INFORMATION SCIENCES **79,** 73–88 (1994)

B-Trees with Lazy Parent Split

YANNIS MANOLOPOULOS*

Computer Science Department, University of Maryland, College Park, Maryland 20742

ABSTRACT

A B-tree variant that postpones parent node splittings due to upcoming items until a later access of the same node is examined. This technique aims to decrease the possibility of propagating splittings to upper levels so that more concurrency is achieved. Insertion and deletion algorithms are given. Time and space performance results are also reported and comparison with conventional B-trees is carried out. It is shown that this technique substantially improves the performance of small degree B-trees so that, indeed, concurrency is enhanced.

1. INTRODUCTION

B-trees and related trees are very popular structures for use in file organizations and database management systems [17]. Since the very first appearance of the basic structure in the literature by Bayer in 1972 [2], numerous variations, which are summarized in [4] and [6], have been proposed.

In a multiuser environment, there is no doubt that locks should be minimized so that many processes can run simultaneously. Search operations do not change the structure and the content of a B-tree. Therefore, search operations do not lock the tree nodes exclusively. However, the case of update operations is different. According to the well known algorithm [2], during an insert or a delete operation, first a path toward the leaf level is followed and then the same path may have to be followed toward the root in order to perform successive splittings or mergings, respectively. Therefore, on the way down to the leaves, some critical nodes will have to be locked by the updaters, apparently because there is a possibility that they may change later. On the other hand, according to the

* On leave from the Department of Informatics, Aristotle University, Thessaloniki, Greece 54006.

655 Avenue of the Americas, New York, NY 10010 0020-0255/94/$7.00

dichromatic framework, which unifies balanced trees, splitting and rebalancing is performed in a top-down manner whenever a full node is encountered [7]. In this algorithm, locks are unavoidable too.

In this work, a variant of B-trees is examined which aims at decreasing the number of critical nodes (which have to be locked) so that more concurrency is achieved. The original idea of this variant was reported by Keller and Wiederhold [9], where it was anticipated that delaying a full parent node splitting would offer greater concurrency in B-trees with variable length entries.

The remainder of this work is organized as follows. The next section gives the basics (definitions and notions) with regard to B-trees and concurrency. The third section describes the variant and shows the basic differences compared with the conventional tree structure and some other variants. Maintenance algorithms and characteristic examples are given in the fourth section. Qualitative and experimental time and space performance results are discussed in the fifth section. Finally, it is concluded that the technique of delaying parent node splittings improves the space and time performance of small degree B-trees.

2. BASIC BACKGROUND

A B-tree of degree d is defined as a multiway tree with the following properties:

(a) The root has from 1 to $2d$ items.
(b) All nodes except the root have from d to $2d$ items.
(c) A node with k items has $k+1$ children.
(d) All leaves are at the same level.

If $d=1$, then every node has two or three children and thus the 2–3 tree is produced. In a similar manner, nodes of the 1–2 tree may have one or two children. These specific trees and some variations of them are designed for use in a one level store and have been extensively examined and analyzed in the past [6]. B-trees are maintained with complex insertion and deletion algorithms (during which node splittings and mergings, respectively, are performed), so that the foregoing properties are fulfilled.

Certainly, in a multiuser environment, care must be taken so that operations may be performed simultaneously [13]. This problem has been studied in depth, not only with respect to the family of B-trees, but also for binary search trees, AVL trees, linear hashing, etc. Biliris provides a long list of pointers to all the relevant work [3]. Srinivasan and Carey have reported an extensive simulation comparing the performance of many B-tree variations with regard to the concurrency achieved [16]. More

recently, Johnson and Shasha have reported a synthetic presentation on the subject and provided a thorough analysis and simulation in the same respect [8]. In the present paper, we do not give concurrency algorithms, but rather a structure that enhances concurrency. All concurrency techniques that have appeared in the literature may be applied in our structure proposal. In this way, our structure may augment the set of techniques examined in [8] and [16].

It is evident that search operations do not change the structure and the content of any file organization Thus, read-locks do not prevent the performance of other search operations. However, the case of update operations in a B-tree, e.g., insertions and deletions, is different. According to the classical algorithms, first a top-down traversing procedure toward the leaf level is performed, but then the same path toward the root may have to be followed to perform successive bottom-up splittings or mergings. Therefore, on the way down to the leaves, some nodes will have to be write-locked by the updaters, e.g., they will have to be unavailable for subsequent operations. The obvious reason is that there is a possibility that these nodes may change later due to side effects of this operation. These nodes are called critical nodes. In the extreme case, splittings or mergings may propagate all the way up toward the root, expanding or shrinking the tree by one level, respectively, and, therefore, an entire path will have to be locked. According to the dichromatic framework of red-black trees proposed by Guibas and Sedgewick [7], splitting and rebalancing is performed in the top-down manner whenever a full node is encountered. More recently Nurmi and Soisalon-Soisinen uncoupled updating and rebalancing in red-black trees, resulting in smaller numbers of locked nodes [11]. However, in this algorithm locks are unavoidable too. The following section provides a literature survey on the subject.

