object (associated with some value) at any level, that is, the "remembered" node through which our search passed at that level. If the correct insertion position

has moved, it has done so in a known fashion, that is, via a node splitting to the right, leaving link pointers with which a search (or insertion) can find it. So the correct insertion position for an object is always accessible by a process starting at the old "expected" insertion position.

6. CORRECTNESS PROOF

In order to prove the correctness of our system, we need to prove that the following two propositions hold for each process:

(1) that it will not deadlock (Theorem 1),
(2) that it has correctly performed the desired operation when it terminates. More specifically:
    (a) that all disk operations preserve the correctness of the tree structure (Theorem 2),
    (b) that a consistent tree is seen by all processes other than the process making the modifications (Interaction Theorem 3).

6.1 Freedom from Deadlock

First, we undertake the proof of deadlock freedom of our system.

In order to do so, we impose an order on the nodes: bottom to top across levels and left to right within a given level. This is formalized in the following lemma.

LEMMA 1. *Locks are placed by the inserter according to a well-ordering on the nodes.*

PROOF. Consider the following ordering ($<$) on the set of nodes in the tree:

(1) At any time, t, if two nodes, a and b, are not at the same distance from the root of the tree (are not on the same level of the tree), then we say "$a < b$" if and only if b is less distant from the root (is at a higher level of the tree) than a.
(2) If a and b are equidistant from the root (are at the same level), then we say "$a < b$" if and only if b is reachable from a by following a chain of one or more link pointers (b is to the right of a).

We see by inspection of the insertion algorithm that if $a < b$ at time t_0, then $a < b$ at all times $t > t_0$, since the node creation procedure simply splits a node, x, into two new nodes, x' and x'', where $x' < x''$, and where

$$\forall_y\, y < x \Leftrightarrow y < x',$$

and

$$\forall_y\, x < y \Leftrightarrow x'' < y.$$

Therefore, the nodes form a well-ordering.

The inserter places locks on nodes, following the well-ordering. Once it places a lock on a node, it never places a lock on any node below it, nor on any node to the left on the same level.

Therefore, the inserter locks the nodes in the given well-order. Q.E.D.

Since the inserter is the only procedure that locks nodes, we immediately have the following theorem.

THEOREM 1: DEADLOCK FREEDOM. *The given system cannot produce a deadlock.*

6.2 Correctness of Tree Modifications

To ensure preservation of the tree structure, we must check all operations that modify that structure. First we note that tree modification can only be performed with a "put" operation. The insertion process has three places in its algorithm where a put is performed.

(1) "put(A, current)" for rewriting a safe node.
(2) "put(B, u)" for unsafe nodes. With this operation, we write the second (rightmost) of the two new nodes that are formed by a node splitting.
(3) "put(A, current)" for unsafe nodes. Here we write the first (leftmost) of the two nodes. We actually rewrite a page (node) that was already in the tree, and modify the link pointer of that page to point to the new node written by "put(B, u)."

Note that in the algorithm (for unsafe nodes), "put(B, u)" immediately precedes "put(A, current)" for unsafe nodes. We show that this ordering reduces the two puts to essentially one operation in the following lemma.

LEMMA 2. *The operation "put(B, u); put(A, current)" is equivalent to one change in the tree structure.*

PROOF. We assume that the two operations write nodes b and a, respectively. At the time "put(B, u)" is performed, no other node contains a pointer to the node (b) being written. Therefore, this put operation has *no effect* on the tree structure.

Now, when "put(A, current)" is performed, this operation modifies the node to which current points (node a). This modification includes changing the link pointer of node a to point to b. At this time, b already exists, and the link pointer of b points to the same node as the link pointer of the old version of a. This has the effect of simultaneously modifying a and introducing b into the tree structure. Q.E.D.

THEOREM 2. *All put operations correctly modify the tree structure.*

PROOF

Case 1. The operation "put(A, current)" for safe nodes. This operation modifies only one locked node in the tree; the correctness of the tree is therefore preserved.
Case 2. The operation "put(B, u)" for unsafe nodes. This operation does not change the tree structure.
Case 3. The operation "put(A, current)" for unsafe nodes. By the lemma, this operation both modifies the current node (say, a) and incorporates an additional node (say, b) into the tree structure: the node written by "put(B, u)." Similarly to case 1, a is locked at the time of "put(A, current)." The difference in this case is that the node is unsafe and must be split. But, by the lemma, we do this with a single operation, preserving the correct tree structure. Q.E.D.

