Information Sciences International Journal
Elsevier Publishing Co., New York, NY, Vol. 48, No 1, June 1989, pp. 27-51.

Operation-Specific Locking in Balanced Structures

ALEXANDROS BILIRIS

Computer Science Department, Boston University, Boston, Massachusetts 02215

ABSTRACT

Balanced structures, variations of *B*-trees, have been used as an access aid for both primary and secondary indexing for quite some time. This paper presents a deadlock-free locking mechanism for *B*-trees in which different processes make use of different lock types in order to reach the leaf nodes. The compatibility relations among locks on a node do not exclusively depend on their type, but also on the node status and the number and kind of processes acting currently on the node. As a result, a number of insertion or deletion processes can operate concurrently on a node. The paper presents an appropriate recovery strategy in case of failure, and discusses the protocol modifications that are required so it can be used in other similar structures such as B^{+}-trees, compressed *B*-trees, and *R*-trees for spatial searching.

1. INTRODUCTION

A great deal of the time spent during the database access is attributable to the searching through indexes. The *B*-tree and its variants have become the most widely used access aids. Maximizing concurrency on them is one of the factors contributing most to the overall degree of concurrency. Figure 1(a) shows a *B*-tree of level three whose nodes may store from two up to four (key, pointer) pairs. We may easily show that taking no precautions against the anomalies of concurrency leads to incorrect results.

Assume that two processes act on the *B*-tree of Figure 1(a). The first is an insertion for key 36, and the other is a search for key 38. Now suppose the sequence of operations in Table 1. It's obvious that the search operation makes a wrong conclusion about the existence of key 38. Between the time the search process finds the appropriate leaf (node N10, step 4) and the time it reads it (step 8), the insert process has already moved 38 from N10 to N13 (step 6), and therefore the searching is incorrect.

Similar problems exist among insertions and deletions. Consider the tree of Figure 1(d), and suppose again the existence of two processes: a deletion of key 32, and an insertion for the key 31. Suppose also the sequence of operations in

0020-0255/88/$03.50

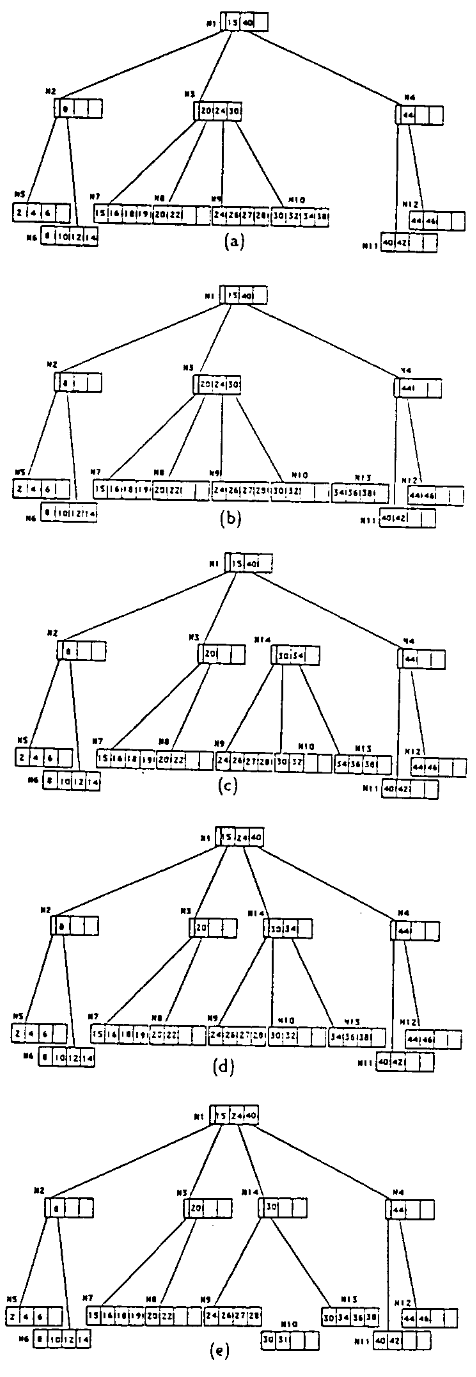

Fig. 1. (a) through (e) illustrate concurrency anomalies.

TABLE 1

	Insert 36	Search 38
1	find proper child of N1 for 36 [child is N3]	
2	Find proper child of N3 for 36 [child is N10]	
3		Find proper child of N1 for 38 (child is N3)
4		Find proper child of N3 for 38 (child is N10)
5	N10 is leaf and full	
6	Add 36 in N10 ⇒ split N10 into N10, N13 [see Figure 1(b)]	
7	Add (36, N13) in N3 ⇒ split N3 into N3, N14 [see figure 1(c)]	
8		N10 is leaf, search N10 for 38 Key 38 not found!
9	Add (24,N14) in N1 [see Figure 1(d)]	

TABLE 2

	Delete 32	Insert 31
1	Find proper child of N1 for 32 (child is N14)	
2		Find proper child of N1 for 31 (child is N14)
3	Find proper child of N14 for 32 (child is N10)	
4		find proper child of N14 or 31 (child is N10)
5	N10 is leaf, delete 32 from N10	
6	Add the rest of N10 in N13	
7		N10 is leaf and not full
8		Add 31 in N10
9	Delete node N10 from N14 [see Figure 1(e)]	

Table 2. The problem here is that the new key (31) has been added to a node (N10) which is deleted later by the deletion process.

Essentially, all the known concurrency control techniques [4] can be employed to synchronize simultaneous access on the tree. It has been observed, however, that semantic knowledge about individual objects that a process manipulates, or about the operations that a process performs on an object, can increase concurrency [13, 24, 34, 35, 6, 37]. For example, non-two-phase locking protocols (graph protocols [38, 18, 3], tree protocols [39, 8]) have been proposed where the additional information on the way that processes access the database is used to increase concurrency.

Working on this direction, we developed a deadlock-free locking protocol that takes advantage of how nodes are organized, how processes access nodes, and what kind of modifications these processes can impose on a tree node. As a result, many insertion or deletion processes can operate concurrently on a node.

We start the description of the proposed protocol by giving some background information in the next section. Section 3 discusses the locking rules and the storage model. Sections 4, 5, and 6 present the algorithms for searching, insertion, and deletion. In Sections 7 and 8, we present an informal proof of correctness and suggest recovery strategies in case of failure. Finally, Section 9 discusses the protocol applicability to *B*-tree-like structures, and Section 10 summarizes this work.

