An Adaptive Overflow Technique to Defer Splitting in B-trees

B. SRINIVASAN
Department of Computer Science, Monash University, Australia 3168 (e-mail: srini@bruce.cs.monash.edu.au)

Time and space trade-off studies by Rosenberg and Snyder (1981) have shown that the space-optimal B-trees are nearly time-optimal. That means efforts to explore a B-tree variant that enhances the space-efficiency of the tree would be worthwhile. Although the compact B-trees of Rosenberg and Snyder (1981) achieve high space utilisation, they require very expensive reorganisations and hence they are impracticable even for relatively small and reasonably stable databases. In addition the storage utilisation of compact B-trees may deteriorate quickly because of splitting of leaf nodes resulting from insertions. The possibility of attaching overflow nodes to the leaf nodes of B-tree to defer the splitting of nodes was proposed in Ref. 7. This paper extends and formalises the new data structure and analyses its performance both quantitatively and by simulation. The performance analysis shows that the additional cost associated with overflow nodes is minimal and the storage utilisation and retrieval performance of the new data structure are similar to that of the compact B-tree.

Received February 1991

1. INTRODUCTION

The B-tree structure of Bayer and McCreight is an effective data structure for organising and maintaining an index for a dynamically changing file on secondary storage.[3] A good discussion of B-trees and some of their variations is presented by Comer and Knuth.[5,10]

We use the term 'key' to mean a record key value as held in the B-tree index. The classical B-tree of order M is characterised by each non-leaf node (or page) except the root having at most $M-1$ keys and M descendants and having not less than $\lfloor M/2 \rfloor$ descendants. We assume the reader is familiar with conventional algorithms for maintenance of B-trees. For details, refer to Comer and Knuth.[5,10]

It is clear that a minimally filled B-tree guarantees storage utilisation of at least 0.5 in all nodes except the root. Such low storage utilisation has the following implications.

(1) It is expensive in terms of physical storage resources.

(2) When the B-tree nodes are traversed to yield a sequential key ordering, the number of nodes which must be read may be very large (compared to other possible index organisations, e.g. indexed sequential, VSAM).

(3) Random operations (key insertion, deletion and retrieval), on the B-tree are more expensive in some cases because, for a given number of keys, the tree may be one level higher than need be.

The average storage utilisation, however, tends to be better than the 0.5 minimum. Yao has shown that the leaf pages are expected to be 0.69 full on average if the distribution of keys is uniform.[16] However, according to a study by Quitzow and Klopprogge the 0.69 average utilisation is obtained only if the file is expanding and no deletions are being made.[15] When deletions and insertions take place with equal probability, the storage utilisation is expected to be close to 0.59.

Rosenberg and Snyder discuss time-optimality and space-optimality of B-trees.[13] A B-tree is called space-optimal if it has a minimal number of nodes for a given number of keys, and time-optimal if the expected number of nodes visited per key retrieval is minimal. Rosenberg and Snyder show that time-optimal B-trees tend to have nearly worst-case space utilisation, but the space-optimal B-trees are nearly time-optimal.[13] It is therefore desirable to aim for space-optimality, not just to save on storage but to be close to time-optimality.

Intuitively this result may be explained as follows.

(1) M is usually large (of the order 10–100) for practical applications.

(2) Time-optimality requires minimal height, which in turn requires high (near 1.0) space utilisation in the non-leaf nodes of the B-tree, but this implies poor average space utilisation in the leaf nodes. Since there are a very large number of leaf nodes compared to non-leaf nodes, the poorer utilisation dominates the total space utilisation.

(3) Space-optimality throughout all levels of the B-tree implies that the height will be close to minimal and hence good time performance may be expected.

2. B*-TREES AND OTHER B-TREE VARIATIONS

Several variations of B-tree have been suggested for improving the storage utilisation. The most well-known variant suggested by Knuth (Ref. 10, page 478) is known as a B*-tree. On insertion to a full node, splitting is delayed until one (or even both) of the adjacent sibling nodes is also full. If the two sibling nodes are not full, insertion to a full node results in a redistribution of keys and pointers between the sibling nodes.

When both left and right siblings are checked and involved in the redistribution, the worst case space utilisation is 0.75, and the average space utilisation has been empirically determined to be 0.87 (Ref. 10, page 479). Quitzow and Klopprogge also calculate storage utilisation when both sibling nodes are checked before splitting and obtain a figure of 0.826 utilisation if deletions and insertions are taking place with equal probability.[15] Checking both siblings is obviously somewhat more expensive than checking only one sibling. It is of course possible to extend the scheme so that more than two sibling nodes are checked before a split is made. Such schemes result in higher storage utilisation at the cost of increased checking of the sibling nodes and an

increasing expense when the keys are redistributed across a larger number of sibling nodes.