6.3 Correct Interaction

It remains to show that other processes still operate correctly regardless of the action of an insertion process modifying the tree.

THEOREM 3: INTERACTION THEOREM. *Actions of an insertion process do not impair the correctness of the actions of other processes.*

In order to prove the theorem, we first consider the case of a search procedure interacting with an insertion, then of the interaction of two insertion procedures. In general, in order to show that an operation by an inserter does not impair the correctness of another process, we consider the behavior of that process relative to the operation in question. In all cases the operation is atomic.

Assume that the inserter performs a "put" at time t_0 on node a. Consider the time, t', at which the other process reads node a from the disk. Since "get" and "put" operations are assumed to be indivisible, either $t' < t_0$, or $t_0 < t'$. We show that the latter case presents no problem in the following lemma.

LEMMA 3. *If a process P reads node a at some time $t' > t_0$, where t_0 is the time at which a was changed by an insertion process, I, then the correctness of P is not affected by that change.*

PROOF. Consider the path that P follows through node a. The path that P follows *before* it reaches a will not be changed by I. Further, by Theorem 2 above, any change that process I makes in the tree structure will produce a correct tree. Therefore, the path followed by P from a (at time $t > t'$) will proceed correctly regardless of the modification. Q.E.D.

In order to easily break the proof of the theorem into cases, we list here the three possible types of insertion that may be performed for a value on a node.

Type 1. The simple addition of a value and associated pointer to a node. This type of insertion occurs when the node is safe.
Type 2. The splitting of a node where the inserted value is placed in the left half of the split node. The left half is the same node as that which was split.
Type 3. Similarly, the splitting of a node where the inserted value is placed in the right half of the split node. The right half is the newly allocated node.

We now undertake the proof of the theorem. We observe that there are several aspects (cases) to the correctness of the theorem, and we prove these separately.

PROOF. By Lemma 3, it is only necessary to consider the case where the search or insertion process P begins to read the node *before* the change is made by the insertion process I.

Part 1. Consider the interaction between the inserter I—which changes node n at time t_0—and a search process S—which reads node n at time $t' < t_0$. Let n' denote the node after the change. (The argument in this section is also applicable to the case where another inserter (I') is interacting with process I, and I' is performing a search.) The sequence of actions to be considered is: S reads node n; then I modifies node n to n'; then S continues the search based on the contents

of n. Consider three types of insertions:

Type 1. Process I performs a simple insertion into node n. For cases where n is a leaf, the inserter does not change any pointer. The result is equivalent to the serial schedule in which S runs before I. If n is a nonleaf node, a pointer/value pair for some node, m', in the next lower level of the tree is inserted in n. Assume that m' is created by splitting I into I' and m'. The only possible interaction is when S obtained the pointer to I prior to the insertion of the pointer to m'. The pointer to I now points to I', and S will use the link pointer in I' to reach m'. Thus the search is correct.

Types 2 and 3. The node n is split into nodes $n1'$ and $n2'$ by the insertion. For the leaf case, the search results on n and on $n1'$ and $n2'$ are the same, except for the newly inserted value which will not be found by S. If n is not a leaf, then a node below it has split, causing a new pointer/value pair to be inserted in node n, which causes n to split. By induction, the split in the level below node n is correct. By Lemma 3, the searching below node n is also correct. Therefore, we must simply show the correctness of the split of node n. Suppose node n splits into nodes $n1'$ and $n2'$ that contain the same set of pointers as node n, with the addition of the newly inserted node. Then starting from node n, the search will reach the same set of nodes in the next level as it would working from $n1'$ with a link pointer to $n2'$. The exceptional case is that in which the search would have followed the newly inserted pointer had it been present when process S read node n. In this case, the pointer followed will be to the left of that new pointer. This will lead the search to a node (say, k) to the left of the node (say, m) to which the new pointer points. Then the link pointer of k will be followed to (eventually) reach m. This is the correct result. (The argument for type 3 is identical to that for type 2, except that the new entry is inserted into the newly created (rather than the old) half of the split node. This makes no difference to the argument, however, since the node is read by S before the split takes place.)

Part 2. We next consider the case where process I interacts with another insertion process, I'. Process I' is either searching for the correct node for an insertion, backtracking to another level, or actually attempting to insert a value/pointer pair into the node n.

In the case where I' is searching for a node into which to insert a value/pointer pair, the search behaves in exactly the same fashion as a search process would. The proof is therefore the same as given above for a search process.