2. BACKGROUND

DEFINITIONS

A *B*-tree of order M is a balanced tree with the following properties [1]: Every node has between M and $2M$ children, except for the root which has between two and $2M$. A leaf node with s pointers contains s keys. A nonleaf node consists of s pointers $(P_0, P_1, \ldots, P_{s-1})$ to its children and $s-1$ keys $(K_1, K_2, \ldots, K_{s-1})$ arranged in such a way that for every key K in the subtree pointed to by P_i, the following relationships hold:

$$\begin{aligned} i = 0 &\rightarrow K < K_1, \\ 0 < i < s-1 &\rightarrow K_i \leqslant K < K_{i+1}, \\ i = s-1 &\rightarrow K_i \leqslant K. \end{aligned}$$

We assume that each node is organized sequentially, and the set of all keys appears in the leaves. In case that P_i belongs to a leaf node it will point to records keeping actual data associated with the key K_i. Data associated with each key are of no interest in the following discussion and are omitted. The operations to be performed, concurrently, on a B-tree structure will be of three kinds: search, insert, and delete. The processes that perform these operations are called read process (RP), insertion process (IP), and deletion process (DP), respectively. IPs and DPs are collectively called *updaters*.

Most of the solutions to the problem of supporting concurrent operations in B-trees make use of the following observations. There exists a node which is the root of a subtree, above which no change in data and structure due to an update can propagate. This node is called a *safe* node [2]. A node consisting of less than $2M$ children is called *insertion-safe* (i-safe), because a new key can be added without forcing a split. A node with more than M children is called *deletion-safe* (d-safe), because a key can be deleted without going below the M-children minimum. The portion of the access path from the deepest safe node to a leaf is called the *scope* of the updater. The child of the updater's deepest safe node on its scope is called the *highest unsafe node*.

RELATED RESEARCH

A number of solutions have been proposed for handling the concurrency control problem on B-trees. Locking techniques [32, 2, 27, 15, 21, 28] require each process to lock a node before it is accessed; appropriate lock relations guarantee the correctness of each operation. In the first solution [32], only one lock type is used, the exclusive lock, regardless of the operation to be performed, while the other solutions provide at least two different lock types to be used by searchers and updaters. None of these solutions, however, permits concurrency among insertion or deletion processes. Lehman and Yao presented an elegant solution in which many updaters may simultaneously access the tree [23]. This, however, is done in such a way that (1) no data sharing among processes is allowed and (92) successive deletions may cause storage underutilization. An algorithm for compressing the tree in the previous solution is considered in [33]. An optimistic technique [20] has been proposed to handle concurrency on B-trees [19], which is not an efficient method when conflicting operations are likely. Finally, Lausen has proposed a solution which switches from locking to the optimistic method when conflicting operations are rather seldom [22]. The drawback of this method is that the locking protocol that is used is the one proposed in [32], an inappropriate solution in that all processes (including readers) use exclusive locks.

3. LOCKING SCHEME AND STORAGE MODEL

The protocol uses four types of locks, viz. read-lock, insert-lock, delete-lock, and exclusive-lock (r-, i-, d-, and e-lock respectively). Let *rl*, *il*, and *dl* be the numbers of r-, i-, and d-locks, respectively, currently placed on a node. The compatibility and convertibility relations are given in Figure 2. The basic idea of the protocol is that a node with s children could accept $2M - s$ i-locks by $2M - s$ IPs and still be a safe node for all of them. In other words, we permit as many IPs to place locks on a node as possible to that their collective action will not require the node to be split. Likewise, a node with s children can accept $s - M$ d-locks by $s - M$ DPs and still be safe for all of them. Readers, on the other hand, are free for reading nodes that belong to a subtree being updated by an insertion or deletion process. i- or d-locks do not give the right to an updater to modify a node. They are used as reservations of free slots in the node. *Should an actual modification be required, they should be converted to e-locks*, a technique used also in [2, 21]. An important point that can not be seen from these tables is that a request to i- or d-lock a node implies also a request for an r-lock; e.g., if an i-lock request for node N is granted, N is both i- and r-locked. When the reading is terminated, the updater must explicitly r-unlock the node. Clearly, in this protocol compatibility relations among locks are not static. The lock assignment on a node does not depend exclusively on the lock type, but also on the status of the node and the kind and number of processes acting currently on that particular node.

	r	i	d	e
r	T	T	T	F
i	T	$il < 2M\text{-}s$ or $il = 0$	F	F
d	T	F	$dl < s\text{-}M$ or $dl = 0$	F
e	F	T	T	F

(a)

	r	i	d	e
r	F	F	F	F
i	F	F	F	T
d	F	F	F	T
e	F	F	F	F

(b)

Fig. 2. (a) compatibility relations among locks (the table is asymmetric). (b) convertibility relations among locks.

STORAGE MODEL

We assume that the nodes of the tree are stored on disks. The disk is partitioned into blocks which are the unit of both storage allocation and data transfer. The size of the node is the same as the block size. The main memory is shared by all processes. Each process may have its own private space, but this has nothing to do with its right to read or modify a node in main memory. Reads and updates of a node may be performed on a *single* copy in main memory. This model is in contrast to the one in [23], where all processes should copy the block which they want to read or write on its own private workspace; e.g., in their model, if at some point in time n readers are reading anode, there are n copies of the same disk block in main memory.

The following modification of the basic B-tree structure is required to accommodate our solution. We assume that two auxiliary pairs exist on each node, as shown in Figure 3. The (K_r, P_r) and (K_l, P_l) pairs of each node are called the *right* and *left link pair*, respectively. The link pairs are used by IPs as an additional method of reaching a newly created node; readers do not depend on the values of the link pairs for their operation. The technique of using additional paths to reach a node is also used in [23], where a link pair on each node always points to the node's right sibling, and in [26] for recovery purposes.

As we will see when we discuss the deletion process, it is very easy to parametrize the threshold, say tm, for merging or rotation. We may define any value of tm such that $0 \leqslant tm \leqslant M$. Assigning a value of tm less than M will further increase concurrency because more DPs may place d-locks on a node and more IPs may place i-locks on a node with fewer pairs. If $tm = 0$, all DPs will reach the leaf node without ever trying to reorganize the tree. This may be useful when (1) it is known *a priori* that deletions are not directed to a small part of the tree, (2) the tree is young and, presumably, it is growing. Notice also, that when $tm = 0$ we may have empty leaf nodes and the protocol behaves almost in the same way as the one presented in [23], where a deletion process always deletes the pair from a leaf regardless of its population. Our solution,

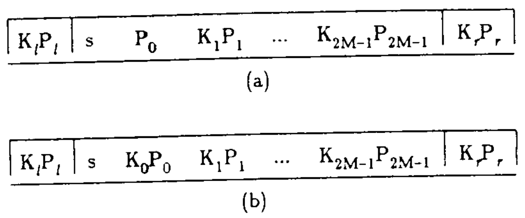

Fig. 3. (a) non-leaf and (b) leaf node with left and right link pairs.

however, has the benefit that this is done in a controlled way; that is, we may dynamically change *tm* to achieve the desired space utilization.