3. LITERATURE SURVEY

It is certain that response time grows in a multiuser environment because of locking. Many B-tree variants along with searching, insertion, and deletion algorithms have been proposed so that the smallest possible number of critical nodes are locked during these operations. The variant examined in the present work is based on an idea reported in [9], which suggested delaying splitting full parent nodes in order to improve concurrency in a B-tree with variable length entries. This idea of delaying splitting full parent nodes was reported in an even earlier work [10].

Let us first present the work by Nurmi et al. [10]. In this work, a B+-tree is examined, i.e., all items reside in leaf nodes while upper levels

contain keys and pointers that serve as an index. Therefore, insertions and deletions have to be performed at the leaf level. If a leaf node overflows due to an insertion, then two new half-full nodes are created and connected one level below the previous leaf node. Thus, the latter node becomes part of the index containing only one comparison key, two pointers, and one bit flag denoting this relaxed balance. If a leaf underflows due to a deletion, then, temporarily, no item redistribution or node merging is triggered. In both cases, side effects are not propagated on the path toward the root. In [10], three methods to alter this relaxation at a later instance, aiming at limiting the number of nodes which have to be locked, are described. Three negative characteristics of this approach are that (a) in case of deletion, leaf nodes may be left with less than 50% storage utilization, (b) in case of insertion, the extra node, which may have to be created, is greatly underutilized (containing only a few data), and (c) in case of searching, the path from the root to the leaves is elongated by one additional node.

Let us now present the technique proposed by Keller and Weiderhold [9]. If a leaf node overflows, then a new node is created and both nodes are half full. In addition, the middle item has to be posted to the parent node. This item is actually stored in the parent node only if there is available space and, thus, the process terminates according to the usual B-tree insertion algorithm. If the parent node is full, then the middle item is stored in one of the two nodes at the leaf level, which are called left and right siblings. Figure 1(a) (as appears in [9]) shows a B-tree with a full parent node, whereas the result of an item insertion in a leaf node is depicted in Figure 1(b). The connection of sibling nodes is shown with a

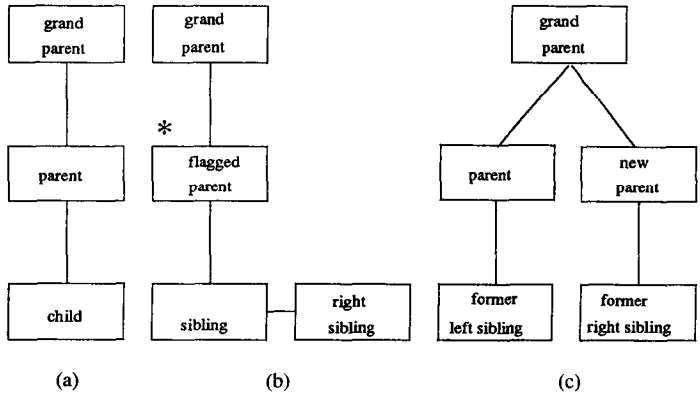

Fig. 1. (a) Initial tree, (b) after insertion, and (c) after cleanup.

horizontal pointer. At the same time, the parent node is marked appropriately, thus showing which pointer refers to sibling nodes.

According to the proposal of [9], siblings are put in order the next time that the parent node is accessed due to any operation. More specifically, the cleanup is performed before any subsequent operation (search, insert, or delete) has to take place below that specific point. This is shown in Figure 1(c). If the grandparent node of two sibling nodes is also full, then the parent node of the sibling nodes splits, creating two new sibling nodes at this level, and the grandparent node is flagged. The grandparent will be cleaned up the next time it is accessed. It is evident that the cleanup phase is not recursive and is executed only once per operation. Deletion is regarded in [9] as any other operation, that is, when traversing down to the leaf level for a deletion, a cleanup phase with node splittings will have to take place if a parent of siblings is encountered. Performing cleanups after deletions in an analogous manner as after insertions or searches may be considered a negative characteristic.

