JOURNAL OF COMPUTER AND SYSTEM SCIENCES 33, 275–296 (1986)

Concurrent Operations on B^*-Trees with Overtaking

YEHOSHUA SAGIV*

Department of Computer Science,
Hebrew University, Givat Ram 91904, Jerusalem, Israel

Received August 20, 1985; revised June 16, 1986

Algorithms for concurrent operations (i.e., searches, insertions, and deletions) on B^*-trees are presented. These algorithms improve those given by Lehman and Yao (*ACM Trans. Database Systems* **6**, No. 4 (1981), 650–670), since an insertion process has to lock only one node at any time (as opposed to locking simultaneously two or three nodes in *ibid.* Another improvement is the ability to compress the tree when some nodes become too sparse as a result of some deletions. Compressing the tree is done by a process that periodically scans the whole tree while running concurrently with the other operations. Alternatively, it is possible to initiate a compression process after each deletion that leaves a node less than half full. These compression processes may run concurrently with the other operations, and they scan only the nodes that have to be compressed. Each compression process has to lock simultaneously three nodes.

1. INTRODUCTION

The B-tree [1] and its variants are widely used as a data structure for large files. Several papers [2, 3, 7, 8, 10, 12] have described how to perform concurrent operations on B-trees. Clearly, as long as we have only *readers*, i.e., processes that only search for data, no scheduling is necessary. When there are also *updaters*, i.e., processes that either insert into or delete from the tree some data, it is easy to show that not every schedule of concurrent processes is correct. An updater is required to make changes in some subtree, which is called the *scope* of the updater. For example, the scope of an insertion is the subtree rooted at the deepest node, on the way to the leaf where the new data is to be inserted, that is not already full. When another process should access the scope of an updater, the two have to be scheduled somehow.

Some of the proposed solutions are "top-down" and others are "bottom-up." In the top-down solutions, an updater locks its scope to prevent other updaters from entering it throughout its operation. These solutions differ in the presence of readers in the scope of an updater. In some of them [12, 2],[1] readers are not allowed in the

* This work was supported in part by a grant of the Charles H. Revson Foundation.

[1] [12] is more restrictive than all other solutions, since it does not allow one reader to overtake another.

0022-0000/86 $3.00

scope of an updater throughout its operation, while in others, readers may be there during the searching phase of the updater [2] or even during the restructuring phase [3, 7]. In the bottom-up solutions [3, 8, 10],[2] an updater behaves as a reader on its way down (i.e., during the searching phase), and then moves up the tree while locking only few nodes simultaneously (at most three in [3, 8]) and making the necessary changes in the tree. The locks are used to guarantee that a change in a node is made as an atomic step, and no overtaking of one updater by another is possible when the two move up along the same path. The bottom-up solutions allow a higher degree of concurrency among updaters that have overlapping scopes.

A solution of another type is given in [5]. The idea is to split "almost-full" nodes on the way down. It is also claimed in [5] that it is possible to do the rebalancing in the "shadow" of the real tree—an approach that appears to be similar to our compression process that will be discussed later. However, no details of the concurrent algorithms (and exactly how they lock nodes) are given. Thus, it is hard to compare the general ideas in [5] with the solutions in [8] and in this paper. We project that the algorithms in [8] and in this paper are simpler and more efficient (i.e., they allow a higher degree of concurrency and create a *B*-tree that uses somewhat less space) than those of [5].

The solution in [8] is the most efficient, since only one type of locks is used (as opposed to four in the bottom-up solution of [3] and three in the top-down solutions of [2, 3, 7]), readers do not use any lock and can read a node even if it is locked by an updater, and an updater starts using locks only when it first reaches a leaf. That solution is based on a small modification in the structure of the *B*-tree, namely, at each level all the nodes are linked together by means of a pointer from a node to its right neighbor,[3] and one more key value is stored in each node.[4] A search in the tree may be prolonged as a result of having to move occasionally from a node to its right neighbor, but we feel that this is more than compensated for the fact that a process has to obtain considerably fewer locks as compared to all other solutions. The disadvantage in [8] is that the solution for deletions is the trivial one, i.e., data is deleted from a leaf, and no further action is taken even if the node becomes less than half full. Thus, space may be wasted and the height of the tree may be bigger than necessary. The bottom-up solution in [3] includes (as in [8]) only an algorithm for concurrent insertions, but it is reported that there is a solution for deletions provided that some modifications are made in the insertion algorithm, and only one deletion process is allowed to run in parallel with multiple insertions and readers.

The results of this paper are an improvement of those in [8]. First, we show that there is no need to prevent one updater from overtaking another on the way up (at

[2] [3] gives both top-down and bottom-up solutions.

[3] In practice many implementations of *B*-trees have these pointers anyhow (at least in the lower levels of the tree) to facilitate easy sequential traversal of the leaves.

[4] Also in [3] additional space is needed in each node for a key value and a pointer.

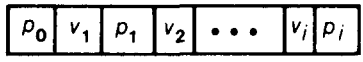

FIGURE 1

least not in the solution of [8]). This simple observation leads to a much simpler proof of correctness of the concurrent operations. It also implies that an insertion process has to lock only one node at any time (as opposed to locking simultaneously two or three nodes in [8]), since locks are needed only to ensure that rewriting a node is done as an atomic step. Second, we describe a compression process that redistributes data in the tree to ensure that each node is at least half full, and releases nodes that become empty.[5] Unlike the solution reported in [3], it is possible to run any number of compression processes in parallel with insertions, searches, and deletions (where a deletion is performed as in [8]). In fact, it is possible to initiate a compression process for each node that becomes less than half full as a result of a deletion. The compression process requires having an additional key value in each node of the tree, and it has to lock three nodes simultaneously. In our solution, concurrent compressions and insertions are deadlock free as a result of the fact that an insertion process has to lock only one node at any time. If our compression process were to run concurrently with the insertion processes of [8], a deadlock could occur.

The concurrent execution of compressions and readers may cause a reader to be sent to a node that does not have the data that the reader is supposed to find there. We solve this problem by restarting a process that reaches a wrong node. In all other solutions, the problem[6] of preventing a process from being sent to the wrong place has been solved by introducing several types of locks; and each process (even a reader) must lock every node before accessing it, and only after obtaining the lock on the next node it can release the lock on the previous node. Our approach seems to be more efficient, since the overhead in restarting some processes is likely to be smaller than in managing queues to grant several types of locks on each node. Essentially, our approach is to solve the problem when it occurs rather than to avoid it at all cost, and it is reasonable to assume that the problem occurs infrequently.

