P-Trees: Storage Efficient Multiway Trees

David M. Arnow, Aaron M. Tenenbaum, and Connie Wu

Department of Computer and Information Science
Brooklyn College
City University of New York

ABSTRACT

A new variation of high order multiway tree structures, the P-tree is presented. P-trees have average access costs that are significantly better than those of B-trees and are no worse (and often better) in storage utilization. Unlike compact B-trees, they can be maintained dynamically, and unlike dense multiway trees and B-trees, their associated insertion algorithm, which is also presented, is cheap and involves (at most) a very localized rearrangement of keys.

1. Introduction

Tree structures are widely used in storing data to permit efficient access, insertion and deletion operations. In organizing data on secondary storage media, the choice of viable tree structures is severely restricted because of the great difference in speed of memory-to-memory and device-to-memory operations, and the need for high space utilization. Any viable tree structure must satisfy several requirements:

a) Primary memory usage must be limited: the number of node images required by the associated maintenance algorithms must be limited to a small fixed number or, in the case of a high order balanced tree, limited to the height of the tree;

b) Both efficiency in transfering nodes between secondary and primary memory and a reasonable storage utilization are required: thus, only high order trees can be considered;

c) It must be possible to access all nodes from the root with few node visits: again, high orders are required (to provide high branching factors) as well as possible rearrangements of the tree structure;

 ACM 0-89791-159-8/85/006/0111 $00.75

Because of their strong viability under these circumstances, the B-tree of high order or a variation has become the tree structure of choice in the implementation of data base systems and in the organization of external files. An algorithm for inserting keys into B-trees and further discussions of their properties may be found in the references [2, 3, 6, 9]. With few exceptions ([4], for example), the possibility of using alternative, high-order trees has been largely ignored. In particular, because of their poor worst-case access properties, M-trees [defined below], have generally not been studied.

An M-tree is the following generalization of the binary search tree:

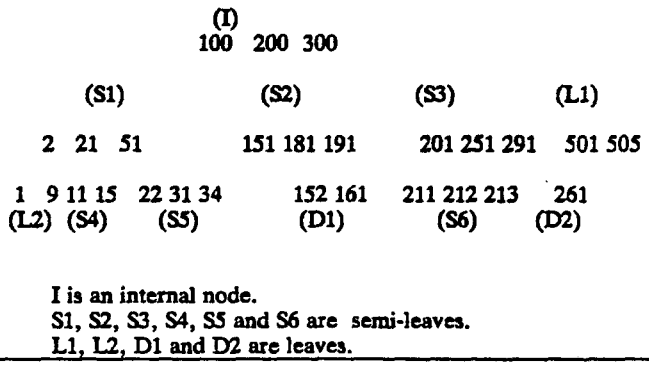
I. Let D be the degree (or order) of the tree. In each node in the tree there are D pointers, numbered 1 through D, which are null or which point to child nodes. The i-th child of a node is the child node pointed to by its i-th pointer. In each node there are up to D-1 keys.

II. The **order requirement** is that (1) in any node the i-th key is less than the (i+1)-th key and (2) all the keys in the i-th child of a node are greater than the (i-1)-th key of the node, if it exists, and less than the i-th key of the node, it if exists.

III. There are three types of nodes (See Figure 1.):

Internal nodes have D-1 keys and D children.
Leaves have fewer than D-1 keys and no children.
Semi-leaves have D-1 keys and fewer than D children.

Figure 1: M-Tree Terminology



I is an internal node.
S1, S2, S3, S4, S5 and S6 are semi-leaves.
L1, L2, D1 and D2 are leaves.

To insert a key, K, into an M-tree:

(1) A search is made for K in the M-tree, terminating (unsuccessfully) either when a leaf is reached or when a semileaf is reached under the following circumstances:

a) i is the smallest positive integer such that the i-th key in the semileaf is greater than K (if no such i exists, then i is set to D);

b) the i-th pointer is null.

(2) **Leaf case:** The key is inserted into the leaf in such a way as to preserve the order condition. If the leaf node now has D-1 keys then it becomes a semi-leaf. The insertion is complete.