NOTATION

The pair $(K_{i,A}, P_{i,A})$, $0 \leqslant i \leqslant 2M-1$, explicitly declares the ith key and pointer of node A. A request to convert an i- or d-lock on A to an e-lock is expressed by the notation convert(x $\rightarrow$ e,(IA), where x is i or d. We use the notation E-lock(x, A) to mean the following three *individual* lock requests: x-lock(A), r-unlock(A), convert(x $\rightarrow$ e, A), where x is i or d. Finally, function Scan(A,(k) reads node A, as if it had no link pairs, and returns the pointer to the appropriate child for an argument key k.

4. SEARCHING

A read process searches the tree for a key k and returns the r-locked leaf node on which k may exist, as follows:

```
S1  [lock root]
    r-lock(root) and set $A := root$
S2  [leaf is found]
    if $A$ is leaf then **return** $A$.
S3  [lock child, unlock parent]
    $C :=$ Scan($A, k$), r-lock($C$), r-unlock($A$), $A := C$, repeat from step 2.
```

This simple lock-coupling technique [2] is sufficient for the correctness of the RP's operations.

5. INSERTION

An insertion process uses i-locks on its passage to a leaf node. Since i-lock implies r-lock, it may read the node with no other control. In each node, the IP checks for the node's safeness, and if the node is safe, it i-unlocks all the ancestors. An r-lock, on the other hand, is released immediately after reading a node, regardless whether the i-lock is kept or not. The outline of the insertion process follows:

ALGORITHM **FindLeafToInsert**. It searches the tree rooted at A for an argument key k; it reorganizes the tree, if necessary, and returns an e-locked nonfull leaf node in which k should be inserted;

```
I1   [lock root]
     set $A :=$ root, i-lock($A$), push(stack $S, A$)
I2   [$A$ is leaf and i-safe]
     if $A$ is leaf and not full then r-unlock($A$), convert(i $\rightarrow$ e, $A$), **return** $A$
I3   [$A$ is leaf and full]
     if $A$ is leaf and full then
I3.1    r-unlock($A$), invoke leaf := **Restructure**($S, k$)
I3.2    E-lock($i$,leaf), e-unlock all e-locked nodes except leaf,
        **return** leaf
I4   [find and lock child]
     $C :=$ Scan($A, k$), r-unlock($A$), i-lock($C$),
I5   [check if child has been split]
     $C :=$ **CheckSplit**($C, k$), $A := C$
I6   [$A$ is i-safe]
     if $A$ is not full then
     while $S$ not empty pop($S$), i-unlock $A$'s ancestors
I7   push($S, A$), repeat from step I2.
```

For the majority of the cases the leaf node is not full and the IP stops at I2. If, however, the leaf is full, the **Restructure** routine reorganizes the tree by initially splitting the leaf node. Since a new node has been created for the leaf node, a new pair should be added on leaf's parent, and if the parent is full, the overflow propagates until the deepest safe node. Since many IPs may operate on the same node simultaneously and because of the just-mentioned upward reorganization, appropriate precautions should be taken against possible concurrency anomalies while another IP goes down the tree to find the proper leaf node. This task is undertaken by the **CheckSplit** routine, whose internals will be explained later.

RESTRUCTURING PHASE

On reaching the full leaf node the IP's scope will be i-locked, but still free for other IPs and RPs. The IP, instead of e-locking its (already i-locked) scope, uses a technique similar to the side branching one, reported in [20] for binary trees and [21] for B-trees, as follows. Let C be the full (of pairs) leaf node. The IP gets a new node B from the free storage (assume $P_r = P_l =$ NIL for all new nodes), and examines the following possibilities:

If $k > K_{M,C}$ (B will be a right branch), the M rightmost pairs of C, (K_M, P_M) through (K_{2M-1}, P_{2M-1}), are read into B.

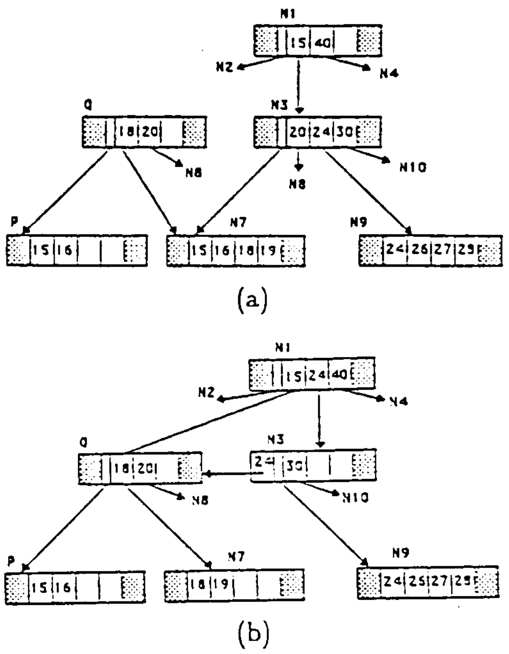

Fig. 4. Tree reorganization in order to insert key 17.

If $k \leqslant K_{M,C}$ (B will be a left branch), the M leftmost pairs of C, (K_0, P_0) through (K_{M-1}, P_{M-1}), are read into B.

The splitting propagates upwards in a similar way, i.e. reading half of the full node, with the additional operation of adding the newly created node, on the lower level, as well as the new separator. The new separator, which should be added to C's parent, is always the K_M key of C before its splitting, regardless of the value of key k and the direction of B.

Figure 4(a) shows the generation of two left branches, P and Q, in order to insert key 17 in the tree of Figure 1(a). Similarly, Figure 5(a) shows the generation of a left (P) and right branch (Q) in order to insert key 29 in the tree of Figure 1(a). It is noteworthy that while the IP creates the side branches (costly operations, since the allocation of a free block may require two disk accesses), no node in the original tree needs to be exclusively locked. Also, the side branches impose no additional space overhead, since a full node has to be split anyway.

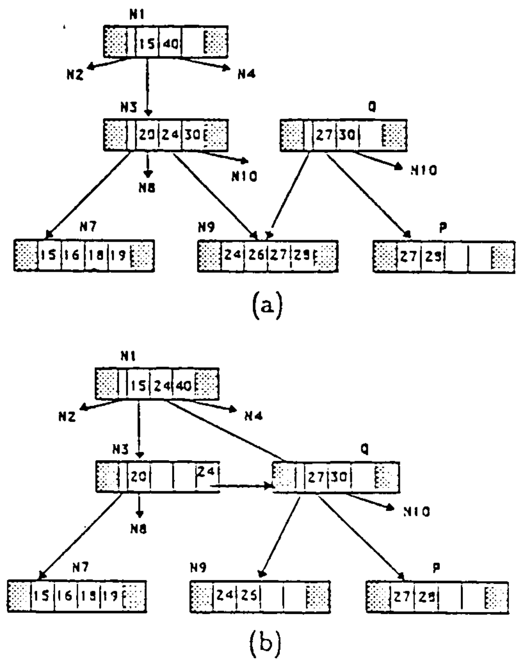

Fig. 5. Tree reorganization in order to insert key 29.

