HYSTERICAL B-TREES

David MAIER * and Sharon C. SALVETER **
Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794, U.S.A.

Received March 1980; revised version received March 1981

B-trees, complexity of algorithms, indexing techniques, data structures

1. Introduction

In an order d B-tree with n nodes, a single insert or delete operation may cause $\theta(\log_d n)$ splits and merges. Moreover, since an insert operation can exactly invert the effect of a delete operation, and vice versa, a sequence of m inserts and deletes can cause $\theta(m \log_d n)$ splits and merges. We present a scheme that reduces the number of splits and merges required to process m inserts and deletes to $\theta(m)$, while only modestly increasing the depth of the tree.

On the surface it may not seem advantageous to limit the number of splits and merges. Before performing an insert or delete operation, it is generally necessary to perform a tree search involving $\theta(\log_d n)$ nodes. There are situations, however, where the number of splits and merges is the bounding factor in the time to insert and delete. If the application involves locality of reference – that is, inserts and deletes tend to occur in close proximity in the tree – maintaining 'fingers' that point into the tree can reduce search time substantially below $\theta(\log_d n)$ [2]. Furthermore, searching involves only reading the contents of nodes, while splitting and merging require modifying nodes. With certain storage strategies, writing into the tree is more costly than reading from the tree.

B-trees are often proposed as index structures in databases. In a concurrent database environment, multiple processes can search the tree simultaneously, since searching involves only reading. However, since insertions and deletions cause modifications to the tree, they may not be arbitrarily performed by concurrent processes; some locking mechanism must be implemented. Several researchers have dealt with locking protocols for B-trees [9,1,8]. Their methods calculate the highest point to which a split or merge may propagate, thus limiting the portion of the tree that must be locked. Our method, while not obviating locking in a concurrent environment, is a heuristic for reducing both the total duration and extent of locks. Since there are fewer splits and merges, the total amount of time that any portion of the tree must be locked is reduced. Since the number of splits and merges is reduced, the average height to which a change may propagate is smaller, and hence the portion of the tree that must be locked is smaller.

The reader unfamiliar with the notation and properties of B-trees is directed to [3,4,6]. In this paper, any node in an order d B-tree has a minimum of d children. Elsewhere, d sometimes denotes the maximum number of keys in a node, the minimum number of keys in a node or the maximum number of children of a node, instead.

2. A worst case example

Consider the order d B-tree shown in Fig. 1. Every non-leaf node along the leftmost path is full: each has $2d - 1$ children. Every other non-leaf node in the tree has the minimum number of children, d. If we insert a node into the tree as shown as Fig. 1, we trigger a sequence of splits that extends all the way to the root.

* This research is supported by NSF grant IST7918264.
** This research is supported by NSF grant ENG7907994.

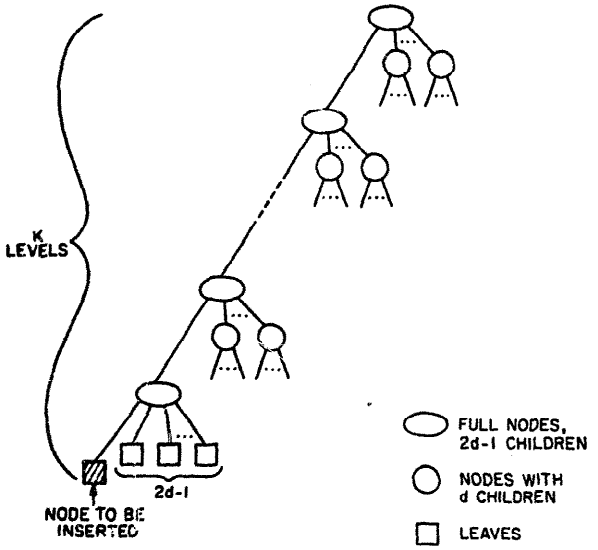

Fig. 1.

The result of such an insert is shown in Fig. 2.

If we now delete the node just added, a sequence of merges extends all the way to the root. The result is the tree shown in Fig. 1, minus the shaded node. Thus, a sequence of 2m insert and delete operations can spawn a series of $\theta(m \log_d m)$ splits and merges:

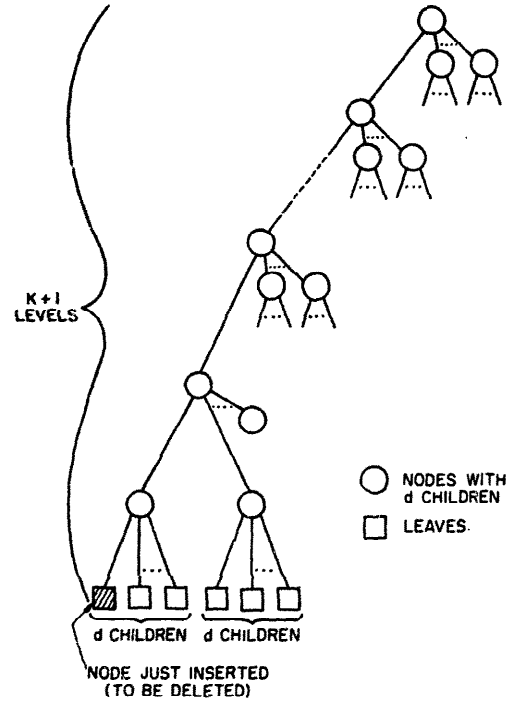

Fig. 2.

build a tree of m nodes and then perform m insert and delete operations as described above. The tree with the bad behavior described can be constructed by inserting m elements into the null tree in largest to smallest order. We propose a method that guarantees at most $\theta(m)$ splits and merges. (We consider the m operations to include those required to initially build the tree as well as those to maintain it.) Note that any B-tree built with m operations can contain at most m nodes.