(3) **Semileaf case:** A new leaf node is allocated, the i-th pointer reset to point to it, and the new key is inserted into the new leaf. (At this point, if the parent of the new leaf has D children it is no longer a semileaf, but an internal node.) The insertion is complete.

Recently, a comparative study of M-trees, B-trees and compact B-trees [1] demonstrated that the average search costs of M-trees are competitive and often better than those of B-trees. At low orders, average storage utilization for M-trees is superior to that of B-trees, but because M-tree storage utilization is extremely, adversely dependent on order, this advantage does not persist at the high orders required in practical situations. This is contrary to the conventional wisdom that B-trees are space-poor and time-efficient. The strong average case access performance of random M-trees is intriguing. Can the storage utilization (and hopefully the worst case access behaviour) of M-trees be improved without disrupting the above properties?

In this paper, we answer that question affirmatively by presenting a new, promising variation of high order multiway tree structures that could potentially be employed to manage large collections of information on rotational storage media, and avoid the poor storage utilization of M-trees at high orders.

2. Drip Trees

In a certain sense, the critical point of any of the tree insertion algorithms occurs at the moment it is necessary to insert a key into a full node. In the case of the B-tree, the node is split and the median key is moved up. Consider an insertion into an M-tree which results in an "attempt" to insert the key into a leaf that is full. Normally, a new node would be created (i.e. an extra block of secondary storage would be allocated) and the new key would become the sole occupant of the extra block. Since the order of the tree is presumably high, such singly occupied blocks would proliferate, thus causing the storage utilization problem. Since it is these sparsely populated leaves of M-trees that cause the disasterous space utilization statistics of high order M-trees, it is necessary to seek ways to pack the data found in these underpopulated nodes into denser structures. One possibility is to associate an overflow node with every full leaf in the event of such an insertion. A refinement of this approach leads to the idea of **drip trees.**

The definition of drip trees includes parts I, II, and III of the definition of M-trees (see above). In addition:

A semileaf may contain one or more **drip regions**, which are sets of contiguous pointers. (When a leaf node becomes a semileaf, all its pointers form one drip region. The drip regions are modified by subsequent insertions, as explained below.) Not all

pointers in a semileaf need be contained in a drip region. The i-th key in a node is said to be "in" a drip region if the i-th and (i+1)-th pointers are both in the same drip region. Let N be the number of pointers in a drip region, and let m=N **div** 2. Then the m-th pointer of the drip region is either null or points to a leaf node, which is termed a **drip node**. All other pointers in a drip region are null.

Viewing the tree in Figure 1 as a drip tree, the following observations can be made:

(1) I is still an internal node.

(2) S1, S2, S3, S4, S5 and S6 are still semi-leaves. S1 has no drip region. S2, S4, S5, and S6 each have one drip region, delimited by pointers 1 and 4. S3 has one drip region, delimited by pointers 3 and 4.

(3) L1 and L2 are leaves, but not drip nodes, while D1 and D2 are leaves and are drip nodes.

To insert a key into a drip tree (See Figure 2.):

Steps 1 and 2 are identical to steps 1 and 2 of the *M*-tree insertion procedure, above.

(3) **Semi-leaf case:** If the i-th pointer is **not** in a drip region, then the actions taken are the same as those in step 3 of the M-tree insertion procedure, above. If the i-th pointer **is** in a drip region:

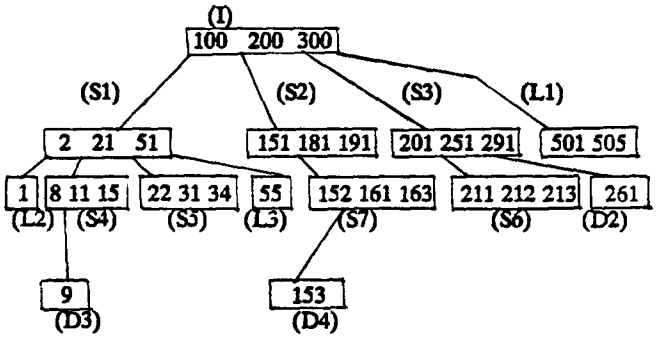
   a) The set of keys consisting of those in the drip region and the new key is considered: the median of these is inserted into the drip node (if no drip node exists then one is allocated) and the others are redistributed in the drip region. Both operations are carried out in a way to preserve the order condition. If the drip node has fewer than D-1 keys then the insertion is complete.