The new B-tree variant proposed here is based on this idea of postponing parent node splitting, and we call it the *lazy parent splitting* technique. In other words, we adopt the method of creating sibling nodes if the parent node is full. At the same time, this parent node is flagged to show which pointer refers to sibling nodes. However, the main differences with [9] are that (a) only constant length entries are considered and (b) as far as the cleanup phase is concerned, deletion maintenance is distinguished from search or insertion maintenance. In other words, in the case of a delete operation (as opposed to the cases of search or insert operations), the clean up phase does not involve any parent node splitting at upper levels. Instead, some item redistribution or/and node mergings are applied so that the structures complies with the standard B-tree definitions. In this way, following the *lazy* approach, splittings are not performed unless they are necessary. Due to this technique, much better time and space performance results are achieved.

In the B-tree variant proposed here, the node type differs only in that nodes have an extra field with integer values ranging from -1 to $2d$. If this value equals -1, then no sibling nodes exist at the immediately lower level. In any other case, the value of this field shows that the corresponding pointer refers to sibling nodes.

It is understood that as a consequence of these techniques, the B-tree definition is violated temporarily. More specifically, paths from root to leaves do not have the same length; therefore, the tree is not balanced. However, any instance of our B-tree variant may be transformed to a conventional B-tree after a finite number of search operations, or even after insertions or deletions. Other B-tree organizations, which diverge

from the classical definition, are presented in the sequel for the purpose of completeness.

First, let us consider the Chained B-trees (CB-trees), which relax the same definition constraint [12]. More specifically, nodes are distinguished into simple and compound types. The latter are twice the size of the former. When simple nodes overflow, they are transformed into compound nodes by binding one more page, whereas overflowing compound nodes do split into two simple nodes. Compound nodes are an integral part of the method; therefore, no cleanup phase is provided. In the worst case of this variant, some paths from the root to the leaves may have even twice the length of the usual tree height. The only advantage of this variant is the smaller average search cost compared with the B-tree average search cost, whereas the storage utilization and the construction cost remain almost the same. A very elegant (though complicated) B-tree variant proposed by Baeza-Yates and Larson elaborates on the idea of having nodes of different sizes [1]. The latter structure is called, B + -tree with partial expansions and it accommodates leaves of many sizes between a minimum and a maximum size.

On the other hand, a recent new and interesting B-tree variation is the one proposed by Srinivasan [15]. According to this technique, a number of leaves share an extra node where any overflowing record out of the leaves is stored. When this overflow node gets full, then a local reorganization takes place. Thus a larger number of leaves are produced that have empty space to accommodate future insertions. The additional cost that is associated with overflow nodes is minimal and the storage and retrieval performance of this structure is similar to that of the Compact B-tree [14]. However, the effects of all the previous techniques on concurrency performance have not been examined.

4. INSERTION AND DELETION ALGORITHMS

Next, our insertion algorithm is described in a structured and more formal way. Note that the term "normal node" means "not flagged" or "not sibling node." Each case arising in the algorithm is depicted in figures. Figure 2(a) shows our initial B-tree variant of degree $d = 1$ without sibling nodes. Next, items with keys 6, 7, and 1 are inserted and, thus, Figures 2(b), 2(c), and 2(d) are produced successively. These examples correspond to cases III, IV, and V of the insertion algorithm, respectively. If items with keys 13 and 4 are inserted successively into the tree of Figure 3(a), then Figures 3(b) and 3(c) are produced, respectively, corresponding to cases I and II of the algorithm.

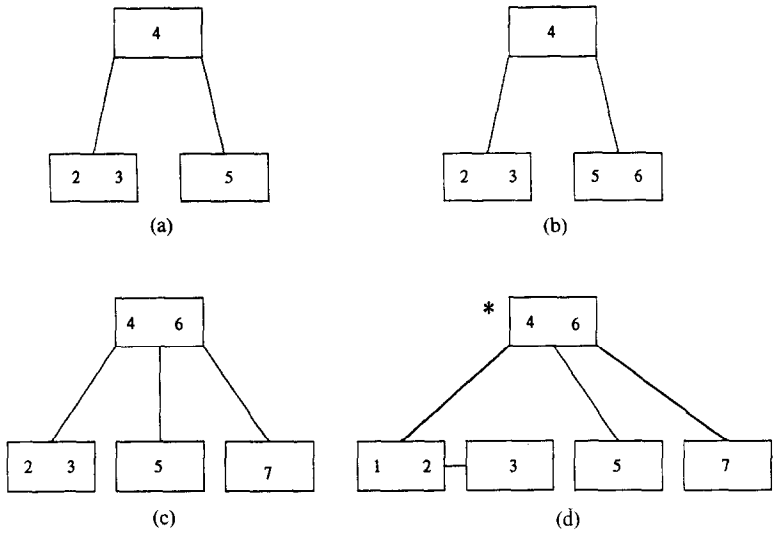

Fig. 2. (a) Initial tree, (b) after insertion of 6, (c) after insertion of 7, and (d) after insertion of 1.