Culik, Ottmann and Wood have introduced m-ary dense B-trees in which all the sibling nodes may need to be looked at before an insertion is made.[6] The scheme is obviously very expensive in terms of number of nodes accessed and number of keys shifted, but it leads to almost 100% storage utilisation.

When the leaf nodes are becoming full in growing index, the B*-tree organisation would require frequent redistributions. Also when the index has achieved very high space utilisation, a small number of subsequent insertions will result in space utilisation going down substantially. This is because space optimality tends to be very unstable and, as shown by Klonk (1983), may be lost with index growth of less than 1% if the order of the B-tree is large (> 50). For example, Klonk (1983) shows that 1% growth in the number of keys in the index may reduce storage utilisation from 1.0 to 0.57 if the order of the B-tree is 150, and to 0.66 if the order is 75.

Rosenberg and Snyder suggest a compacting algorithm which could be used for achieving space-optimality at the end of (say) each day.[13] One of the disadvantages of this scheme is that if the index is growing the space utilisation may go down substantially soon after compacting due to the instability of space-optimality. Also in some applications it may not be feasible to accept the disruption caused by major index reorganisation. Henceforth we shall only consider schemes which support dynamic index updates without periodic restructuring of the whole index.

3. B-TREE WITH DEFERRED SPLITTING

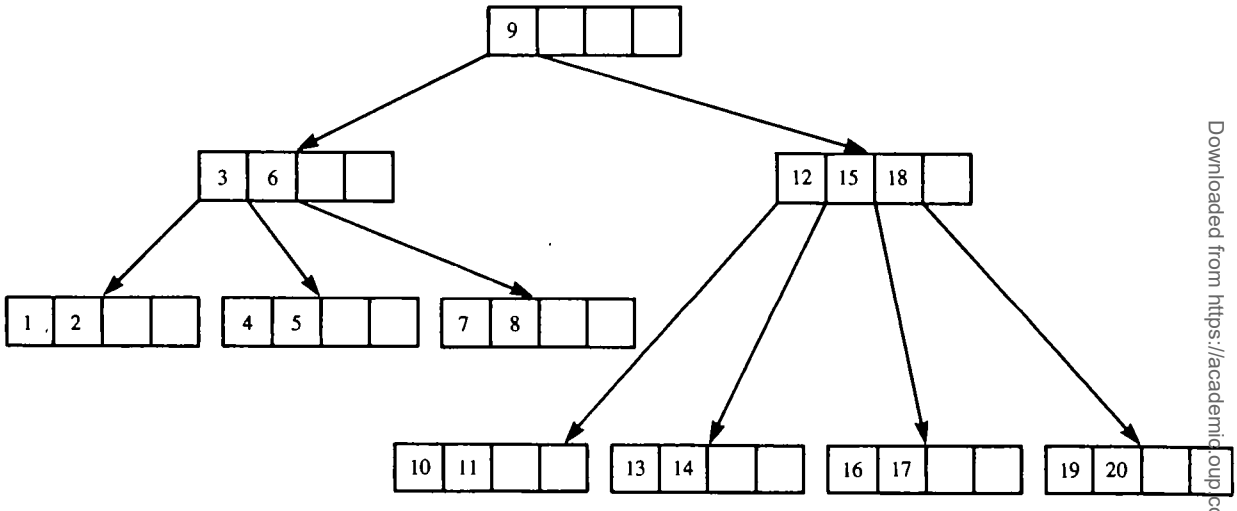
Culik *et al.*[6] observe that the simple B-tree scheme and m-ary dense B-trees lie at the extreme of a spectrum of B-tree structures characterised by (*a*) the extent to which node splitting is deferred and the cost of this deferment, and (*b*) expected space utilisation.

A technique to improve space utilisation over the simple B-tree, without incurring the deferment costs of m-ary dense B-trees is proposed in Ref. 7. This method has some similarity to the overflow strategy employed in linear hashing to avoid major reorganisation as the allocated pages of a file become full.[14] In the remainder of this section we formalise their data structure.