   b) If the drip node has D-1 keys (i.e. is full): The drip node becomes a semi-leaf. The pointer to this node ceases to be part of the drip region (in its parent). The set of pointers to the left of this pointer and the set of pointers to the right of this pointer become separate drip regions, unless either set is null or consists of only one pointer; in that case the pointer (if any) is no longer part of any drip region.

The storage utilization for such a tree must be greater than 33%, regardless of order, because for every full node (an internal node or a semileaf) there are at most two nodes (leaves) which are not full. On the average, the storage utilization for drip trees is much better.

Although this marked improvement over storage utilization in M-trees (the *average* of which tends to zero with high order!) is heartening, this "drip" approach will lead to pathologically structured trees. To see this, consider the point when a semi-leaf node with one all-encompassing drip region has a single full drip node. Half of the keys

Figure 2: The Drip Tree of Figure 1 After Inserting 8, 55, 163, and 153



Insertion of 8 into S4 caused drip node D3 to be created with key 9 "dripping" down. Insertion of 55 into S1 caused new leaf L3 to be created. Insertion of 163 into the former drip node D2 caused it to become a semileaf (S7). Subsequent insertion of 153 into S7 resulted in the new drip node D4.

making up the two nodes reside in the drip node. If the input has been random and if the order is (as it must be) high, then the keys in the drip node cover, on the average, half of the key space that is covered by the parent node. This means that very nearly half of the keys whose search paths lead through the parent will also go through this first child, while the other half of those keys will be distributed over the other children. This will clearly lead to an excessively unbalanced tree, somewhat reminiscent of a poplar tree, or, in cases of very large data sets, of a douglas fir.

3. P-Trees

This pathological tendency can be curbed by amending, in the drip tree insertion procedure above, step 3b, which handles the full drip node case, to produce a new type of tree, the **P-tree:**

If the drip node has D-1 keys (i.e is full):

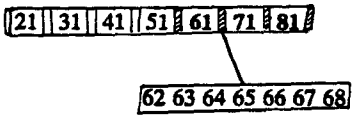
i) The parent drip region contains four or more pointers: the drip region is divided into two drip regions, the number of pointers of which differ by at most one. A new node is allocated so that both drip regions have a drip node in their center. The keys in the original drip region and the original drip node, and the new key are redistributed over the two drip regions and their drip nodes so that the parent node stays full, the number of keys in the two drip nodes differ by at most one, and the order condition is preserved. (See Figure 3.a.)

ii) The parent drip region contains three pointers: two new nodes are allocated and the two null pointers of the drip region are set to point to them. The keys in the drip region and the drip node and the new key are redistributed over the three child nodes and the drip region so that the parent stays full, the number of keys in any two of the child nodes differ by at most one, and the order condition is preserved. The three child nodes are now leaves, but not drip nodes, and the three pointers in the parent cease to constitute a drip region. (See Figure 3.b.)

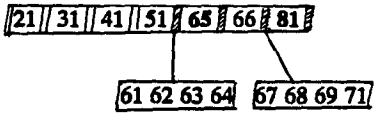
iii) The parent drip region contains two pointers: a new node is allocated and the null pointer of the drip region is set to point to it. The key in the drip region and the keys in the drip node and the new key are redistributed over the two child nodes and the drip region so that the parent stays full, the number of keys in the two child nodes differ by at most one, and the order condition is preserved. The two child nodes are now leaves, but not drip nodes, and the two pointers in the parent cease to constitute a drip region. (Similar to Figure 3.b.)