2. PRELIMINARIES

2.1. *B*-Trees and B^{link}-Trees*

The data structure used in the paper is a simple variation of the *B**-tree described in [14]. A *B**-tree serves as a dense index, i.e., the leaves contain pairs (v, p), where

[5] A similar idea, as an addition to the original algorithms of [8], has been independently proposed in [11].

[6] Of course, [8] solves this problem by using links, but this idea does not seem to carry over when we want to compress the tree.

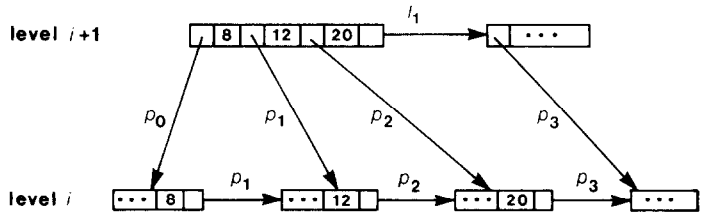

FIG. 2. If we omit the leftmost pointer and the links at level $i+1$, then we have the sequence $8p_1\,12p_2\,20p_3\cdots$ and that is also the sequence of high values and links at level i.

p points to the record with key value v. All the leaves are the same distance from the root, and we regard them as being at level 0; the nodes immediately above the leaves form level 1, and so on. The internal nodes have i values and $i+1$ pointers arranged in a sequence that starts with a pointer and alternates pointers with values, where $k \leqslant i \leqslant 2k$ for some fixed $k \geqslant 1$. The values in each level of the tree are in ascending order from left to right. A typical node is shown in Fig. 1.

The pointers in an internal node point to subtrees, where p_j $(0 \leqslant j \leqslant i)$ points to the subtree whose leaves have all key values v, such that $v_j < v \leqslant v_{j+1}$. There is no field for either v_0 or v_{i+1} in the node, but their exact values can be determined from the data in other nodes of the tree. During a search in the tree for a record with a key value v, we follow the pointer p_j provided that $v_j < v \leqslant v_{j+1}$, and we may assume that v_0 is $-\infty$ and v_{i+1} is $+\infty$. A leaf is similar to an internal node, with the only exception that p_0 is not used. Thus, a leaf has at least k and at most $2k$ pairs (v_j, p_j), where p_j points to the record with the key value v_j. Leaf and nonleaf nodes have the same structure, and in a leaf the field for p_0 has a special value indicating that the node is a leaf.

The B^{link}-trees [8] are obtained from the B^*-trees by adding to each node an additional pair (v_{i+1}, p_{i+1}). The value v_{i+1} is the *high value* of the node, i.e., the largest key value in the subtree rooted at that node. The pointer p_{i+1} points to the next node at the same level, and we call it a *link*. Thus, starting at the leftmost node at any level, we can traverse all the nodes at that level by following the links. The rightmost node at each level has $+\infty$ as its high value, and **nil** as its link. Now at each node, v_0 (which is still not physically present at the node) is either $-\infty$ (if we are at the leftmost node) or the high value of the left neighbor of the node. An internal node is, therefore, the root of the subtree having all values v, such that $v_0 < v \leqslant v_{i+1}$.

When a B^{link}-tree is created, the high value in each leaf is the same as the highest key value stored in that leaf, which means that the highest key value appears twice.[7] Now consider any level of the tree, say level i, and the next higher level, $i+1$. If we ignore the leftmost pointer at level $i+1$ (which points to the leftmost node at level i) and the links at that level, then level $i+1$ is a sequence of pairs (v, p), where v is a key value and p is a pointer to the right neighbor of the node (at level i) having

[7] That may change later due to deletions.

the high value v. This sequence is identical to the one consisting of the pairs (h, l), where h is a high value stored at level i and l is the link adjacent to it. Thus, level $i+1$ is actually repeated at level i. This simple observation (which is illustrated in Fig. 2) leads to the correctness of our algorithm.

2.2. *The Model*

Each node of a tree corresponds to a *page* or *block* of secondary storage. The function *get*(x) returns the contents of the node pointed to by x, and the procedure *put*(A, x) writes the data of buffer A in the node pointed to by x. When a *get* or a *put* is performed by some process on the node (pointed to by) x, no other process can read or write node x until the first one finishes. Thus, reading and writing of nodes are indivisible operations.

A process can lock a node, and that prevents any other process from locking the same node until the first process releases the lock. A lock on a node, however, does not prevent other processes from reading the locked node. The procedures *lock*(x) and *unlock*(x) obtain and release, respectively, the lock on the node pointed to by x. Locks are used to perform the sequence of reading a node and then rewriting it as an atomic operation, i.e., a process first locks the node, then reads it, changes the data and rewrites it, and finally unlocks it. In all other solutions (e.g., [8]) locks are also used to guarantee that one updater cannot overtake another.

3. Concurrent Searches and Insertions

3.1. *Ideas*

A search for a record with a key value v starts at the root, and at each level we follow the pointer p_j, such that $v_j < v \leqslant v_{j+1}$. When a leaf is reached, we check whether v is in the leaf and if so, we follow the associated pointer to the record. The search fails if v is not found in the leaf.

Now suppose that a new record r with a key value v is to be inserted into the tree. (It is assumed that space has already been allocated to r, and p points to it.) First we have to search for v, and if it is not found in the leaf, say A, where the search ends, then the pair (v, p) has to be inserted into A. If A is not full, this is done by rewriting A in a single indivisible operation, and the insertion is completed.

When A is full, it has to be *split*, i.e., a new leaf is created and one half of the pairs in A are shifted to it. The new leaf is inserted immediately to the right of A, and it gets the high value and the link of A. Leaf A has a new high value that is equal to the largest key value that remains in it, and its link points to the new leaf. Since A had $2k$ pairs, at least k pairs remain in each leaf after the insertion.