The update of the deepest safe node is performed as follows:

ALGORITHM **UpdateDeepestSafe**. It updates the deepest safe node A, given its unsafe childe C, and the newly created sibling B of C. In case that there is no such deepest safe, node A is passed with value NIL [i.e., C is the full (of pairs) root node].

```
U1    [deepest safe node $A$ exists]
      if $A \neq$ NIL then
U1.1    convert($\mathrm{i} \rightarrow \mathrm{e}, A$), convert($\mathrm{i} \rightarrow \mathrm{e}, C$), add $B$ and $K_{M,C}$, the new separa-
        tor, on $A$
U1.2    if $B$ is a right branch then
        $(K_{r,B}, P_{r,B}) := (K_{r,C}, P_{r,C})$ and $(K_{r,C}, P_{r,C}) := (K_{M,C}, B)$
U1.3    if $B$ is a left branch then
        $(K_{l,B}, P_{l,B}) := (K_{l,C}, P_{l,C})$ and $(K_{l,C}, P_{l,C}) := (K_{M,C}, B)$
U1.4    remove left (if $B$ is left branch) or right (if $B$ is right branch) half of
U2    [$C$ is the full (of pairs) root node; see Figure 6(a), (b)]
      if $A =$ NIL then
```

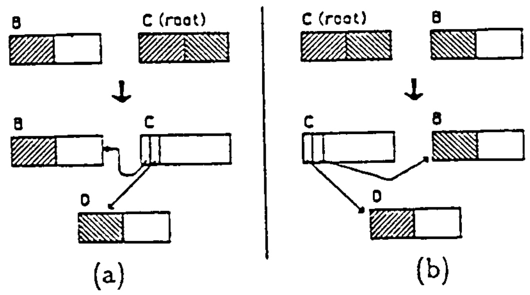

Fig. 6. Root splitting for (a) left and (b) right branch.

```
U2.1   convert(i → e, $C$), get a node $D$ from the free storage
U2.2a    if $B$ is right branch, copy left half of $C$ to $D$,
         set $P_{0,C} := D$, $(K_{1,C}, P_{1,C}) := (K_{M,C}, B)$
U2.2b    if $B$ is left branch, copy right half of $C$ to $D$,
         set $P_{0,C} := B$, $(K_{1,C}, P_{1,C}) := (K_{M,C}, D)$
U2.3   set counter of root $C$ to two
```

Note that if the root is full, its update is done in place; that is, its address never changes. Steps U1.2 and U1.3 are needed in order to guarantee that even a very slow IP will find its path. Assume the situation after an IP (say, IP_1) has removed the C's right half. Assume also another IP (say, IP_2) that, holding an i-lock on A before IP_1 updated A, wants to proceed to i-lock C because that's the proper node for the key that wants to insert. When IP_2 i-locks node C, half of C's pairs have been removed by IP_1. Even worse, it could be the case that other IPs insert a number of pairs in C, causing resplitting of C, perhaps before IP_1 has managed to read this node. IP_1 is able to find its path, through **CheckSplit**, which is described below:

ALGORITHM **CheckSplit**. Given an r- and i-locked node C and a key k, check whether C or one of its branches is the proper node for k.

```
C1  if ($P_r \neq$ NIL and $k \geq K_r$) then Branch := $P_r$
    else if ($P_l \neq$ NIL and $k < K_l$) then Branch := $P_l$
    else Branch := NIL
C2  [appropriate node found]
    if Branch = NIL then **return** $C$
C3  [transfer locks to branch node]
    i-lock(Branch), r- and i-unlock($C$), $C$ := Branch,
    repeat from step C1.
```

Since steps U1.1 and U1.3 actually insert a new node into a linear linked list containing some of A's children, we immediately have that *for any pair of nodes, x and y, which have the same parent, it cannot be the case that x is reachable from y and y is reachable from x*. Therefore, IPs cannot deadlock while moving locks on sibling nodes (step C3). Thus, we can always check whether a node has been split and if so, which node should be scanned in order to find the appropriate path. Note also that there is no need to set the link pair on nodes other than the highest unsafe, since all IPs should first read this node before they proceed to lower levels.

The remaining task fo the IP process is to proceed to lower level nodes and remove the appropriate half from those nodes. Its steps are summarized here.

ALGORITHM **Restructure**. It accepts the key k for insertion, and the i-locked IP's scope. It reorganizes the tree and returns a nonfull leaf node in which k should be inserted. After completion, all nodes in IP's scope are e-locked.

```
RI1  make branch for the leaf node, and call this node $A$
RI2  make branches upwards, until the highest unsafe node $C$
RI3  **UpdateDeepestSafe**
RI4  if $C$ is leaf then **return** $A$
RI5  convert(i $\rightarrow$ e, child of $C$), remove left or right half of $C$, reset $P_r$ and $P_l$
     of child to NIL, $C :=$ child
RI6  repeat from step 4
```

Figures 4(b) and 5(b) show the tree state after the **Restructure** routine is completed for the operations of the previous examples.

6. DELETION

There are two major characteristics of the way DPs work. First, reorganization of the tree is always done in a top down manner; secondly, each DP attempts to correct actions taken by other DPs, never its own. The idea of relaxing the responsibility of a process for finishing its own work is also discussed in [9, 10] for AVL and 2–3 trees, and in [25] for binary trees. On each level, the DP d-locks the proper node; examines whether this node has no less the *tm* pairs, where *tm* is the threshold for merging; and if so, r- and d-unlocks the parent. If it has less then *tm* pairs, it immediately unlocks this node, e-locks the parent, and invokes the **Reorganize** routine, which operates as follows:

ALGORITHM **Reorganize**. It accepts an e-locked node A and a key k; it reorganizes the subtree rooted on A and the child C for key k; after the reorganization, it returns the proper child C and its sibling B.

```
RD1  [find child]
     $C :=$ Scan($A, k$), E-lock($d, C$), $B :=$ NIL
RD2  [$C$ is d-safe]
     if $C$ has no less than $tm$ children return $C, B$
RD3  [$C$ is unsafe]
     $B :=$ an immediate sibling node of $C$, E-lock($d$, $B$)
RD4a   [merging]
       If total number of pairs in $C$ and $B$ is less than $2M$ then move all
       pairs of $C$ into $B$, delete child $C$ from $A$, FreeNode($C$), interchange
       $C$ and $B$
RD4b   [rotation]
       elseif $C$ is a leaf node then rotate $B$ into $C$
RD5  return $C$ and $B$
```

Our solution does not guarantee that every node has at least tm pairs because, as we see from the above algorithm, only merging is performed on nonleaf nodes. The fact, however, that two nodes cannot be merged implies that they have, collectively, more than $2M$ pairs and therefore space underutilization problems cannot result from this action. Rotations, on the other hand, may be performed on leaf nodes.