INSERTION ALGORITHM

```
Navigate the tree to the proper leaf where the new
  item has to inserted.

IF a flagged node is met on the path to the leaves
    THEN {clean up} form two normal nodes from its
    sibling children AND post the middle item from
    the siblings in the flagged node
  IF there is free space in the parent of the
    flagged node THEN deflag the node AND split it
    in two normal nodes AND store the middle item in
    its parent {case I}
  ELSE deflag the node AND create two siblings AND
    flag the parent {case II}
ELSE {leaf accessed}
  IF there is free space in the proper leaf THEN
    store the item {case III}
  ELSE IF the parent has free space THEN split the
    leaf in two normal nodes AND store the middle
    item in the parent {case IV}
  ELSE split the leaf in two siblings AND flag the
    parent {case V}.
```

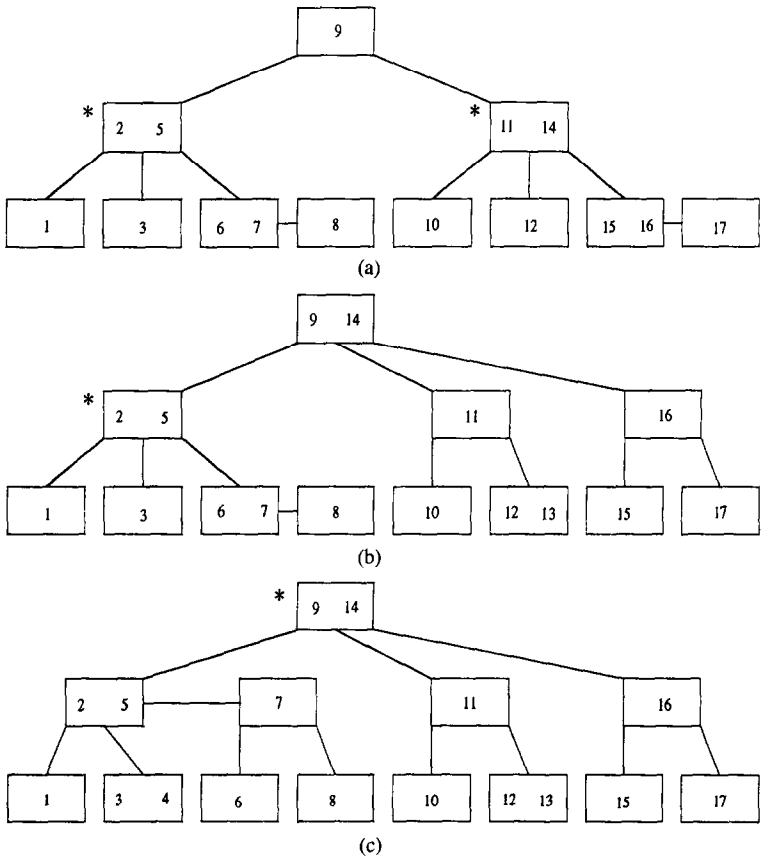

Fig. 3. (a) Initial tree, (b) after insertion of 13 and (c) after insertion of 4.

The deletion is a more complicated procedure, as it happens, even in a simple B-tree. The key to be deleted may be stored in a leaf node or in an internal node. It is known to the reader that the latter case may be transformed easily to the former case. Therefore, let us examine the former case. If no sibling node interferes in the process, then the procedure is identical to the recursive procedure of B-trees. If a sibling node is encountered, it is certain that the process will terminate at this level and no recursion toward the root will be needed. Four distinct cases for the sibling nodes that require special attention may arise:

1. They are leaf nodes and the deletion affects a brother node.
2. They are leaf nodes and the deletion affects them.

3. They are internal nodes and the deletion affects their children.
4. They are internal nodes and the deletion affects children of a brother node.

The deletion algorithm that follows is described in a structured manner too. As can be seen, this algorithm is complicated, and therefore, some abstraction is used for brevity.