After the creation of the new leaf, a new pair (v', p') has to be inserted at level 1, where v' is the new high value of A and p' is a pointer to the new leaf. The new pair has to be inserted immediately to the left of the smallest key value u, such that $v' < u$. (Actually u is just the previous high value of A.) The insertion of (v', p') may

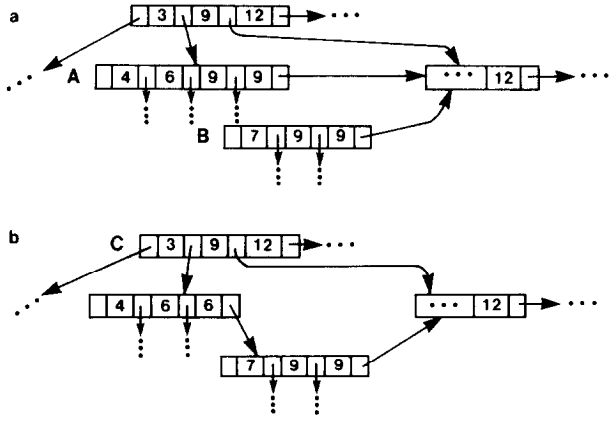

FIG. 3. (a) To insert the key value 7 and a pointer into the left node A, we first create the new node B and transfer the required data into it. (b) Then we write the new data in the old node. Later the pair $(6, p)$, where p is a pointer to the new node, will have to be inserted in node C immediately to the left of the value 9.

require the creation of a new node at level 1, and subsequently, another pair will have to be inserted at level 2. This may ripple through all the levels of the tree, and ultimately may result in the creation of a new root (having the old root and a new node as its children).

As shown in [8], the insertion[8] into node A can be done as an atomic step (even if the node is split). The idea is to create the new node, B, write into it the data that have to be shifted from A, and then write in A its new contents including the new high value and the link that points to B. The steps are illustrated in Fig. 3. Clearly, node A must be locked from the moment we first reach it, and determine that the new pair must be inserted into it, until after A is rewritten. Immediately after A is rewritten, the new node B is accessible through the link in A. The search procedure has to be changed slightly in order to locate the data stored in B once it is accessible. If during a search for a value v we are at a node C whose high value is smaller than v, then the next step is to follow the link to the right neighbor of C.

As explained in Section 2.1, each level above the leaves is a sequence of pairs (v, p), where v is a key value and p is a pointer. The creation of a new node at one level requires the insertion of a new pair (v, p) at the next higher level, but it does not change any existing pair at the higher level. In addition to that, the relative positions of the pairs at any level are always the same, i.e., in sorted order (according to key values). Therefore, several processes that concurrently insert at one level, do not have to perform the insertions at any higher level in the same

[8] We distinguish between an insertion into the tree, which may require insertions into several nodes at different levels, and an insertion into a node, which includes only the operations done at the level where the node is.

order as at the first level. Consequently, a lock on a node can be released immediately after the node is rewritten (and before a lock on any other node is obtained).

3.2. *Algorithms*

The procedure for a search in a B^{link}-tree is given in Fig. 4 (it is the same as in [8]). The procedure *next*(A, v) accepts the data of a node A and a value v, and returns either a pointer to the next level or the link of A (if v is larger than the high value of A). The procedure *movedown* starts at the root, and moves down the levels of the tree (using pointers and links) until a leaf is reached. The procedre *moveright* follows the links until a leaf with a high value equal to or larger than v is reached; this is the leaf where v belongs.

An insertion into the tree of a new record r, having a key value v, begins with a search for the leaf where v belongs (see Fig. 5). The procedure *movedown-and-stack* is similar to *movedown* with the addition of stacking the last node visited at each nonleaf level. Once a leaf is reached, we lock it and check whether v is in that leaf. Insertion is done only if v is neither in that leaf nor larger than its high value. If v does not belong in that leaf, then we have to unlock that leaf and call again the procedure *moveright* (see Fig. 4). Whenever *moveright* finds the leaf, A, where v belongs, we lock A and read it again to check whether v belongs in A, since A might have been split between the time we first read it and the moment we lock it.

```
procedure movedown;
begin
current := root;
A := get(current);
while A is not a leaf do
   begin
   current := next(A,v);
   A := get(current)
   end;
end;


procedure moveright;
begin
while t := next(A,v) is a link do
   begin
   current := t;
   A := get(current);
   end;
end;


procedure search(v:keyvalue);
begin
movedown;
moveright;
if v is in A then return pointer to record
else return nil    (* not found *)
end;
```

FIGURE 4

```
procedure movedown-and-stack;
begin
initialize stack;
current := root;
A := get(current);
while A is not a leaf do
   begin
   t := current;
   current := next(A,v);
   if current is not a link then push(stack,t);
   A := get(current)
   end;
end;


procedure insert(r:record);
begin
v := the key value of r;
p := pointer to pages allocated for r;    (* r is copied into these pages *)
completed := false;
movedown-and-stack;
repeat
  repeat
    found := true;
    lock(current);
    A := get(current);
    if v is in A then begin    (* v might already be in A only if A is a leaf *)
                      unlock(current);
                      print "v is already in the tree;"
                      release the space allocated for the new record;
                      stop
                      end;
    if v > highvalue(A) then begin
                      unlock(current);
                      found := false
                      moveright;
                      end;
  until found;
  if A is safe then insert-into-safe
  else if A is not the root then insert-into-unsafe   (* A is the root if current = root *)
       else insert-into-unsafe-root
until completed;
end;
```