The problem of root is handled in a similar manner to IPs, in that the address of root does not change; simply the contents of the root are updated. This will happen iff the root has exactly one child. The update is done by moving all pairs from this unique child to the root (Figure 7).

ALGORITHM **ReorganizeRoot**. It fills up the root A with all the pairs of its unique child. The address of root does not change.

```
RR1  [e-lock the root]
     r-unlock($A$), convert(d $\rightarrow$ e, $A$),
```

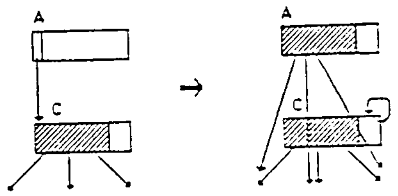

Fig. 7. Root update because of deletions.

```
RR2  [e-lock its unique child]
     $C := P_{0,A}$, E-lock(d, $C$), copy $C$ to $A$, set $P_r$ of $C$ to point to itself
```

Notice that the unique child of the root is not returned to the free storage; This is the only case where a node is lost "forever." We present now the deletion process.

ALGORITHM **FindLeafToDelete**. Given a tree A and a key k, it returns an e-locked leaf node from which k, if it exists, should be deleted. As it goes down, it readjusts nodes with less than tm pairs.

```
       [boolean ExL is true iff the parent node is e-locked]
D1     [d-lock root]
       make $A$ the root, d-lock($A$), set ExL := false,
       if root $A$ is leaf then r-unlock($A$) and convert(d $\rightarrow$ e, $A$)
D2     [adjust root]
       If root $A$ has one child $C$, and $A$ is not leaf then
       invoke ReorganizeRoot($A$), e-unlock($C$), ExL := true
D3     [leaf is found]
       If $A$ is leaf then return $A$.
D4     [find and lock child]
       $C$ := Scan($A, k$), d-lock($C$)
D5a      [child is d-safe]
         If child $C$ has no less than $tm$ pairs then
D5a.1      If $C$ is leaf then r-unlock($C$), convert(d $\rightarrow$ e, $C$)
D5a.2      [unlock parent]
           unlock node $A$, set $A$ to be $C$, set ExL to false
D5b      [child is not d-safe]
         If child $C$ has less then $tm$ pairs then
D5b.1      [unlock child, e-lock parent]
           r-unlock ($C$), d-unlock($C$); if not ExL then r-unlock $A$,
           convert(D $\rightarrow$ e, $A$)
D5b.2      [check node]
           If $P_{r,A} = A$ then e-unlock($A$) and repeat from step D1
D5b.3      [reorganize $A$]
           invoke Reorganize($A, k$), which returns child $C$ and its sibling $B$
D5b.4      e-unlock($A$), set $A$ to be $C$ and ExL to be true; if $B \neq$ NIL then
           e-unlock($B$)
D6     repeat from step D3
```

An example of a deletion process will be given in Section 8, where we discuss the recovery mechanism.

7. CORRECTNESS OF INTERACTION AND DEADLOCK FREEDOM

In order to prove correctness, we will *not* prove that log sequences of events produced by this protocol are serializable [11], because they are not.[1] The sequence of low level intermediate reads and writes is unimportant in this application as long as the *results* of the high level operations search, insert, and delete are consistent (equivalent to a serial execution). The distiction between serializability of log sequences of events and operation sequences is discussed also in [5, 12, 30]. We will prove that all processes can navigate correctly into the tree, in order to find the appropriate leaf node, given the modifications which are imposed by the updaters' reorganizations, and that they never deadlock. We do that by examining the synchronization achieved between the three types of processes.

In the following discussion, nodes A and C are the deepest safe and highest unsafe node, respectively. The time that a process P_i is granted an x-lock (x = r, i, d, or e) on some node N is expressed by $\mathrm{x}_i(N)$. Likewise, the notation $\mathrm{xu}_i(N)$ means the time that P_i x-unlocks node N.

READ/INSERT

IPs do not perform any modification during the searching phase, and for the part of the restructuring phase from leaf to the deepest safe node. Recall that, during that time, side branches are not yet linked to the tree and they are invisible to RPs. Therefore the compatibility of r- and i-locks does not create any problem for RPs. Visible modifications are performed after the IP e-locks node A. Let $e_1(A), e_1(C)$ be the time that IP_1 e-locks nodes A and C, respectively. Let also $r_2(A), r_2(C)$ be the times that an RP_2 r-locks nodes A and C, respectively. Because of the lock-coupling technique, we have either $r_2(A) < r_2(C) < e_1(A) < e_1(C)$ or $e_1(A) < e_1(C) < r_2(A) < r_2(C)$. In the first case RP_2 reads C before half of its pairs are removed, and in the latter RP_2 reads C after reading A, which already has been updated to include also the branch of C. Thus, in either case, the RP does not have to read the link pairs. In summary, once the IP e-locks the deepest safe node, readers cannot interleave on the tree rooted on this node.[2] Since an RP and an IP place their r- and e-locks,

[1]Assume for example a tree of height four and a path consisting of nodes A, B, C, D on levels 1 through 4. Assume also two DPs, DP_1 and DP_2, passing through the above nodes. DP_1 may rearrange nodes A, B before DP_2 reads these nodes, and DP_2 may bypass DP_1 and rearrange C, D before DP_1 reads C and D.

[2]Strictly speaking, this is required for recovery purposes only. The execution is still correct even if readers interleave with an IP in its scope, provided that the IP places its e-lock requests according to the lock-coupling technique.

respectively, on nodes visible to both in a top down way only, they cannot deadlock.

INSERT/INSERT

Clearly, the point that needs discussion is the IP's operations on the deepest safe node, since this is the case where many IPs may coexist on a node while updates take place. Assume that an IP, say IP_1, has already i-locked its scope and created all the side branches, and it is ready to update the deepest safe node A. Define $e_1(A), e_1(C)$ to be the time when IP_1 e-locks nodes A and C, respectively. Assume, also, the existence of another IP, say IP_2, whose path passes through nodes A and C (before its splitting). Let $i_2(A), i_2(C)$ be the times when IP_2 i-locks nodes A and C, respectively. Clearly, $e_1(A) < e_1(C)$ and $i_2(A) < i_2(C)$. Also, since node C is full, IP_2 cannot i-lock C before IP_1 unlocks this node, and thus we have $e_1(C) < i_2(C)$. Therefore, we have to examine the following two cases:

Case 1: $e_1(A) < e_1(C) < i_2(A) < i_2(C)$ *or* $e_1(A) < i_2(A) < e_1(C) < i_2(C)$. The important part of both of these inequalities is that $e_1(A) < i_2(A)$, which means that the side branch (say, B) has already been added to A when IP_2 reads this node and therefore IP_2 may access C or B in the usual way (reading A, the parent of both C and B).