DELETION ALGORITHM

```
Navigate the tree to the proper node where the
  item is stored.

IF an internal node contains the item under deletion
  THEN traverse the structure to locate the lexico-
  graphically previous item AND store it in the
  place of the key to be deleted
ELSE the item is actually deleted from a leaf.

Assertion: an item has been deleted from a leaf.

IF the leaf is normal THEN
  IF it contains d-1 items THEN
    IF a normal right brother exists with >d items
       THEN
      IF the parent is normal THEN redistribute
      ELSE {sibling parent} redistribute with no
        clean up
    ELSE IF a sibling right brother exists THEN
      clean up AND deflag AND redistribute {case I}
    ELSE IF a normal left brother exists with >d
       items THEN
      IF the parent is normal THEN redistribute
      ELSE {sibling parent} redistribute with no
        clean up
    ELSE IF a sibling left brother exists THEN
       clean up AND deflag AND redistribute
    ELSE IF a normal right brother with d items
       exists THEN
      IF the parent is flagged THEN clean up AND
        deflag AND merge {case II}
      ELSE IF the parent is sibling THEN clean up
        AND deflag AND merge {case III}
      ELSE {normal parent}
        IF the parent has >d items THEN merge
```

```
        ELSE {parent has d items}
          IF {abstractly} no sibling interferes THEN
            redistribute OR merge recursively as in
            B-trees
          ELSE clean up AND deflag AND redistribute
            {case IV}
      ELSE {normal left brother has d items}
        IF the parent is flagged THEN clean up AND
          deflag AND merge
        ELSE IF the parent is sibling THEN clean up
          AND deflag AND merge
        ELSE {normal parent}
          IF the parent has >d items THEN merge
          ELSE {the parent has d items}
            IF {abstractly} no sibling interferes THEN
              redistribute OR merge recursively as in
              B-trees
            ELSE clean up AND deflag AND redistribute
  ELSE {sibling leaf} clean up AND deflag.
```

Because the possibilities are numerous, we give only some characteristic examples for the most interesting cases in which an eventual cleanup is involved. These examples correspond to the four cases described in the deletion algorithm and thus the reader may skip its details and understand more easily the principles of our *lazy* approach. If the item with key 3 or 1 is deleted from the initial tree in Figure 4(a), then Figure 4(b) or 4(c) is produced, respectively. These two examples belong to the first of the four previously mentioned categories and correspond to cases I and II of the deletion algorithm. If we delete the item with key 3 from the initial tree in Figure 5(a), then the scheme of Figure 5(b) is derived. This instance belongs to the third of the four categories and corresponds to case III of the deletion algorithm. Finally, if we delete 10 from the initial tree in Figure 5(a), then Figure 5(c) is produced. This corresponds to case IV of the deletion algorithm and belongs to the last of the four categories.

5. NUMERICAL RESULTS

We have implemented this B-tree variant in Turbo Pascal language. It should be noted that the code of the insertion and deletion procedures for this variant is more than twice the size of the corresponding procedures of the B-tree as given in [18]. An extensive simulation has been carried out to

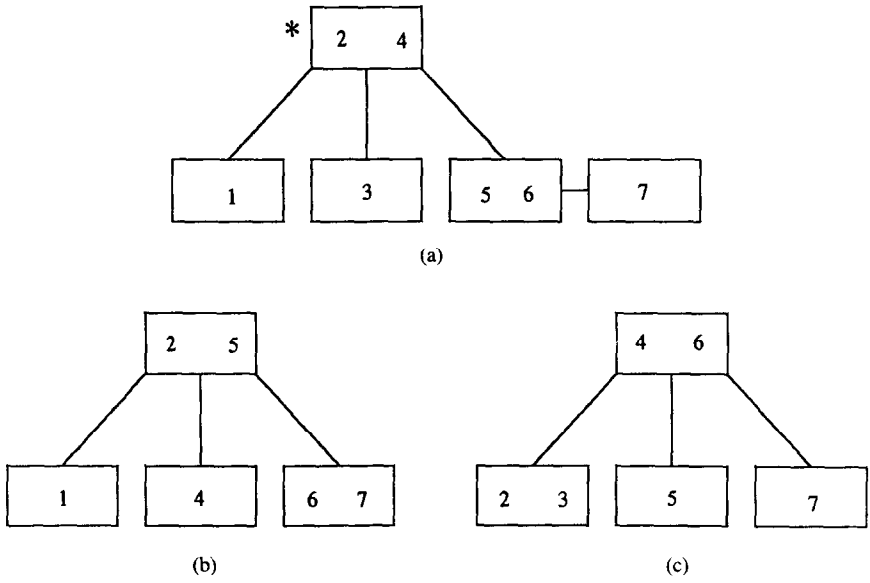

Fig. 4. (a) Initial tree, (b) after deletion of 3 or (c) after deletion of 1.

compare the two structures. Both of the structures have been simulated by inserting a set of random integer keys (in the range from 1 up to 10^{16}) and then deleting one key chosen at random. The ratio of the number of insertions to the number of deletions is a parameter of the simulation. We have tested ratio values equal to 10 and 5, but performance results have not shown significant divergence. Figures 6–8 depict the case that this ratio equals 5.