FIGURE 5

When inserting into A, three cases are possible and they are handled by the procedures in Fig. 6. If A is *safe*, i.e., has fewer than $2k$ pairs, then we call the procedure *insert-into-safe*. If A is unsafe, then we insert the new pair into A, as described earlier, and then have to insert another pair at the next higher level. A special case (ignored in [8]) is when a new root has to be created for the next insertion, and this is handled by the procedure *insert-into-unsafe-root*. In this case we have to lock A until the new root is created, in order to avoid the creation of two roots simultaneously. If the next higher level already exists, then we call the procedure *insert-into-unsafe*. The node where the next insertion takes place is either the one which is presently at the top of the stack (i.e., the one through which we came down) or further to the right as a result of node splitting. Thus, we pop the stack and repeat the main loop of the procedure *insert*. There is one minor detail (ignored in [8]) of handling the case where the stack is empty although an inser-

```
procedure insert-into-safe;
begin
insert the pair (v,p) into A;
put(A,current);
unlock(current);
completed:=true;
end;


procedure insert-into-unsafe;
begin
q := pointer to a new page for B;
A,B := rearrange old A, adding (v,p), to make 2 nodes,
       where (link of A, link of B) := (q, link of A),
       highvalue(B) := highvalue(A),
       and A gets a new high value;
put(B,q);
put(A,current);
unlock(current);
p := q;
v := highvalue(A);
if stack is not empty then current := pop(stack);
else current := pointer to leftmost node at next higher level;
end;


procedure insert-into-unsafe-root;
begin
q := pointer to a new page for B;
A,B := rearrange old A, adding (v,p), to make 2 nodes,
       where (link of A, link of B) := (q, link of A),
       highvalue(B) := highvalue(A),
       and A gets a new high value;
put(B,q);
put(A,current);
v := highvalue(A);
u := highvalue(B);
r := pointer to a new page for the new root R;
R := (current,v,q,u,nil)
put(R,r);
root := r;
update the number of levels in the tree;
unlock(current);
completed := true;
end;
```

FIGURE 6

tion is required at a higher level that already exists. This may occur when the number of levels in the tree has been increased while our process is running. Thus, if the stack is empty, then at the end of the procedure *insert-into-unsafe* we assign *current* a pointer to the leftmost node at the next higher level.

3.3. *Creating a New Root*

We now discuss some of the finer details regarding the creation of a new root and the pointers to the leftmost node at each level. The B^{link}-tree has a *prime* block containing the number of levels in the tree and an array of pointers to the leftmost node at each level. Since the leftmost node at each level is never changed (once it is created), the creation of a new root entails incrementing the number of levels in the tree and adding one more pointer (to the new root) at the end of the array. The

address of the prime block must be known to the operating system, since the prime block contains the pointer to the root and, so, each access to the tree must begin by reading it. In the procedure *insert-into-unsafe-root*, assigning *r* to *root* and updating the number of levels are done by rewriting the prime block. The prime block does not have to be locked when it is rewritten, since a process rewrites it only when it has a lock on the root. Also note that the address of the prime block, unlike that of the root, never changes.

Testing whether *A* is the root (in the procedure *insert*) requires that the prime block be read. Note that this test is performed when *A* is locked and, so, it is impossible for a process to get a positive answer when some other process is creating a new root, regardless of whether that other process has already rewritten the prime block or not. In order to save reading the prime block, we can have in each node a bit indicating whether it is the root. In the procedure *insert-into-unsafe-root*, the bit of the old root is changed when it is rewritten.

To further illustrate that some details of a concurrent algorithm are rather complicated although the main idea is simple, we consider the following scenario. Suppose that one process is in the middle of creating a new root, while another reaches node *B*, the new right neighbor of the old root. The second process wants to insert into *B*, but *B* is already full and, so, it must split *B* and insert at the next higher level. The first process has not yet rewritten the prime block and, hence, the second process (when reading the prime block) does not find a pointer to the next higher level. This unlikely scenario may occur only if the process creating the new root is extremely slow compared to all other processes; nevertheless, it must be dealt with. One solution is to let the second process wait for a while and then read again the prime block. Another solution is to lock *B* when it is created, and release the lock when the lock on *A* (i.e., the old root) is released. This solution prevents the second process from inserting into *B* before the prime block is rewritten by the first process.

4. Correctness of Searches, Insertions, and Deletions

In this section we assume that deletions are handled as in [8], i.e., by locating the leaf where the record to be deleted is stored, and removing the record by rewriting this leaf. No further changes are made even if that leaf becomes less that half full. Thus, the execution of a deletion is similar to that of an insertion when no splitting occurs. In the next section, we will describe a *compression process* that redistributes data in the tree so that each node is at least half full.

Searches, insertions, and deletions return some values and possibly make some changes in the data structure. For example, if an insertion (or a deletion) is successful, then it returns **true** and adds (or deletes) a record; if it is unsuccessful, then it returns **false** and makes no changes. A search returns either a record or **false**, and makes no changes. Changes may occur only in the *logical data*, i.e., the records

stored in the leaf nodes, or also in the *search structure*, i.e., the nonleaf nodes.[9] It is unnecessary to require that all the changes done by concurrent operations be serializable.[10] Only changes of the logical data must be serializable. Changes to the search structure need not be serializable, but rather should preserve the validity of the search structure in the sense that all searches end in the right node. In the remainder of this section, we formally define and prove the correctness of our algorithms.

A *physical operation* is either a *get* or a *put*. A *logical operation* is either a *search*, an *insert*, or a *delete*. A schedule is a pair $S = (P, <)$, where P is a set of physical operations generated by a set of logical operations on a given tree, and $<$ is a partial order on the elements of P that specifies when one operation of P occurs before another. Two physical operations are temporally ordered if either they operate on the same block or both are generated by the same logical operation. In the terminology of traditional concurrency control theory, logical operations are *transactions*. But contrary to traditional theory, two schedules over the same set of logical operations need not have the same set of physical operations when they operate on a dynamic data structure. Two schedules (with the same initial tree) are *data equivalent* if

(1) they have the same set of logical operations,

(2) each operation returns the same value in both, and

(3) the logical data [11] are the same at the end of both schedules.

A schedule is *correct* if it is data equivalent to a serial schedule and the validity of the search structure is preserved. In the case of B^{link}-trees, the validity of the search structure has two aspects. First, during the execution of updating processes, searches must always end in the right node. Second, when all updating processes are completed, the new search structure must be correct in the sense that every possible search reaches the right node using only pointers (and no links).

THEOREM 1. *If S is a schedule of searches, insertions, and deletions that are executed by the algorithms described earlier, then S is correct and deadlock free.*

Proof. The schedule is deadlock free since each process locks only one node at any time.

In proving that S is a correct schedule, we ignore the details of creating a new root and handling the prime block, which were discussed in Section 3.3. We first show that S preserves the validity of the data structure. When S is completed, the search structure is valid, since each nonleaf level is just the sequence of high values

[9] The high values and links stored in the leaf nodes also belong to the search structure rather than the logical data.

[10] For an exposition of serializability of concurrent operations, see [13].