Case 2: $i_2(A) < e_1(A) < e_1(C) < i_2(C)$. IP_2 reads A before IP_1 adds B in this node, and reads C after IP_1 removes the left or right part of C. In this case, IP_2 makes use of the link pairs to find its path via the **CheckSplit** routine. Note that if IP_2 delays placing its i-lock request for node C [after $e_1(C)$], other IPs may bypass IP_2, add pairs in C, and resplit this node. This could happen at most $2M - tm - 1$ times.[3]

IPs place their i- and e-locks in a top down manner, except when a branch node (on the same level) should be i-locked. However, it has been shown that no cycles are possible among nodes of the same level and thus IPs cannot deadlock.

READ/DELETE

DPs always e-lock the parent node before they e-lock and rearrange two of parent's children. Therefore, because of the lock-coupling technique, readers

[3]Every time a splitting of C is performed, a new pair is added to A: when A reaches the point where it has exactly $2M - 1$ children and one i-lock (the IP_2's lock), no other i-lock may be placed on A.

cannot interfere with DPs while updates take place. Readers and DPs place their r- and e-locks, respectively, in a top down way, and therefore they cannot deadlock.

DELETE/DELETE

There are two places where a DP performs its updates: (1) to update the root (step D2) and (2) to reorganize a node other than the root (step D5b.3). We examine those two cases.

Case 1: **ReorganizeRoot**. Assume a DP, say DP_1, d-locks the root A (step D1). Since the root has exactly one child, no other DP may have any kind of lock on the root. Let $e_1(A)$ and $e_1(C)$ be the times when DP_1 e-locks A and its unique child C and updates the root. Conflict may arise when another DP, say DP_2, holds a d- and an r-lock on C and one of C's children, say D. If D is unsafe, DP_2 r-unlocks C, $ru_2(C)$, and converts its d- to an e-lock on C, $e_2(C)$. Although the schedule $ru_2(C)e_1(C)e_2(C)$ is not correct, DP_2 recovers by first checking whether C is part of the tree (step D5b.4) and if it is not, it unlocks C and starts again from the root. No other interaction can create problems on the root.

Case 2: **Reorganize**. This subprocess starts working with the parent node A being e-locked. All subsequent operations are done after child nodes, C and B, have been e-locked, at $e_1(C)$ and $e_1(B)$ respectively (steps, RD1 and RD3). Also, this subprocess does not depend on values passed by the parent process for their operations; i.e., it is its own responsibility to find the child for some key, the sibling, etc. Therefore, here we have a *serial* execution between those subprocesses that act on a subtree rooted on the same node. The order of the serial execution is determined by the order in which e-locks are placed on the parent node A. The same is not true when **Reorganize** subprocesses work on different levels. Assume a DP_2 that holds a d- and an r-lock on B as well as on one of B's children, say D. If D has less than tm pairs, DP_2 unlocks D and then r-unlocks B, $ru_2(B)$, and converts its d- to an e-lock on B, $e_2(B)$, in order to reorganize B and C. Again, the schedule $ru_2(B)e_1(B)e_2(B)$ is not correct. However, no pair has been deleted from B; simply all pairs from C have been moved to B. Since the **Reorganize** subprocess invoked by DP_2 reads node B after $e_1(B)$, no harm can be resulted from DP_1's action on B.

INSERT/DELETE

Operations performed by IPs or DPs which modify the tree are done on their scope which is i- or d-locked, respectively. Since these locks are not compatible, there is no interference among these processes.

8. RECOVERY

We know of two conditions that should be satisfied so that a locking protocol be recoverable: (1) the locking granularity must be at least as fine as the recovery granularity, and (2) objects updated by a transaction T_i must be unavailable to other transactions until T_i commits (to avoid cascading roll-backs). While the first condition is satisfied, the latter is not. That is, we would like to unlock updated nodes on higher levels on the tree before the transaction actually updates the leaf node and commits, without sacrificing recoverability.

Operations acting on a *B*-tree are actually a part of a longer transaction running on the database. For example, we insert a key and a pointer to a *B*-tree because we insert a tuple on a relation indexed by this tree. We expect a transaction to be atomic, that is, either happen or have no effect at all [14, 17]. For instance, in the above example we do not want to have either the index updated with the new pair without the tuple in the relation or the tuple in the relation without the corresponding pair in the tree. In general, therefore, tree updates must be revocable until the transaction that calls for those updates commits. Let's, however, distinguish the operations of the insert or delete processes into two categories. In the first category belong operations that reorganize the tree, and in the second, the action of inserting (deleting) a specific pair in (from) the leaf node. Although those operations have been initiated by the same transaction, they are not logically related. That is, actions of the first category may be committed regardless of the fact that the transaction that initiated the insertion or deletion may later abort. Consider the case of insertion after the tree is reorganized and before the actual insertion on the leaf is performed. If we commit those updates, the tree is consistent no matter what will happen to the parent transaction (the one that called the IP). Similarly, if a DP reorganizes a part of the tree, these updates can be committed; the reorganization on that subtree has nothing to do with the lack either of the parent transaction or of the delete process itself on a lower level in the tree. To summarize, there will never be a need to undo updates that have to do with the reorganization of the tree.

In order to implement the above idea, we need a notion similar, but not identical, to the transaction "save point" [14]. We want the data manager to be able to perform a *subcommit*(T_i) action, which is defined as follows:

All updates made by T_i become permanent (not revocable), and no lock held by this transaction is released. The parent transaction is the one which determines when locks held by child transactions will be released.

To be more specific, *subcommits* must be issued when the following sub-transactions terminate: **Restructure** (step RI4), **ReorganizeRoot** (step RR2), and **Reorganize** (step RD5). Since updates become permanent, e-locks may be released to increase concurrency. Which locks, and when they will be released, depends on the transaction that invoked the above subtransactions. The insertion keeps the e-lock on the leaf and unlocks all the other nodes (step I3.2); the deletion keeps the e-lock on the updated child and unlocks the parent (step D5b.4). In either case, insert or delete, an e-locked leaf node is returned to the parent transaction. If this parent transaction follows the 2PL protocol, the leaf will be unlocked at commit time. If the transaction gets into trouble, it is sufficient to back up to the immediate preceding *subcommit* rather than undoing all the work.