The costs examined concern maintenance and storage utilization. Therefore, read and write operations during both of the tree constructions and the percentage of storage utilization are counted every 1000 insertions (plus the corresponding deletions). It is noted that the cost metric of the read and write operations is expressed by the number of nodes involved in the path from the root to the leaves and back to the root. The experiment has been repeated 20 times and the relevant expected values of the two structures for every instance have been compared so as to estimate the gain. Figure 6, 7, and 8 show the gain in read operations, the gain in write operations, and the gain in the percentage of storage utilization, respectively, as a function of the number of insertions.

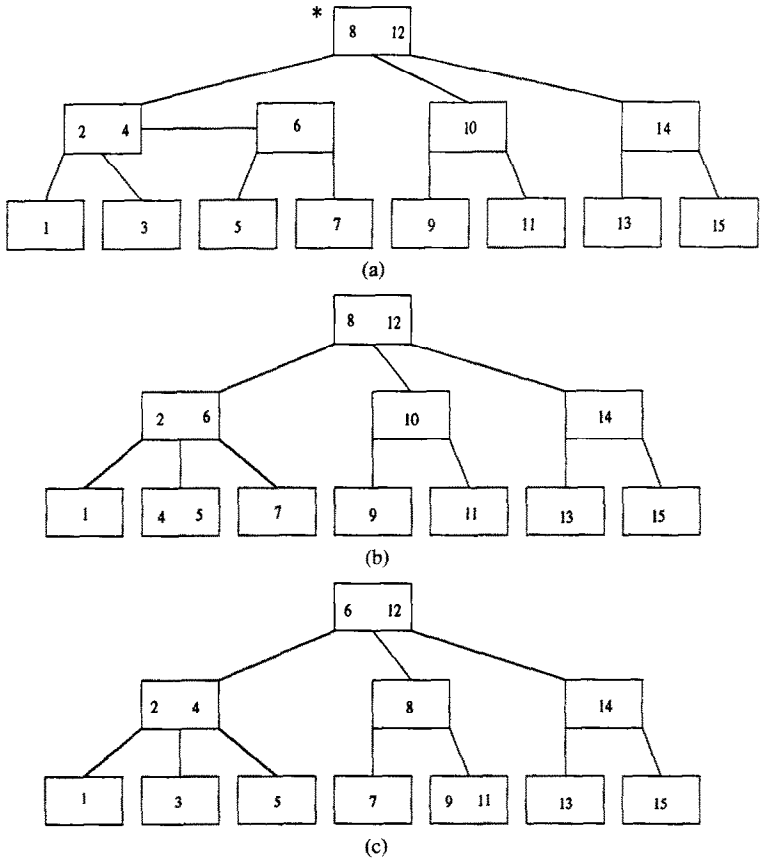

Fig. 5. (a) Initial tree, (b) after deletion of 3 or (c) after deletion of 10.

The figures depict that considerable gain in storage can be achieved by using this technique of *lazy parent node splitting*. In addition, it is evident that our B-tree variant has average height smaller than the average height of the simple B-tree. The search performance is still attributed an $O(\log_d n)$ complexity, where n is the number of keys. Thus, we can conclude that it is guaranteed that the new variant outperforms the simple B-tree as far as searching is concerned. Note also that a consequence of data compactness is improved tree search performance [14]. As far as the write operations are concerned, it is remarkable that when $d = 1$, the gain lies in the area of 24–25%, whereas for $d = 2$ and $d = 3$, the gain is in the area of 5–10%.

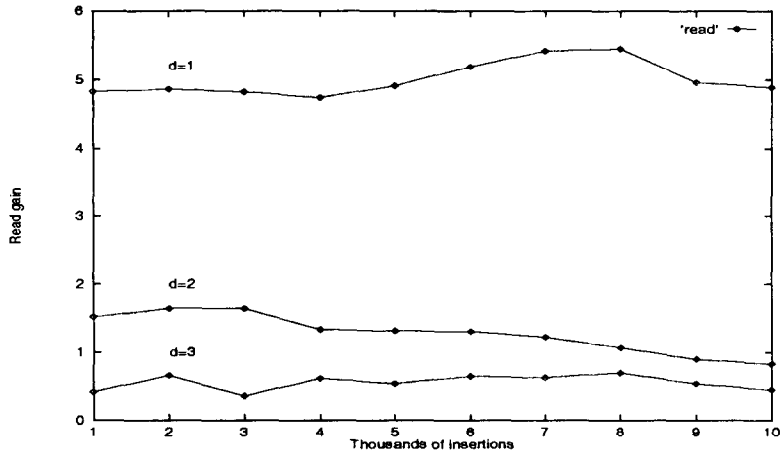

Fig. 6. Gain in read operations (%) vs. the number of insertions ($\times 1000$).