[11] A priori it is not clear that the logical data at the end of a schedule are well defined, since they might depend on the order of executing operations not related by $<$. For schedules produced by our algorithms, however, the theorem proved in this section shows that it cannot happen.

and links from the next lower level and, hence, searches never use links. During the execution of S the following assertion is always true. Let v be a high value of some block A in the chain of blocks at level i. Consider the leftmost value u at level $i+1$ such that $v \leqslant u$. Then the pointer immediately to the left of u points either to A or to a block on the left side of A. Since this assertion always holds during the execution of S, every search reaches the right node using pointers and links. Thus, S preserves the validity of the search structure.

We will now show that schedule S is data equivalent to some serial schedule. We define a precedence relation among the logical operations of S as follows. Consider two logical operations a_1 and a_2 on the same key value. If the last leaf read by one of them, say a_1, is to the left of the last leaf read by a_2, then a_1 must precede a_2. Now suppose that the same leaf, n, is the last one to be read by both a_1 and a_2. If a_1 performs its last physical operation on n before a_2 performs its last physical operation on n, then a_1 must precede a_2; otherwise, a_2 must precede a_1. The precedence relation just defined is acyclic, since only one physical operation can be performed on any node at any given moment, and for every two leaves n_1 and n_2, exactly one of the following is always true:

(1) n_1 and n_2 are the same leaf,

(2) n_1 is always to the left of n_2, or

(3) n_2 is always to the left of n_1.

Since the precedence relation is acyclic, there is at least one serial schedule L, such that if a_1 precedes a_2 in S, then a_1 precedes a_2 also in L. Clearly, S and L are data equivalent, since the value of each logical operation and its effect on the tree are solely determined by its last physical operation (and in S each logical operation reaches the right node by the validity of the search structure). ▮

5. Compressing the Tree

5.1. *A Single Compression Process*

The *compression process* scans the levels of the tree while redistributing data, so that each node has at least k pairs, and releasing nodes that become empty. This process can run concurrently with searches, insertions, and deletions. The idea is to traverse each level of the tree while examining pairs of nodes, say A and B. If A and B have together $2k$ or fewer pairs, then all the data is moved to one of them and the other is deleted. If one of them has fewer than k pairs but together they have more than $2k$ pairs, then the data is redistributed between them so that each one will have at least k pairs. In either case, the high value of A is changed and, consequently, a change is also required in the parent of A.

The heart of the compression process is the procedure *compress-level*(i), given in Fig. 7. This procedure reads a node, F, from level $i+1$, and examines pairs of

```
procedure compress-level(i);
begin
current := pointer to leftmost node at level i+1;
one := nil;
while current ≠ nil do
   begin
   lock(current);
   F := get(current);
   if one is not the rightmost pointer in F then
      begin
      if one = nil then one := leftmost pointer in F
      else one := the next pointer following one (in a left-to-right order);
      lock(one);
      A := get(one);
      two := link of A;
      if two = nil then return;
      lock(two);
      B := get(two);
      if two is in F then
            begin
            if either A or B has less than k pairs then
                  begin
                  rearrange A and B; (* if B becomes empty, its deletion bit is set on *)
                  put(A,one); unlock(one); put(F,current); unlock(current);
                  put(B,two); unlock(two);
                  end;
            if B was not deleted then one := two;
            end
      else begin  (* two is not in F *)
            unlock(current); unlock(one); unlock(two);
            if two should be stored in F (this depends only on the high values of B and F)
            then if A and B have to be rearranged
                  then wait & later restart the loop with one = previous value of one
            else begin
                  current := link of F;
                  one := nil;
                  end
            end
      end
   else begin  (* all pointers in F have been processed *)
       unlock(current);
       current := link of F;
       one := nil;
       end;
   end;
end;
```

FIGURE 7

adjacent children of F. For each pair, it rearranges the pair if necessary, makes the required changes in F and then moves to the next pair. When all the children of F have been examined, the next node at level $i+1$ is read. (If F has an odd number of children, then the last one will not be compressed even if it has fewer than k pairs.) The complete process consists of applying *compress-level* to all the levels of the tree, except the root, starting at level 0. As for the root, after applying *compress-level* to the level below the root, we examine the root and if it has only one child, then the root is removed and its child becomes the new root. (The exact details of removing the root are described in Sect. 5.4.)

Suppose that a tree T becomes empty as a result of deletions. One pass of *compress-level* over all the levels of T is not going to reduce the tree to a single node;

rather, $O(\log_2 n)$ passes over the tree are required, where n is the number of leaves in T. In a typical environment the number of insertions far exceeds the number of deletions, and running the compression process in the background as a low priority job is expected to allow only a small percentage of the nodes to be less than half full.

The compression process requires, as explained later, that v_0 be explicitly stored in each node. Recall that v_0 is the low value of a node, i.e., it is equal to the high value of the left neighbor of the node (or to $-\infty$ if the node is the leftmost one in its level). In addition to a low value, each node has a deletion bit indicating whether the node is deleted.

5.2. *The Algorithm*

The procedure *compress-level*(i) has a single loop that starts by locking a node, F, at level $i+1$, and reading it. Then *compress-level* chooses a pointer, *one*, that is either the leftmost pointer in F (if F has just been read for the first time) or the right neighbor of the pair of pointers chosen when F was previously read. (As explained shortly, it is also possible that the chosen pointer is the same as the one chosen when F was previously read.) The procedure locks the node, A, that *one* points to, and then reads A. The third node to be locked, B, is the one pointed to by the link of A. If F does not have a pointer to B, then all three nodes are unlocked and one of the following three cases applies:

(1) If *two*, the pointer to B, should be stored in F, and A and B have to be rearranged, then *compress-level* waits until *two* is inserted into F. In practice, *compress-level* is stopped for a while before locking again the three nodes and checking that F has a pointer to B. It is possible that *compress-level* will be waiting forever, because of constant splitting of node A, but it is expected that the chances of that happening are minuscule.

(2) If A and B do not have to be rearranged, then *compress-level* restarts the loop, and this time the next two children of F are going to be examined.

(3) If *two* does not belong in F, then *compress-level* moves to the right neighbor of F.

Once the pointer to B is in F, we check whether A and B have to be rearranged. If so, then two cases are possible:

(1) If there are no more than $2k$ pairs in A and B, then all the pairs from B are shifted into A (the high value and link of B replace those of A), the deletion bit in B is set on, and the old high value of A and the pointer to B are deleted from F. Finally, A, F, and B are rewritten and unlocked. Note that a node can be unlocked immediately after it is rewritten.