Part 3. In the case where I' is backtracking up the tree, as a result of node split in the level below, I' needed to back up in order to insert a pointer to the new half of the split node. Backtracking is done using the record kept in a stack during

the descent through the tree. At each level, the node that is pushed onto the stack is the rightmost node among those that were examined at that level.

Consider what may have happened to a given node, n, between the time we inserted it into the stack and the time we return to the node as we backtrack through the tree. The node may have split one or more times. These splits will have caused the formation of new nodes to the "right" of the node n. Since all nodes to the right of node n are reachable (via link pointers) the appropriate place to insert the value will be reachable by the insertion algorithm.

Part 4. In the case where process I' is attempting an insertion into node n, it will attempt to lock that node. But the process I will already hold the lock on node n. Eventually, I will release that lock, and I' will lock the node and then read it into memory. By the lemma above, the interaction is correct since the reading by I' takes place before the insertion by I. Either node n will be the correct place to make the insertion—in which case it will do so—or the search will have to follow the link pointer from the node to its right twin. Q.E.D.

6.4 Livelock

We wish to point out here that our algorithms do not prevent the possibility of livelock (where one process runs indefinitely). This can happen if a process never terminates because it keeps having to follow link pointers created by other processes. This might happen in the case of a process being run on a (relatively) very slow processor in a multiprocessor system.

We believe, however, that this is extremely unlikely to be a problem in a practical implementation, given the following observations.

(1) In most systems that we know of, processors run with comparable speeds.
(2) Node creation and deletion occur only a small percent of the time in a B-tree, so even a slow processor is likely to encounter little difficulty due to node creation or deletion (that is, it will be required to follow only a small number of link pointers).[2]
(3) Only a fixed number of nodes can be created on any given level of the tree, bounding the amount of "catching up" that a slow processor must do.[3]

We believe that these ideas combine to produce a vanishingly small probability of livelock for a process in a practical system (except perhaps in the case where the speeds of the processes involved are *radically* different). A simulation would enable us to verify that our system does work under "reasonable" conditions, and help us to put bounds on the admissible relative speeds of the processes.

In the case where processes do run at radically different speeds, we might introduce some additional mechanism to prevent livelock. Several alternatives are available for the implementation of such a mechanism. A complete discussion of methods for avoiding livelock is beyond the scope of this paper, but one

[2] It is interesting to note that all of the cases of any difficulty in the present system and in other related systems for concurrency occur only a very small fraction of the time. For example, in a B-tree nodes need be split infrequently compared to the number of insertions performed.

[3] Strictly speaking, this statement ignores the problem of "ghost" nodes created by deletion, which somewhat increases the number of nodes that can be viewed as being on any given level.

example of such a method might be to assign priorities to each process, based, perhaps, on the "age" of the process. This would guarantee that each process would terminate, since it would eventually become the oldest process, and hence the process with the highest priority.

7. DELETION

A simple way of handling deletions is to allow fewer than k entries in a leaf node. This is unnecessary for nonleaf nodes, since deletion only removes keys from a leaf node; a key in a nonleaf node only serves as an *upper bound* for its associated pointer; it is not removed during deletion.

In order, then, to delete an entry from a leaf node, we perform operations on that node quite similar to those described above for case 1 of insertion. In particular, we perform a search for the node in which v should lie. We lock this node, read it into memory, and rewrite the node after removing the value v from the copy in primary memory. Occasionally, this will produce a node with fewer than k entries.

Proofs of the correctness of this algorithm are analogous to the proofs for insertion. For example, the proof of deadlock freedom is trivial, since only one node need be locked by the deleter.

Similarly, correct operation relies on the observation that if a searcher reads the node before the value v is deleted, it will report the presence of v in the node. This reduces to the serial schedule in which the search runs first.

The system we have just sketched is far simpler than one that requires underflows and concatenations. It uses very little extra storage under the assumption that insertions take place more often than deletions. In situations where excessive deletions cause the storage utilization of tree nodes to be unacceptably low, a batch reorganization or an underflow operation which locks the entire tree can be performed.

8. LOCKING EFFICIENCY

Clearly, at least one lock is required in a concurrent scheme, in order to prevent simultaneous update of the same node by distinct processes.

The solution given above for insertion uses at most a constant number of locks (three) for any process at any time. It does this only under the following circumstances: an inserter has just inserted an entry into some node (leaf or nonleaf), and has caused that node to be split. In backing up the tree, in order to insert a pointer to the split half of the new node, the inserter finds that the old father of the split node is no longer the correct place to perform the insertion and begins chaining across the level of nodes containing the father in order to find the correct insertion position for the pointer. Three nodes are locked only for the duration of one operation.