The gain in read operations is not very great because all operations on the path from the root to the leaf nodes are counted in our experiment, whereas write operations are performed only in a restricted area of the tree and, therefore, the relative gain is apparent. It should be remembered that when the number of read or write operations decreases, the number of locks of critical nodes decreases too, allowing, in this way, greater

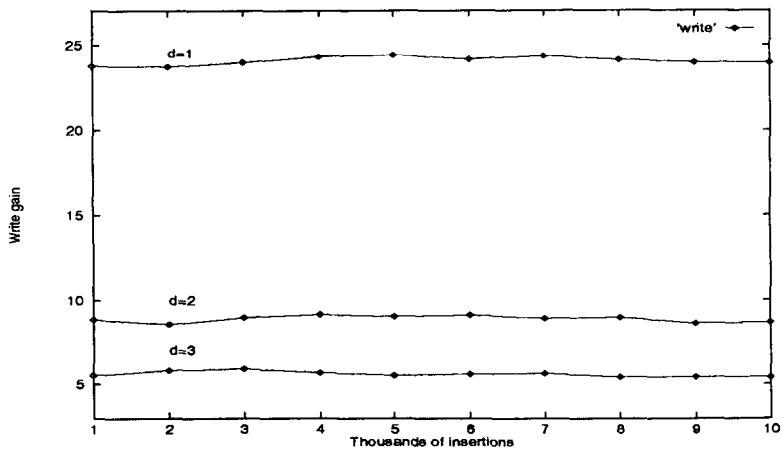

Fig. 7. Gain in write operations (%) vs. the number of insertions ($\times 1000$).

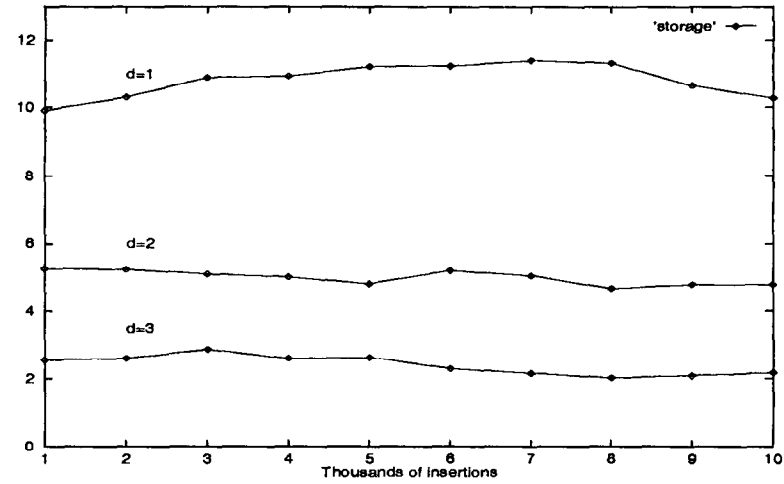

Fig. 8. Gain in storage utilization (%) vs. the number of insertions ($\times 1000$).

concurrent use of the tree structure. If only the number of read nodes under the critical nodes were counted, then it is anticipated that the read gain would be considerably greater.

It is certain that some of the critical nodes do split after an insertion. The question that arises is "How often does a splitting take place?" For example, in the important work [5], it is found that after insertion of the $(n+1)$th item in a random 2–3 tree, the probability of zero, one, two, or more than two splits is 0.57, 0.25, 0.1, and 0.08, respectively. These values also give the probability that the deepest safe node lies in the first, second, third, and above the third lowest level of the tree. In the same reference, it is also found that the lower and upper bounds of split are 0.69 and $0.46 + 0.08 \times \log(n+1)$, respectively. Therefore, it is evident why our lazy parent node splitting technique enhances concurrency.

Figures 6, 7, and 8 also show that the rule of time vs. space trade-offs may have exceptions. In other words, the average storage utilization in B-trees, which is approximately 69% on the average, may be improved by substantially improving the other time-related measures at the same time. Note also that insertions in the variant of B-trees use the techniques of item redistribution or 2–3 node splitting to increase the storage utilization (at minimum 67% instead of 50%), but, on the other hand, the tree maintenance is more expensive. For greater tree degree values ($d > 3$), our variant performance gradually converges to that of the B-tree.