(2) If there are more than $2k$ pairs in A and B, then pairs are shifted from

one to the other so that each one will have at least k pairs, and the high value[12] of A is updated in A, B, and F. Finally, A, F, and B are rewritten and unlocked.

Immediately before any one of A, B, and F is rewritten and after all of them are rewritten, the tree is a valid data structure. Moreover, an insertion into or a deletion from any one of A, B, and F does not interfere with the rewriting of these nodes, since it requires a lock. Thus, we only have to consider what happens when a process reads one of A, B, and F before it is rewritten and another (that may be the same as the first) after it is rewritten. Several cases are possible, but the only two that cause a problem are as follows:

(1) A process reads a deleted node.

(2) A process reads a node in search of a value v, and v should be on the left side of that node, i.e., $v \leqslant v_0$, where v_0 is the low value of that node.

Case (1) can be handled as in [4], i.e., when node B is deleted, we can put a pointer to A in B. As a result, when a process reads the deleted B, it continues to A instead of having to restart.

In order to minimize the likelihood of case (2), we do the following: If some data are shifted from A to B or vice versa, then we should first rewrite the child that obtains new data, then the parent F, and finally the other child. Consequently, case (2) occurs only when data are shifted from B to A and a process reads B in search of a value that is already in A. Obviously, it is easy to detect whether case (2) has occurred, and when it happens to a process, the simplest solution is to restart that process. When restarting an insertion process, we only have to restart the search for the node where the current pair has to be inserted. Since it is easy to remember the level, say j, where the process is, we can start searching from the root for the node at level j in which the current pair has to be inserted. If we have to restart a search for a leaf,[13] then we may try at first to backtrack to the previous node visited, and only if we cannot resume the search from that node, we should restart at the root. A process must be extremely slow compared to the compression process in order for the backtracking not to succeed. Similarly, if we have to restart a search for a nonleaf node, then we can use the stack to restart at the next higher level rather than at the root (if we restart at a node whose low value is greater than or equal to the one we are looking for, then we should restart again at the root or at the next higher level).

5.3. *Releasing Deleted Nodes*

When a node is deleted, we cannot remove it, because other processes may have to read it. One solution [6, 9] is to record in the node the time of its deletion, and also store for each running process its starting time. A deleted node can be released

[12] Recall that the high value of A is also the low value of B.

[13] If this happens during either an insertion or a deletion, it means that no insertion into or deletion from any node has yet been done.

when all the currently running processes have started after its deletion time. A variation of this solution is to determine a fixed constant t_0, and restart a process that does not finish within t_0 seconds (a process is restarted as described earlier). In this way, a deleted node can be released after t_0 seconds.

5.4. *Multiple Compression Processes*

Instead of passing through the whole tree, we can check at the end of a deletion from a node whether it is less than half full and if so, put (a pointer to) it on a queue. Later a variant of *compress-level* will read a node, A, from the queue, find the parent and one of the neighbors of A, and either redistribute the data between the two neighbors or merge them. If A is merged with its neighbor, then either the merged node or its parent (or both) may be less than half full and, hence, must be put on the queue.

Actually, there are three possible ways of implementing the above idea:

(1) There is a single compression process that has a queue of nodes to be compressed.

(2) There are several compression processes that share a single queue. If a deletion process causes a leaf to become less than half full, then the process puts this leaf on the common queue. Each compression process removes one node at a time from the queue and compresses it. A process also puts on the queue nodes that become less than half full as a result of its actions. The advantage of this approach is the ability to dynamically change the number of compression processes according to the load on the system. A compression process can be stopped as soon as it finishes compressing a node, and the compression of the tree may continue with fewer processes, or may even be postponed for a while.

(3) There are several compression processes each having its own queue. Each one of these processes is initiated by a deletion that causes some leaf to become less than half full. A compression process starts with a leaf on its queue, and compresses it. Subsequently, it puts on its queue other nodes that become less than half full as a result of its actions. The compression process terminates when it attempts to remove the next node from its queue (in order to compress it) and the queue is empty.

In case (2), accessing the common queue requires locking it with an exclusive lock. In either case (2) or (3), a compression (or a deletion) process can put a node A on the queue only if it holds a lock on A; removing a node from the queue requires only a lock on the queue (if it is shared), but not on the node itself. Since a node is put on the queue only when it is being changed (i.e., rewritten), no extra lock has to be obtained in order to put A on the queue; rather, the current lock on A must be kept by the process until it puts A on the queue. It is expected that in most practical situations the queue is small enough to be kept in main memory.

We shall now give more details. Generally, they apply to each one of the three possible methods. When there are differences among these methods, we point them

out. If a node A has to be compressed, then the the following information is put on the queue:

(1) A pointer to node A.

(2) The level of A.

(3) The high value of A.

(4) The stack of A, i.e., a stack of pointers (one from each level of the tree) that point to the nodes on the path from the root to A. Note that when a deletion process has to put a leaf A on the queue, then there is a stack of pointers that was created by the procedure *movedown-and-stack* when the deletion process moved from the root to A. The deletion process puts this stack, along with the other information about A, on the queue. When a compression process removes A from the queue, it uses (as explained below) the stack of A in order to locate the parent of A. If as a result of compressing A, its parent becomes less than half full (and, hence, has to be put on the queue), then the stack of the parent is obtained by popping the stack of A.

Note that a record of information on the queue for a particular node is uniquely identified by the pointer to that node.

When several processes share a common queue, a process must check that A is not already on the queue before putting if there. If A is already on the queue, then the process must update the information about A rather than store it a second time. Since a process attempts to put a node on the queue only when it has a lock on this node, it follows that the high value of A available to the process is either identical to or more recent than the one stored on the queue. As we shall see later, it is important to update the high value of A. As for the stack of A, the process does not necessarily have a stack that reflects a more recent structure of the tree (as compared to the stack which is already stored on the queue), although this is probably true in most practical cases. In any case, it is not necessary to update the stack of A. Note that the level of a node A is never changed.