3.1 The method

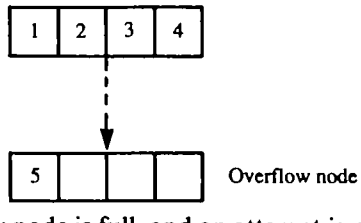
We illustrate the basic idea behind the scheme with an example. We will use the classical B-tree structure, although B-tree variations requiring all keys to be resident in the leaves are more commonly used.

Initially consider a conventional B-tree of order 5 and build the tree by inserting keys 1, 2, 3,...,20 in that order. It is known that building a B-tree by such ordered insertions results in a worst-case B-tree. The final B-tree with the 20 keys is

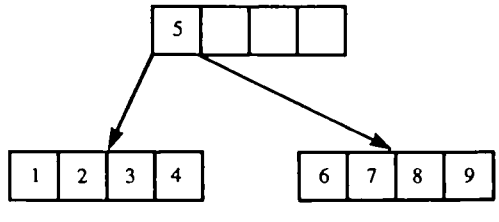


We have 10 nodes and 40 key positions, but only 20 key values, and therefore the storage utilisation is 0.5.

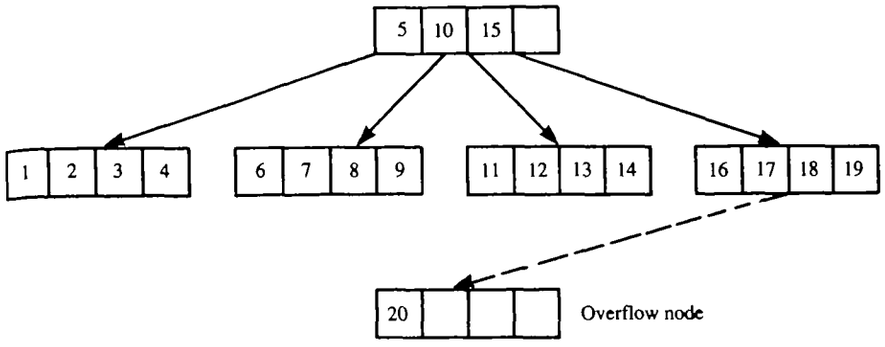
Now consider the proposed technique which employs deferred splitting. Once a leaf node is full, rather than splitting it immediately, an overflow node is added to it and the splitting is deferred. The sequence of keys is strictly ascending in each leaf node and its associated overflow node. For simplicity we shall consider all nodes (internal, leaf and overflow) to have a homogeneous structure. This may represent suboptimal space utilisation in the leaf and overflow nodes, however, an improvement in storage utilisation can be obtained by allocating an overflow node to a group of leaf nodes (see below). Using this technique we have the following configuration after inserting key 5.



Once the overflow node is full, and an attempt is made to insert a new key, we split (or reorganise) the leaf node and the overflow node into two leaf nodes of the B-tree. Therefore after the insertion of key 9 we have two leaf nodes and no overflow nodes.



Further insertions directed to a full node result in an overflow node being attached to that node. Again, when the overflow node is full, it and the leaf node to which it is attached are reorganised into two leaf nodes. This results in the following final B-tree.



There are 6 nodes (1 internal node, 4 leaf nodes and an overflow node) and a storage utilisation of $20/24 = 0.83$.

Table 1 illustrates that over a range of insertion sequences (with key values being 1 to 20), the overflow technique with deferred splitting method provides space utilisation which is never worse but often better than the conventional method for this simple example.

The idea of attaching one overflow node to each full leaf node may be generalised. If an insertion is made to a full leaf node, rather than allowing a separate overflow node to each leaf node, we may allow one overflow node to be shared by g ($g \geqslant 1$) leaf nodes. Such sharing of overflow nodes is often used in hashing schemes to minimize the space overhead involved in using overflow nodes.

Consider the case where insertions are made to an empty B-tree. Splitting will not occur until the root node and its associated overflow node are full. After the split there will be two full leaf nodes. The next split occurs when both leaf nodes and their shared overflow node (assuming $g > 1$) are completely full; after the split there will be three full leaf nodes and two keys in the root node. This process may be repeated until all the g leaf nodes sharing an overflow node are full and that the overflow 'node itself is full, and another insertion is made; at which time the g leaf nodes, the overflow node and the new key are reorganised into $g+1$ leaf nodes, with one key migrating to a parent (non-leaf) node. Assuming that no deletions are taking place, repeated application of this insertion procedure guarantees that the leaf nodes in the file will always be full and the only unused space will be in the overflow nodes!