Brown and Tarjan [2] show that such bad behavior cannot occur if the m operations are all inserts, all deletes, or a sequence of inserts and deletes where all the inserts occur 'far enough away' from the deletes. Although their results are for 2-3 trees, they would seem to generalize to arbitrary B-trees. The undesirable behavior can occur, however, with mixed inserts and deletes when there is locality of reference in the tree.

B^*-trees are a variant of B-trees that attempt to avoid splitting full nodes by shifting children to the right or left sibling as space is available. If a node and its sibling are both full, the two nodes are split into three nodes, each resultant node having about 2/3 m children, where m is the maximum number of children. The general requirement is that every node have between m and $(2m - 1)/3$ children, except the root may have up to 4/3 m children. For deletions, if borrowing is not possible, three nodes are merged into two.

The same bad behavior as in B-trees can be produced in B^*-trees. All we need to show is that a deletion can exactly undo the effect of an insertion that produces a split. Let $m = 10$ and let nodes N_1 and N_2 be siblings with 10 children each. Inserting a child of N_1 causes a split into N_1', N_2' and N_3', each with 7 children. Removing the just inserted child brings N_1' to 6 children, fewer than allowed. N_1' cannot borrow from N_2', so N_1', N_2' and N_3' are merged back into N_1 and N_2, each with 10 children.

3. Hysterical B-trees

We present a modification of a B-tree called a *hysterical B-tree* [1] that is similar to the standard B-tree except for the criterion for merging nodes.

[1] Hysteretic is the proper adjectival form of hysteresis, but frankly, the field of data structures could use a little levity.

Definition. A d_p *B-tree* is the same as an order d B-tree, except a node can have as few as d − p children. (The maximum is still 2d − 1.)

We also call a d_p B-tree an order d B-tree with *hysteresis* p. We assume d > p.

The insert operation for hysterical B-trees is the same as for regular B-trees. The delete operations are modified as follows:

delete node
if parent of node now has < d − p children
  then if parent has a neighbor with ≥ d − p + 1 nodes
    then borrow one of the neighbor's children
    else merge parent with a neighbor to get a node with 2(d − p) − 1 children.

As with regular B-trees, insert and delete operations propagate up the tree in the case of a split or merge.

Consider the B-tree in Fig. 1 again. This time, we assume it is a d_p B-tree: the full nodes still have 2d − 1 children, but the nodes represented by circles have d − p nodes. If we insert a node at the left of the tree, the result is again the tree shown in Fig. 2. However, when we delete the node just inserted, we do not revert to the tree in Fig. 1. Rather, we obtain the tree shown in Fig. 3. Note that no merging takes place.

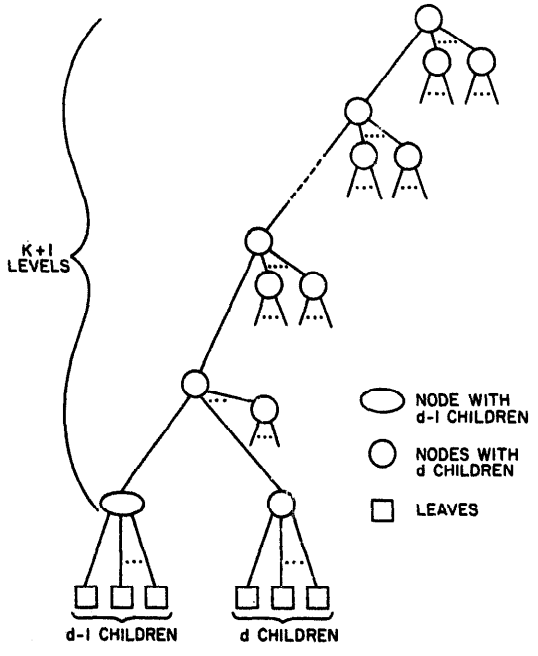

Fig. 3.

4. Analysis

Since search time is proportional to the depth of the B-tree, we do not want hysterical B-trees to be drastically deeper than regular B-trees. For a regular order d B-tree with n nodes, we get the maximum depth when every node has the minimum number of children, d. In this case, the tree has depth $\lceil \log_d n \rceil$, for a d_p B-tree with n nodes, we can have as few as d − p children at each node, for a maximum depth of $\lceil \log_{(d-p)} n \rceil$. Since $\log_d n = \log_2 n / \log_2 d$, $\log_{(d-p)} n = \log_2 n / \log_2 (d-p)$ and $\log_n (n + d - p) = \log_2 (n + d - p) / \log_2 n$, we obtain the following result, using a little algebra.