When a compression process removes a node A (i.e., the above information about A) from the queue in order to compress it, then at first the process must locate the parent, F, of A. In essence, it has to find the node, in the level immediately above A, that should contain the high value of A. The search is started in the node, D, which is pointed to by the top of the stack of A. It could be that this pointer is outdated, i.e., the node to be found is on the left side of D, or that there is no pointer at all, i.e., the stack of A is empty. In either case, the process has to restart the search from the root (or, alternatively, from the leftmost node at the level of D), as explained in Section 5.2 (therefore, the process has to know the level of A in order to determine the level of its parent). Note that the search for F is done in the same way as the search, in the procedure *insert*, for the parent of a node that has been split. More specifically, a node is locked only after it has been found to be the one that should contain the high value of A (as determined from the low and

high values of that node); and after it has been locked, it is read again to make sure that it is still the required node.

Once the parent, F, is located, the compression process, while keeping the lock on F, has to check whether F still has the pair (p, v), where p is the pointer to A and v is the high value of A (i.e., the high value that the process has obtained from the queue). If F does not have this pair[14] and the current high value of A is different from the one obtained from the queue, then (as we shall argue later) the process does not have to consider A, i.e., it can unlock F and continue by removing the next node from the queue. If F does not have this pair and the current high value of A is still the same, then A has to be put back on the queue in order to be considered again later. If F does have this pair, then node A and one of its neighbors must be locked according to one of the following two cases:

(1) If the pointer to A is not the rightmost pointer in F, then the process locks node A and then the node, say B, pointed to by the link of A. Next, the process checks whether there is a pointer to B in F. If there is, then the process proceeds to rewrite[15] the locked nodes and then to unlock them as done in the procedure *compress-level*. If there is no pointer to B in F, then the nodes F, A, and B have to be unlocked and A has to be put again on the queue (note that this is done while A is still locked; also note that if A is already on the queue when the process attempts to put it back, then the process should update the information about A according to the current status of A). Later A will have to be considered again (and at that time the parent of A will hopefully have a pointer to the right neighbor of A). Alternatively, instead of putting A back on the queue, the process may try (after unlocking B and while F and A are still locked) to lock the left neighbor of A, as explained in the next case.

(2) If the pointer to A is the rightmost pointer in F (or if F does not have a pointer to the right neighbor of A), then the compression process may try to lock the left neighbor of A. This is done by picking from F the pointer that precedes the pointer to A, and then locking the node, say B, it points to. Next, the process reads B and checks whether the link of B points to A, and if so, it also locks[16] A, and proceeds to rewrite (see footnote 15) the locked nodes and then to unlock them as done in the procedure *compress-level*. If the link of B does not point to A, then the nodes have to be unlocked and A has to be put back on the queue in order to be considered again later. Note that this is a case in which a process attempts to put a node on the queue without holding a lock on the node (but while locking the queue). This certainly does not cause any problem if A is not already on the queue when the process attempts to put it back. If A is already on the queue, then the process should not update the information on the queue, because the information

[14] Note that this includes the case where both p and v appear in F, but v does not immediately follow p.

[15] It could be that A does not have to be compressed, since it is now at least half full. In this case, F, A, and B are unlocked without rewriting them.

[16] Note that if the previous case was tried first, then A is already locked.

on the queue must have been put there after the process removed A and, hence, is more recent.

In either case (1) or (2), if A is merged with its neighbor, then the process has to put (or update) F on the queue if it has become less than half full. As for the two nodes that are merged, one of them is deleted and, hence, the compression process should remove it from the queue if it is there. The other node has to be put (or updated) on the queue if it is less than half full.

In addition to cases (1) and (2) above, there are few more special cases. First, consider the case where the pointer to A is the only one in F, and F is not the root (we assume that the root is marked by a special bit, as explained in Sect. 3.3). In this case A must be put back on the queue. Moreover, since F has only one pointer, then either F is also on the queue and, hence, must be compressed before A,[17] or more pointers should be inserted into F as a result of recent splits and, therefore, A cannot be compressed until a pointer to one of its neighbors is inserted into F.

A second special case is when F is the root, and F has either only one child or two children that can be merged. In this case the height of the tree can be reduced. If F has only one pointer (which must be the pointer to A, or else A should not be compressed currently), then the process locks and reads A, and checks whether its link is **nil**. If it is **nil**, then A is the only child of the root (since the leftmost pointer is never removed from F), and, therefore, the process can make A the new root as will be explained shortly. If F has two pointers, then the process first locks the leftmost child and checks that its link is the same as the other pointer in F. If it is not the same, then A has to be put back on the queue according to either case (1) or (2) above. If it is the same, then the process locks the node pointed to by this link, and now A can be compressed. However, if the second child that has been locked has a **nil** link and the two children can be merged, then the merged node is made the new root as follows:

(1) First, the process rewrites the leftmost child of F. When rewriting this child, the process sets on the bit indicating that is now the root.

(2) The process rewrites the prime block, and then releases the lock on the new root.

(3) The process rewrites the other child of the old root to mark it as deleted, and then releases the lock on this child.

(4) The process rewrites F to mark it as deleted, and then releases the lock on F.

If F has only one child (which must be A) and A itself has only one pointer, then after locking A (and before the above 4 steps are performed), the node pointed to by A is read and locked in order to determine whether it is the only child of A. This may continue to any number of levels until a node, D, is locked, where D either has

[17] Therefore, it is a good idea to give priority to nodes having a higher level and remove them first from the queue.

more than one child or is a leaf. In a sequence of steps similar to the above 4 steps, node D is made the new root.

Another rare case occurs when the top of the stack of A points to an empty node. In general, this is not a problem, since a deleted node points to the node with which it was merged, and the search continues from that node. However, it could be that the whole level is deleted, and this is detected when the search for the parent of A reaches a deleted node whose link is **nil**. In this case nothing has to be done about A, since it follows that the level of A has become the root after A was put on the queue.

We will now describe when a deleted node can be released to the operating system. The stack of a node A has a time stamp which is equal to the starting time of the deletion process that has created it. A node that becomes empty at time t can be released when all active searches, insertions, and deletions have started after time t, and the stacks of the nodes that are either currently being compressed or are on the queue (or queues) have only time stamps that are younger than t.

Following is a proof of correctness of the concurrent operations described in this paper (we assume multiple compression processes with one or more queues, as described in this section; the same is true for a single compression process, as described in Sect. 5.1).