6. CONCLUDING REMARKS

In the present work, we have given a B-tree variant that adopts a *lazy* attitude during update operations. In other words, if a node overflows due to an insertion, then, temporarily, no splitting takes place at the parent level; splitting occurs only at a later phase due to another insert or search operation, whereas delete operations perform item redistributions or node mergings to make the tree agree with the B-tree definition. It has been shown experimentally that by using this technique, gain is achieved over the standard B-tree in terms of read, write, and space metrics. In addition, fewer nodes have to be locked and, thus, concurrency is improved.

An interesting remark is that the gain decreases with increasing tree degree and, therefore, the technique of performing *lazy node splittings* fits very well in small degree B-trees. This technique outperforms the method proposed in [10], at least as far as storage utilization is concerned. However, a direct comparison with [10] with respect to the other performance measures (tree maintenance cost and concurrency availability) is not possible in the present state because [10] examines B+-trees that are leaf oriented.

Future work will include an implementation of *lazy B+-trees*. Thus a comparison of the new technique to the one proposed in [10] may be performed. It is also anticipated that if our *lazy parent node splitting* technique is implemented in B*-trees, then read and write gain, additional storage gain, and greater concurrency will be achieved. Performance improvement is expected also if this technique is applied in Prefix B-trees or in B-trees with variable length entries [4]. Fringe analysis would also derive formal estimates for the storage utilization and other measures as studied in [5]. Finally, the new structure would be examined further in order to embed additional linking techniques and provide concurrency algorithms, which may have comparable performance to the best methods as reported in [8] and [16].

Thanks are due George Moustakidis for his assistance in programming and experimentation. The helpful suggestions made by the two referees improved the quality of the presentation.

REFERENCES

1. R. A. Baeza-Yates and P. A. Larson, Performance of B+trees with partial expansions, *IEEE Trans. Knowledge Data Eng.* 1(2):248–257 (1989).
2. R. Bayer and E. McCreight, Organization and maintenance of large ordered indexes, *Acta Informatica* 1(3):173–189 (1972).

3. A. Biliris, Operation specific locking in balanced structures, *Inform. Sci.* 48:27–51 (1989).
4. D. Comer, The ubiquitous B-tree, *ACM Comput. Surveys* 11(2):121–137 (1979).
5. B. Eisenbarth, N. Ziviani, G. Gonnet, K. Mehlhorn, and D. Wood, The theory of fringe analysis and its applications to 2–3 trees and B-trees, *Inform. Control* 55:125–174 (1982).
6. G. Gonnet, *Handbook of Data Structures and Algorithms*, Computer Science Press, Rockville, MD, 1984.
7. L. J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees, in: *Proceedings 19th IEEE FOCS Conference*, 1978, pp. 8–21.
8. T. Johnson and D. Shasha, The performance of concurrent B-tree algorithms, *ACM Trans. Database Syst.* 18(1):51–101 (1993).
9. A. M. Keller and G. Wiederhold, Concurrent use of B-trees with variable length entries, *ACM SIGMOD Record* 17(2):89–90 (1988).
10. O. Nurmi, E. Soisalon-Soisinen, and D. Wood, Concurrency control in database structures with relaxed balance, in: *Proceedings 6th PODS Conference*, 1987, pp. 170–176.
11. O. Nurmi and E. Soisalon-Soisinen, Uncoupling updating and rebalancing in chromatic binary search trees, in: *Proceedings 10th PODS Conference*, 1991, pp. 192–198.
12. T. V. Prabhakar and H. V. Sahasrabuddhe, Towards an optimal data-structure: CB-trees, in: *Proceedings 10th VLDB Conference*, 1984, pp. 235–244.
13. M. J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw Hill, New York, 1988.
14. A. L. Rosenberg and L. Snyder, Time- and space-optimality in B-trees, *ACM Trans. Database Syst.* 6(1):174–183 (1981).
15. B. Srinivasan, An adaptive overflow technique to defer splitting in B-trees, *Comput. J.* 34(5):397–405 (1991).
16. V. Srinivasan and M. J. Carey, Performance of B-tree concurrency algorithms, in: *Proceedings ACM SIGMOD 91 Conference*, 1991, pp. 416–425.
17. G. Wiederhold, *File Organizations for Database Design*, McGraw-Hill, New York, 1987.
18. N. Wirth, *Algorithms and Data Structures*, Prentice Hall, Englewood Cliffs, NJ, 1986.

Received March 1992; revised December 1992, June 1993; accepted November 1993