This type of locking occurs rarely in a B^{link}-tree with a large capacity in each node. Therefore, we can expect an extremely small collision probability for this structure unless there are many concurrent processes running.

The behavior of this system could be quantified by simulation, which would be parameterized by the number of concurrent processes, the capacity of each node, and the relative frequencies of search, insert, and delete operations. Such a simulation would also be useful for comparison with other concurrency schemes.

9. SUMMARY AND CONCLUSIONS

The B-tree has been found to be widely useful in maintaining large databases. Concurrent manipulation of such data has the appeal that many users would be able to share data; further, this should be feasible, since there are few cases, in a large database, where the data needs of users will conflict.

We have given an algorithm which performs correct concurrent operations on a variant of the B-tree. The algorithm has the property that only a (small) constant number of locks need be used by any process at any time. The algorithm is straightforward and differs only slightly from the sequential algorithm for the same problem. (The gain in efficiency of the algorithm presented above, as compared with sequential algorithms, or other concurrent algorithms could be quantified by simulation.)

This effect is achieved by a small modification to the data structure that allows recovery in the case where the position of a process is invalidated by the action of another process (cf. [8]).

We hope to expand this work to a more general scheme for concurrent database manipulation. We wish to find a general scheme that entails only a small modification to the data structure and to the sequential algorithm for a database problem. This modification should nevertheless allow a process to recover when its actions have been rendered incorrect by changes to the data structure that have been made by another process.

Another direction for further work is the study of a general method for "parallelizing" algorithms: techniques for converting a (well-understood) sequential algorithm into a concurrent algorithm for the same problem. The goal is to exploit as much as possible the concurrent nature of the problem that the algorithm is designed to solve, without sacrificing the correctness of the algorithm.

REFERENCES

(Note. References [4, 9, 14] are not cited in the text.)

1. ASTRAHAN, M.M., ET AL. System R: Relational approach to database management. *ACM Trans. Database Syst. 1*, 2 (June 1976), 97–137.
2. BAYER, R., AND MCCREIGHT, E. Organization and maintenance of large ordered indexes. *Acta Inf. 1* (1972), 173–189.
3. BAYER, R., AND SCHKOLNICK, M. Concurrency of operations on B-trees. *Acta Inf. 9* (1977), 1–21.
4. DIJKSTRA, E.W., ET AL. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM 21*, 11 (Nov. 1978), 966–976.
5. DIJKSTRA, E.W. Cooperating sequential processes. In *Programming Languages*, F. Genuys, Ed. Academic Press, New York, 1968, pp. 43–112.
6. ELLIS, C.S. Concurrent search and insertion in 2-3 trees. Tech. Rep. 78-05-01, Dep. Computer Science, Univ. Washington, Seattle, May 1978.

6a. GUIBAS, L.J., AND SEDGEWICK, R. A dichromatic framework for balanced trees. In *Proc. 19th Ann. Symp. Foundation of Computer Science*, IEEE, 1978.

7. KNUTH, D.E. *The Art of Computer Programming*, vol. 3, Sorting and Searching. Addison-Wesley, Reading, Mass., 1973.
8. KUNG, H.T., AND LEHMAN, P.L. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst. 5*, 3 (Sept. 1980), 354–382.
9. KUNG, H.T., AND SONG, S.W. A parallel garbage collection algorithm and its correctness proof. In *Proc. 18th Ann. Symp. Foundations of Computer Science*, IEEE, Oct. 1977, pp. 120–131.
10. KWONG, Y.S., AND WOOD, D. Concurrency in B- and T-trees. In preparation.
11. LAMPORT, L. Concurrent reading and writing. *Commun. ACM 20*, 11 (Nov. 1977), 806–811.

12. MILLER, R., AND SNYDER, L. Multiple access to B-trees. In *Proc. Conf. Information Sciences and Systems* (preliminary version), Johns Hopkins Univ., Baltimore, March 1978.
13. SAMADI, B. B-trees in a system with multiple users. *Inf. Process. Lett. 5*, 4 (Oct. 1976), 107–112.
14. STEELE, G.L., JR. Multiprocessing compactifying garbage collection. *Commun. ACM 18*, 9 (Sept. 1975), 125–143.
15. WEDEKIND, H. On the selection of access paths in a data base system. In *Data Base Management*, J.W. Klimbie and K.L. Koffeman, Eds. North-Holland, Amsterdam, 1974, pp. 385–397.

Received June 1979; revised May 1980; accepted October 1980