Lemma. If b is the maximum depth of an order d B-tree with n nodes and h is the maximum depth of a d_p B-tree with n nodes, then

$$h \leqslant b \cdot \frac{\log_2 d}{\log_2 (d-p)} \cdot \log_n (n + d - p) .$$

Note that $\log_n (n + d - p)$ tends toward 1 as n gets large. Table 1 gives values of $\log_2 d / \log_2 (d - p)$ for various values for d and p. For small values of p, this ratio is only slightly larger than 1. We do not consider values of p greater than d/2, since the depth of the tree begins to increase significantly for practical values of d.

We now turn our attention to the number of splits and merges necessary to maintain a d_p B-tree. Observe that a node splits when it has 2d children; the two children resulting nodes each have d children. For a merge, a node must have d − p − 1 children its neighbor d − p. The resulting node has 2(d − p) − 1 children.

Table 1

p	d			
	4	8	16	32
1	1.26	1.07	1.02	1.01
2	2.00	1.16	1.05	1.02
3	×	1.29	1.08	1.03
⋮				
d/2	×	1.50	1.33	1.25

We now calculate the minimum numbers of insert and delete operations that must take place at a node newly formed by splitting or merging before another split or merge takes place involving the node:
(1) split–split:

$2d - d = d$ operations (inserts),

(2) split–merge:

$d - [(d - p) - 1] = p + 1$ operations (deletes),

(3) merge–split:

$2d - [2(d - p) - 1] = 2p + 1$ operations (inserts),

(4) merge–merge:

$[2(d - p) - 1] - [(d - p) - 1] =$

$= d - p$ operations (deletes).

We shall only consider d_p B-trees where $p \leqslant (d - 1)/2$. Therefore, the minimum number of inserts and deletes between splits and merges is $p + 1$. We now prove our main result.

Theorem. If m insert and delete operations are performed over the entire history of a d_p B-tree (including the operations to initially construct the tree), then at most m/p splits and merges are made over the same period.

Proof. Consider the sequence of m insert and delete operations and the resulting sequence of splits and merges. We want to assign at least p inserts and deletes to each split and merge to achieve our result. The assignment is carried out in two phases. First we assign inserts and deletes to individual nodes in the tree. Then, whenever a node is split or merged, we re-assign the operations assigned to that node to the split or merge.

We observe that for any insert or delete operation, there is exactly one node in the tree that has the number of its children changed without the node itself splitting or merging. Either it is the node where a sequence of (possibly zero) splits and merges terminates, or the neighbor of that node. It is the neighbor exactly in the case that a node-borrowing was possible. The neighbor loses one child; the node that borrowed the child has the same number of children, namely d -- p.

We therefore assign each insert or delete operation to the node whose number of children changes without splitting or merging. By our previous remarks, a node must have the number of its children changed at least p times before a split or merge takes place. (The p + 1 change can cause a merge.) Hence, when a node splits or merges, it has at least p insert or delete operations assigned to it to reassign to the split or merge.

As we have noted, m operations on a regular order d B-tree can cause $\theta(m \log_d m)$ splits and merges. Our m/p bound for d_p trees is a considerable improvement. Note the m/p bound does not depend on d, except for $1 \leqslant p \leqslant (d - 1)/2$.

5. Refinements and related work

We have seen that a d_p B-tree can be deeper than a corresponding order d B-tree produced by the same sequence of operations. If we know that during certain phases of a given application no changes to the tree will take place, we can traverse the d_p tree in a depth-first manner, combining nodes with fewer than d children (or any other value between d – p and 2d -- 1, for that matter). The compressed tree will be comparable in depth to the regular B-tree.

We note that Huddleston [5] independently obtained similar results for 2-3-4 trees. Melhorn [7] has independently defined 'weak' B-trees, which are essentially the same as hysterical B-trees.

References

[1] R. Bayer and M. Schkolnick, Concurrency of operations on B-trees, IBM Res. Rep. RJ1791 (1976).
[2] M. Brown and R. Tarjan, Design and analysis of a data structure for representing sorted lists, Stanford University, Computer Science Department, Tech. Rep. #STAN-CS-78-709 (1978).
[3] D. Comer, The ubiquitous B-tree, Comput. Surveys 11 (2) (1979).
[4] E. Horowitz and S. Sahni, Fundamentals of Data Structures (Computer Science Press, 1976).
[5] S. Huddleston, The average number of restructurings in 2-3-4 trees with insertions and deletions is constant, unpublished notes (1979).
[6] D. Knuth, The Art of Computer Programming (Sorting and Searching) (Addison-Wesley, Reading, MA, 1973).
[7] K. Melhorn, A new data structure for representing sorted list, unpublished notes (1979).
[8] R. Miller and L. Snyder, Multiple access to B-trees, Proc. Conf. Information Science and System (1973) pp. 400–407.
[9] B. Samadi, B-trees in a system of multiple users, Information Processing Lett. 5 (4) (1976) 107–112.