EXAMPLE. Assume that the threshold for merging *tm* is set to M, in the tree of Figure 1(e) (i.e., $tm = 2$), and that the following operations have been performed in this tree: delete 38; delete 42; delete 46; delete 40, which, according to the algorithm, also merges N11 into N12 before deleting 40. Figure 8(a) shows the resulting tree after the above operations. Now, suppose that transaction T_1 have been initiated to perform the following operations:

delete tuple (44, Alex, Boston) from the relation R(id, name, city) indexed by the tree of Figure 8 on the first attribute, and then do some "other work."

The interaction of T_1 with the various subtransactions are illustrated in Table 3.

Note the difference between the scheme described and the nested transaction model [31, 29]. The latter model permits transactions to be nested; however, it requires that transactions on the same level be serializable, which is not the case here. For example, although we require the **Reorganize** subtransactions to be

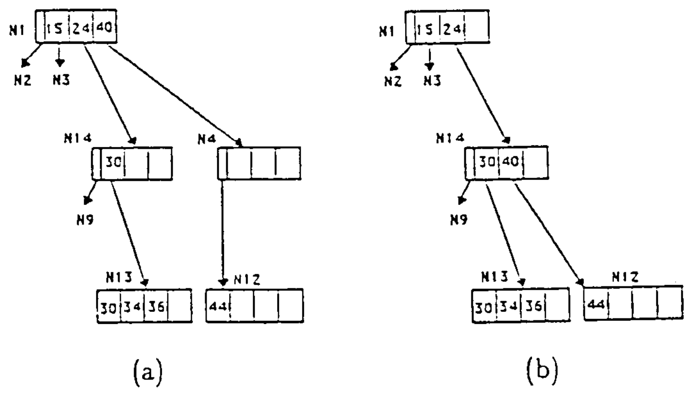

Fig. 8. A tree for the example of section 8.

TABLE 3

Transaction T_1	Subtransaction $T_{1,1}$ **FindLeaf**(k)	Subtransaction $T_{1,2}$ **Reorganize**(N, k)
invoke $T_{1,1}$ with $k = 44$		
	d-lock N1	
	find proper child of N1 for 44	
	[child is N4]	
	d-lock N4, N4 has less then 2 pairs	
	r- and d-unlock N4, r-unlock N1,	
	convert(d → e, N1)	
	invoke $T_{1,2}$ with N = N1 and $k = 44$	
		e-lock N4 and N14
		merge N4 into N14
		delete N4 from N1, Free(N4)
		return child-N14, sibling = N4
		and subcommit [see Figure 8(b)]
		[$T_{1,2}$'s updates are now irrevocable]
	e-unlock N1 and N4	
	find proper child of N14 for 44	
	[child is N12]	
	d-lock N12, N12 is leaf and	
	has less than 2 pairs	
	r- and d-unlock N12, r-unlock N14,	
	convert(d → e, N14)	
	invoke $T_{1,2}$ with N = N14 and k = 44	
		e-lock N12 and N13
		rotate N13 into N12 through N14
		return child = N12, sibling = N13
		and subcommit
		[$T_{1,2}$'s updates are now irrevocable]
	e-unlock N14 and N13	
	return N12	
	[at this point only the leaf node,	
	N12, is e-locked]	
delete (N12, 44)		
delete tuple (44, Alex, Boston) from R		
[do "other work"]		
commit and release all e-locks		

serializable, the DPs that are calling them are not. A paper by Moss, Griffeth, and Graham [30] presents a thorough discussion of this concept.

9. PROTOCOL APPLICABILITY TO SIMILAR STRUCTURES

Since this protocol is so tight to the specifics of the B-tree structure, we might not expect to apply it to other similar structures "as it is," without sacrificing correctness and/or performance. For some structures, it will be required to modify the algorithms for the three operations (which is natural anyway, since they act on a different tree); for some others, however, the locking rules and/or the definition of i- and d-safeness must be changed. Moreover, there is nothing to be gained if we apply this protocol to structures in which we know *a priori* that updaters modify all nodes on their access path, from root to the leaf node (e.g. OB-trees [36]). In the following, we discuss the appropriate modifications for B^+-trees, compressed B-trees [7], and R-trees [16].

B^+-TREES

Leaf nodes belonging to a B^+-tree have pointers pointing to their right sibling. The P_r pointer of the right link pair can also be used for this purpose. The modification that is required is to restrict the splitting of leaf nodes to right branches only and set the P_r to the right branch even when the leaf is not the deepest safe node. The locking rules are the same as in B-trees.

COMPRESSED B-TREES

Key and pointer compression on a B-tree have been utilized as a mean to increase the capacity of each node and therefore decrease the retrieval cost. We examine the case of key compression only. Instead of defining the safeness of a node N as a function of the number of pairs stored in this node, we may use another storage unit, e.g. a byte, and say that N is i-safe iff b storage units can be inserted without forcing splitting of N. An i-lock request must have the form i-lock(N, b), where b is the number of storage units that a key occupies. Assume, for example, that the storage unit is one byte, a pointer occupies four bytes, and each tree node can store from M up to $2M$ bytes. Let s be the number of bytes that are already occupied by keys and pointers on a node, and il the number of bytes reserved by IP processes already holding an i-lock on that node. Each time the lock manager accepts an i-lock(N, b), il is set to $il + b + 4$ for the node N. The criterion for assigning an i-lock will be the following:

A request of an IP to i-lock a node (with associated storage cost b) is granted iff the node is not e- or d-locked and

$$il + b + 4 < 2M - s \quad \text{or} \quad il = 0.$$

Similar locking rules may be applied for DPs.

R-TREES

R-trees are similar to *B*-trees, and they may be used for multidimensional (spatial) searching. Leaf nodes contain index records of the form (e, tuple-identifier) where tuple-identifier refers to a ruple in a database and e is the extend of the object indexed, the extend being the smallest rectangular surrounding the object. Index records at higher levels are of the form (e, N), where e is the extend of the extend of all the objects pointed at by the indexes of the node N one level below (informally, the node's extend). Insertions of new objects may affect higher nodes even when the leaf node is not full (because the extend must be updated). Thus, the definition of the i-safeness of a node should be modified as follows:

A node is i-safe when it contains less than $2M$ pairs and the extend of the node covers the extend of the object being inserted.

The locking rules are the same as in *B*-trees.

10. CONCLUSION

We have given algorithms and locking rules to manipulate the concurrency control problem in *B*-trees, and we have discussed the appropriate modifications which are required so that they can be used for some variants of the *B*-tree such as B^+-trees, compressed *B*-trees, and *R*-trees.

The algorithms introduce some concurrency control overhead with respect to the number of messages required to be sent by an updater to the lock manager; for example, an insertion process has to send three messages in order to e-lock a node, while in other protocols this number is lower, e.g. two in [2, 21] and one in [32, 23, 33]. The storage model introduces some space overhead by requiring each node to have two additional pairs (the link pairs), but as the order of the tree increases this overhead becomes insignificant. We believe the protocol presented in this paper enables a higher degree of concurrency by allowing a number of insertion or deletion processes to operate concurrently on a node. This effect is achieved by using an operation-specific locking mechanism in which each of the three processes (read, insert, delete) uses different lock types to reach the leaf node. It also permits data sharing among processes and thus eliminates the cost associated with storing multiple copies of the same disk block in main memory.