THEOREM 2. *The concurrent execution of any number of searches, insertions, deletions, and compressions is correct.*

Proof. When insertion, deletion, and compression processes run concurrently, no deadlock is possible for the following reasons. Insertions and deletions processes lock only one node at any time. Compression processes lock three nodes simultaneously, but they first lock a node at one level and then lock two children of that node. To see that a deadlock is impossible, we draw arcs among the nodes of the tree as follows: Once a compression process locks the first node, we put an arc from that node to the second node that the process has to lock. Once the second node is locked, we add an arc from the second node to the third node that the process has to lock. Once the process has locked all three nodes, we remove its two arcs, since now the process can finish and will eventually unlock the three nodes (before trying to acquire any new locks). A deadlock situation occurs when some of the arcs form a cycle. But a cycle cannot be formed, since arcs go either from one level to the next lower level or from one node to its sibling; and among the children of any node there can be at most one arc at any time, since the second and third nodes that a compression process locks are the children of the first node it locks. Note that a compression process does not have to lock the two children (of the first node it locks) in any particular order, i.e., it can lock any of the two before the other one.

A process may be in a live lock if it repeatedly fails to get a lock on some node or if it repeatedly has to restart a search for some node. However, in this formal proof, we consider a finite concurrent schedule and prove its correctness. Since the schedule is finite, we can prove that all concurrent processes will terminate within a

finite amount of time. If a process has to restart a search from the root, it means that one more node has been compressed. Similarly, when a process is waiting for a lock, another insertion, deletion, or compression is being done during that time. Thus, while some processes are waiting or restarting, other processes are able to finish and, so, the waiting and restarting processes are also going to finish eventually. In a practical environment where new processes are constantly being created, the system may suspend from time to time the creation of new processes in order to make sure that all currently running processes are going to finish.

A concurrent schedule of searches, insertions, deletions, and compressions is correct in the sense that its logical operations are data equivalent to those of a serial schedule. A formal proof of this fact is similar to that of Theorem 1, and will not be repeated. Suffices to say that this fact is true simply because each process performs its logical operation as an atomic step (i.e., by writing a single block). Moreover, using the low and high values of a node, a process always verifies that it does its logical operation in the right node.

The schedule cannot end with a tree which either is not fully compressed or has a node without a pointer to that node in the next higher level, since this would imply that there is either a compression process or an insertion process that has not yet finished. The only nontrivial point to be proved is the correctness of the action taken by a compression process that has to compress a node A and does not find the pair (p, v) in the parent F, where v is the high value of node A obtained from the queue and p is the pointer to A, and the current high value of A is different from v. Recall that in this case, the process discards A and continues with another node from the queue. We have to show that in spite of this action, node A is compressed when the schedule ends. So, suppose that the process does not find the pair (v, p) when it locks and reads F, and the current high value of A is different from v. This means that A was involved in a split or a compression after time t_1, which is either

(1) the time in which the process removed A from the queue, if the queue is shared, or

(2) the time in which the process put A on the queue, if each process has its own queue.

In either case, if A still has to be compressed as a result of an action after time t_1, then either A is currently on the queue (or on a queue of some process) or another process is currently attempting to compress A. Thus, the process currently locking F can unlock F and discard the information about A, since another process will eventually compress A.

Now consider deletions of an existing root by compression processes, and creations of a new root by insertion processes. These are done while holding a lock on the current root and, therefore, only one process can delete or create a root at any time; and only that process can rewrite the prime block while it is deleting or creating the root (since rewriting the prime block requires locking the current root). Therefore, operations are done correctly. ▮

ACKNOWLEDGMENTS

The author thanks Danny Rechter and Betty Salzberg for helpful comments, including the suggestion that the child which gains new data should be rewritten first and then the parent and the other child.

REFERENCES

1. R. BAYER AND E. MCCREIGHT, Organization and maintenance of large ordered indexes, *Acta Inform.* **1** (1972), 173–189.
2. R. BAYER AND M. SCHKOLNICK, Concurrency of operations on *B*-trees, *Acta Inform.* **9** (1977), 1–21.
3. C. S. ELLIS, Concurrent search and insertion in 2–3 trees," *Acta Inform.* **14** No. 1 (1980), 63–86.
4. C. S. ELLIS, Extendible hashing for concurrent operations and distributed data, *in* "Proceedings, 2nd ACM SIGACT-SIGMOD Sympos. on Principles of Database Systems," Atlanta, March 1983, pp. 106–115.
5. L. J. GUIBAS AND R. SEDGEWICK, A dichromatic framework for balanced trees, *in* "Proceedings, 19th Annu. Sympos. Found. of Comput. Sci.," 1978, pp. 8–21.
6. H. T. KUNG, AND P. L. LEHMAN, Concurrent manipulation of binary search trees," *ACM Trans. Database Systems* **5** No. 3 (1980), 354–382.
7. Y. S. KWONG AND D. WOOD, A new method for concurrency in *B*-trees, *IEEE Trans. Software Engrg.* **SE-8** No. 3 (1982), 211–222.
8. P. L. LEHMAN AND S. BING YAO, Efficient locking for concurrent operations on *B*-trees, *ACM Trans. Database Systems*, **6**, No. 4 (1981), 650–670.
9. U. MANBER AND R. E. LADNER, Concurrency control in a dynamic search structure, *ACM Trans. Database Systems* **9**, No. 3 (1984), 439–455.
10. R. MILLER AND L. SNYDER, Multiple access to *B*-trees, *in* "Proc. Conf. Inform. Sci. and Systems," Johns Hopkins Univ., Baltimore, March 1978.
11. B. SALZBERG, "Restructuring the Lehman–Yao Tree," Tech. Report BS-85-21, College of Computer Science, Northeastern University, Boston, Mass., Jan. 1985.
12. B. SAMADI, *B*-trees in system with multiple users, *Inform. Process. Lett.* **5**, No. 4 (1976), 107–112.
13. J. D. ULLMAN, "Principles of Database Systems," 2nd Ed., Computer Sci., Rockville, 1982.
14. H. WEDEKIND, On the selection of access paths in a data base systems, *in* "Data base Management," (J. W. Klimbie and K. L. Koffeman, Eds.), pp. 385–397, North-Holland, Amsterdam, 1974.