Figure 3: Percolations (Local Redistributions of Keys)

(a)

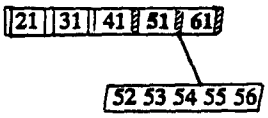


The insertion of the key 69 into a drip region of >3 pointers yields:

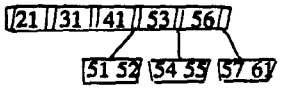


Initially, the parent has one drip region (bold), which includes the keys 61, 71, and 81. After insertion and percolation, the parent has two drip regions: one which includes the key 65, the other which includes the key 81.

(b)



The insertion of the key 57 into a drip region of 3 pointers yields:



Initially, the parent has one drip region in question, which includes the keys 51 and 61. After the insertion and percolation, the pointers in what was the drip region are no longer part of a drip region.

Intuitively, the effect of this percolation-like (hence the name "P-tree") redistribution is two-fold:

(1) Growth in the breadth, as distinct from depth, is encouraged. A cursory examination of the P-tree algorithm reveals that it is impossible for a node to have grandchildren before it has at least $\log_2 D$ children.

(2) Assignment is made, as uniformly as possible, of subtrees of a tree to different subregions of key space, without damaging storage utilization.

4. Some Properties of P-Trees

Storage Utilization. The storage utilization of P-trees exhibits an oscillatory dependence on order, with the best orders being of the form 2^N and the worst orders being of the form $2^N+2^{N-1}=3x2^{N-1}$ (in both cases N is an integer and is greater than 0). To see

trees. In fact, for "good" orders (i.e. powers of 2), *the average storage utilization of P-trees is better than that of B-trees.*

Table 2 presents the search cost vs. file size for order-16 P-trees and B-trees. Although ordinary M-trees are occasionally worse than B-trees in this respect, *P-trees consistently have a superior average search time cost than B-trees.* Furthermore, of the set of P-trees sampled for each category, the access cost of the worst P-tree was consistently superior to the average access cost for B-trees. Although not presented here, similar results were found for orders ranging from 3 to 32, file sizes from 1,000 to 20,000, and sample sizes of up to 1,000.

Table 1:
Average Storage Utilization for P-trees, M-trees and B-trees
(10000 keys)

Order	P-tree	B-tree	M-tree
3	81	67	84
4	83	68	72
6	81	68	57
8	82	68	49
12	68	68	38
16	77	68	32
24	65	69	<24
32	72	69	--

Table 2:
Access Costs For Order 16 P-trees and B-trees
(sample size = 20)

File Size	Average P-tree	Worst P-tree	Average B-tree
1000	2.73	2.73	2.91
2000	2.90	2.93	3.26
3000	3.02	3.08	3.92
4000	3.15	3.20	3.92
5000	3.25	3.30	3.93
10000	3.58	3.59	3.95

6. P-Trees and Compact B-trees: A Practical Hybrid Structure

Rosenberg and Snyder [7, 8] have shown that optimal storage B-trees, which they termed "compact B-trees", are near optimal in access time cost. Although Rosenberg

this, we explore the worst case storage utilization by considering the circumstances under which a node can be less than 50% full:

(1) it is the sole child of a full parent, in which case the storage utilization of the two nodes together is better than 50%.

(2) it is a node that resulted from a three-way split of a full drip node that belonged to a drip region of three pointers.

Thus, storage utilization can't possibly be less than 33% and if the three-way splits can be avoided, then storage utilization will exceed 50%. The latter can be accomplished by insisting that the order be of the form 2^N. The greatest number of three-way splits will occur when the order is of the form 2^N+2^{N-1}. (For example, (24) --> (12,12) --> (6,6,6,6) --> (3,3,3,3,3,3,3,3).) As seen in the next section, the average storage utilization is much better, though also somewhat dependent on order.

Access Costs. Figure 4 shows the canonical worst case structure for a P-tree. It is apparent that the worst case access cost for an order D P-tree of N keys is $N/((D-1)\log_2 D)$, as compared with N/(D-1) for M-trees. Although the dependence on file size is still linear (as in the case of an M-tree), the height-decreasing factor, $\log_2 D$, suggests that the average case should be significantly improved as well. The results of the next section indicate that this is indeed the case.