Finally, we have given a specific and simple recovery mechanism which permits us to unlock nodes updated by some transaction before this transaction commits.

REFERENCES

1. R. Bayer and E. McCreight, Organization and maintenance of large ordered indexes, *Acta Inform.* 1(3):173–189 (1972).
2. R. Bayer and M. Schkolnick, Concurrency of operations on *B*-trees, *Acta Inform.* 9(1):1–21 (1977).
3. G. Buckley and A. Silberschatz, Beyond two-phase locking, *J. Assoc. Comput. Mach.*, 32(2):314–326 (Apr. 1985).
4. P. A. Bernstein and N. Goodman, Concurrency control in distributed database systems, *ACM Comput. Surveys* 13(2):185–221 (1981).
5. P. Bernstein, N. Goodman and M. Lai, Analyzing concurrency control algorithms when user and system operations differ, *IEEE Trans. Software Engrg.* SE-9(3):233–239 (May 1983).
6. A. Biliris, Concurrency control on Database Indexes: Design and Evaluation, Ph.D. Thesis, EECS Dept., George Washington Univ., June 1985.
7. D. Comer, The ubiquitous *B*-tree, *ACM Comput. Surveys* 11(2):121–137 (1979), 11(4):412 (1979).
8. A. Croker and D. Maier, A dynamic tree-locking protocol in *Proceedings of the Second IEEE International Conference on Data Engineering*, Los Angeles, Calif., Feb. 1986, pp. 49–57.
9. C. Ellis, Concurrent search and insertion in AVL trees, *IEEE Trans. Comput.* C-29(9):811–817 (Sept. 1980).
10. C. Ellis, Concurrent search and insertion in 2-3 trees, *Acta Inform.* 14(1):63–86 (1980).
11. K. Eswaren, J. Gray, R. Lorie and I. Traiger, The notions of consistency and predicate locks in a database system, *Comm. ACM* 19(11):624–633 (Nov. 1976).
12. R. Ford and J. Calhoun, Concurrency control and the serializability of concurrent tree algorithms in *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Waterloo, Ontario, Apr. 1984, pp. 51–60.
13. H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database, *ACM Trans. Database Systems* 8(2):186–213 (June 1983).
14. J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, The recovery manager of the system R database manager, *ACM Comput. Surveys* 13(2):223–242 (June 1981).
15. C. S. Guibas and R. Sedgewick, A dichromatic framework for balanced trees, in *Proceedings of the* 19*th Annual Symposium on the Foundations of Computer Science*, 1978, pp. 8–21.
16. R. Guttman, *R*-trees: A dynamic index structure for spatial searching, in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Boston, June 1984, pp. 47–57.
17. T. Haerder and A. Reuter, Principles of transactions-oriented database recovery, *ACM Comput. Surveys* 15(4):287–318 (Dec. 1983).
18. Z. M. Kedem and A. Silberschatz, Locking protocols: From exclusive to shared locks, *J. Assoc. Comput. Mach.* 30(4):787–804 (Oct. 1983).

19. M. L. Kersten and H. Tebra, Application of an optimistic concurrency control method, *Software Practice and Experience*, 14(2):153–168 (Feb. 1984).
20. H. T. Kung and P. L. Lehman, A concurrent database manipulation problem: Binary search trees, *ACM Trans. Database Systems* 5(3):339–353 (1980).
21. Y. Kwong and D. Wood, A new method for concurrency in *B*-trees, *IEEE Trans. Software Engrg.* 8(3):211–222 (1982).
22. G. Lausen, Integrated concurrency control in shared *B*-trees, *Computing* 33(1):13–26 (1984).
23. P. L. Lehman and S. B. Yao, Efficient locking for concurrent operation on *B*-trees, *ACM Trans. Database Systems* 6(4):650–670 (1981).
24. N. A. Lynch, Multilevel atomicity—a new correctness criterion for database concurrency control, *ACM Trans. Database Systems* 8(4):484–502 (Dec. 1983).
25. U. Manber and R. E. Ladner, Concurrency control in a dynamic search structure, *ACM Trans. Database Systems* 9(3):439–455 (Sept. 1984).
26. D. A. Menasce and O. Landes, Dynamic crash recovery of balanced trees in *Proceedings of the Symposium on Reliability in Distributed Software and Database Systems*, Computer Science Press, July 1981, pp. 131–137.
27. R. Miller and I. Snyder, Multiple access to *B*-trees, in *Proceedings of the 12th Annual Conference in Information Science and Systems*, Mar. 1978.
28. Y. Mond and Y. Raz, Concurrency control in B^+-tree databases using preparatory operations in *Proceedings of the 11th International Conference on Very Large Databases*, Stockholm, Aug. 1985, pp. 331–334.
29. J. E. B. Moss, Nested transactions: An approach to reliable distributed computing, Ph.D. Thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Inst. of Technology, Apr. 1981.
30. J. E. B. Moss, N. D. Griffeth and M. H. Graham, Abstraction in recovery management, in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, May 1986, pp. 72–83.
31. D. Reed, Naming and synchronization in a decentralized computer system, Ph.D. Thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Inst. of Technology, June 1978.
32. B. Samadi, *B* – trees in a system with multiple users, *Inform. Process. Lett.* 5(4):107–112 (1976).
33. Y. Sagiv, Concurrent operations on *B*-trees with overtaking, in *Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Portland, Ore., Mar. 1985, pp. 28–37.
34. A. Spector and P. Schwartz, Transactions: A construct for reliable distributed computing, *Oper. Systems Rev.* 17(2):18–35 (Apr. 1983).
35. A. Z. Spector, J. Butcher, D. S. Daniels, D. J. Duchamp, J. L. Eppinger, C. E. Fineman, A. Heddaya, and P. Schwarz, Support for distributed transactions in the TABS prototype, *IEEE Trans. Software Engrg.* SE-11(6):520–530 (June 1985).
36. M. Stonebraker and L. A. Rowe, Database portals: A new application program interface, in *Proceedings of the 10th International Conference on Very Large Databases*, Singapore, Aug. 1984, pp. 3–13.
37. G. Weikum, A theoretical foundation of multi-level concurrency control, *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Boston, Mar. 1986, pp. 31–42.
38. M. Yannakakis, C. H. Papadimitriou, and T. H. T. Kung, Locking policies: Safety and freedom from deadlock, in *Proceedings of the 20th IEEE Symposium on Foundations of Computer Science*, Oct. 1979, pp. 286–297.

Received 1 August 1987; revised 8 december 1987