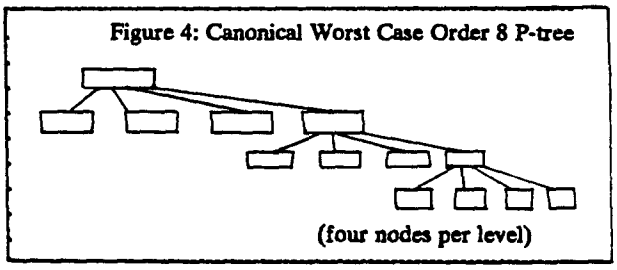


Figure 4: Canonical Worst Case Order 8 P-tree

5. Empirical Comparison With Other Structures

Simulations suggest that the average case storage utilization and the average case search cost of P-trees are quite good. Table 1 presents the average storage utilization vs. order for P-trees, M-trees, and B-trees with 10,000 keys. Whereas the average storage utilization of B-trees is essentially order-independent, the average storage utilization for P-trees, while showing some dependence on order is generally good and is competitive with B-

and Snyder provided an algorithm for transforming an existing B-tree into a compact one, there is no known efficient algorithm for compactness-preserving insertion. The programme that they suggested included periodic compaction, perhaps concurrently with backups. Between compactions, insertions and deletions into the structure are to be carried out in the usual B-tree fashion. Other work [5] indicates that periodic reorganization of B-tree files is not unacceptable and, in fact, can be cost-effective.

Unfortunately, recent work [1] makes it clear that unless the structure is very non-volatile, the use of compact B-trees provides no genuine improvement in storage utilization. This is because of the rapid decline to near-pessimal levels in storage utilization exhibited by high order compact B-trees as a result of insertions. Even if the structure is growing very slowly, the use of compact B-trees doubles insertion cost and at high order provides little search cost advantage over that of a random B-tree.

These considerations, along with the properties of the P-tree suggest the use of a hybrid arrangement: periodically compact the data structure into compact B-tree form, as suggested by Rosenberg and Snyder, but instead of letting it grow as a B-tree between compactions, let it grow as a P-tree. The consequences of this are as follows:

(1) **Storage Utilization** will fall rapidly with insertion into the compact structure, but will not reach near-pessimal levels. Rather, it will "bottom out" at 67%.

(2) **Insertion Cost** will be low because P-tree insertion only involves local rearrangements, rather than the multiple splits of B-tree insertion into a compact B-tree.

(3) **Access Cost:** average case access costs will be good because both compact B-trees and P-trees are excellent in that regard. Worst case access costs will be much less of a danger than they would to "pure" P-trees, since the periodic compaction would smooth out any pathological structures.

Thus, P-trees and compact B-trees appear to solve each other's problems without the introduction of any new difficulties!

7. References

(1) Arnow, D.M. and Tenenbaum, A.M., (1984) An empirical comparison of B-trees, multi-way trees and compact B-trees. **Proc. of ACM BSIGMOD**, 33-46.

(2) Bayer, R. and McCreight, E., (1972) Organization and maintenance of large ordered indexes. **Acta Informatica 1**, 173-189.

(3) Comer, D., (1978) The ubiquitous B-tree. **Computing Surveys 11,2**, 121-138.

(4) Culik, K., Ottmann, Th., and Wood, D., (1981) Dense Multiway Trees. **ACM TODS 6,3**, 486-512.

(5) Gudes, E. and Tsur, S., (1980) Experiments with B-tree reorganization. **Proc. of ACM SIGMOD**, 200-206.

(6) Knuth, D.E., (1973) **The Art of Computer Programming, Vol. 3.** Addison-Wesley, Reading, Mass.

(7) Rosenberg, A.L. and Snyder, L., (1979) Compact B-trees. **Proc. of ACM_ SIGMOD**, 43-51.

(8) Rosenberg, A.L. and Snyder, L., (1981) Time- and space-optimality in B-trees. **ACM TODS 6**,1, 174-193.

(9) Yao, A.C., (1978) On random 2,3 trees. **Acta Informatica 9**,3, 159-170.