Therefore, in the simple case, the storage utilisation will be very high and has a guaranteed worst case of $g/(g+1)$. Unfortunately the real-life situation is somewhat more complex. A file undergoes deletions as well as insertions and therefore the leaf nodes cannot be expected to be full without incurring substantial reorganisation costs at each deletion. In the following sequel we show that it is possible to design algorithms which will ensure high storage utilisation of the leaf nodes without substantial reorganisation cost.

In order for the B-tree with overflow nodes for deferred splitting to be useful as a secondary index structure we need to answer the following questions.

(*a*) How do we always find exactly g nodes to share an overflow node?

(*b*) How do we deal with deletions in the B-tree index?

We will concern ourselves only with the leaf nodes in the B-tree, since a very large majority of the keys reside there for B-trees with typical orders and numbers of keys. Further, all insertions are directed to leaf or overflow nodes and any subsequent changes to internal nodes (as a consequence of splitting) are identical to the simple B-tree case. For the method to work, it is not essential to have a fixed value of g. Also during reorganisation we deal with a group of nodes together, and the group includes g leaf nodes and possibly a single shared overflow node. In order for the reorganisation to be local to the tree, we ensure that these g leaf nodes have the same parent, hence the propagation of the reorganisation is localised to a subtree. Further, g is constrained by the order of the B-tree and we have

$$1 \leqslant g \leqslant G \leqslant M-1,$$

where G is a design parameter.

Table 1. Comparative space utilisation for a small index

Key sequence	Space utilisation: Conventional method	Space utilisation: Deferred splitting
1 2 5 6 7 3 4 10 11 12 8 9 15 16 17 13 14 18 19 20 ('best case' for the conventional method)	0.833	0.833
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ('worst case' for the conventional method)	0.5	0.833
1 2 3 4 6 7 8 9 10 12 13 14 15 16 17 18 19 20 11 5 ('worst case' for deferred splitting)	0.556	0.714

We shall define n to be the number of keys in the nodes of a particular group. In the classical B-tree organisation sibling nodes may be merged during deletion to guarantee minimum space utilisation of 0.5. In the proposed method an analogous scheme is required whereby a reorganisation procedure in the group is invoked whenever the number of keys in a leaf node falls below the threshold value $M-1-s$, where s is also a design parameter such that

$$0 < s \leqslant \lceil (M-1)/2 \rceil.$$

Suitable values of G and s may be computed once the value of M and minimum storage utilisation have been decided.

The maximum and minimum number of keys (n_{max} and n_{min}) in a group therefore are given by

$$n_{max} = (g+1)(M-1)$$

and

$$n_{min} = g(M-1-s).$$

Formally a B-tree with overflow nodes for deferred splitting can be defined as follows.

(1) The structure of the tree from the root of the tree to the parents of the leaf nodes is the same as for a classical B-tree.

(2) A leaf node contains k keys such that

$$M > k \geqslant M-1-s,$$

where s is a positive integer design parameter, and that determines the storage utilisation. $M-1-s$ is called the 'threshold value' of the leaf node.

(3) Each group of g leaf nodes share an overflow node. The g leaf nodes and the associated overflow node is called a 'group'. The value of g is such that

$$G \geqslant g \geqslant G/2.$$

G is called the order of the group. All leaf nodes in a group have the same parent.

(4) G, s, M and minimum storage utilisation in the leaf nodes (η_{min}) are related as follows:

$$s = (1-\eta_{min})(M-1).$$

During reorganisation (see below), we allow almost $s/2$ keys from a leaf node to reside in the corresponding overflow node. Hence an overflow node can at most be shared by G leaf nodes, where G is given by

$$G = \frac{2(M-1)}{s}.$$

(5) If an attempt is made to insert a key into a node that has $M-1$ keys, we say that node **overflows**. If an attempt is made to delete a key from a node with $M-1-s$ keys, we say that the node **underflows**. When a node overflows and the group overflow node is full, we say that a **group overflow** has taken place. Similarly when a node underflows and the group overflow node has no key belonging to the node, we say that a **group underflow** has taken place.

The retrieval, insertion and deletion algorithms are quite similar to that of B-tree except that we have to take into account the keys in the overflow node of the group. The retrieval operation proceeds in the normal way as in the B-tree, and if the key is not found in the corresponding leaf node, the corresponding overflow node is searched for the key before reporting a failure. The difference between B-tree and B-tree with overflow nodes for deferred splitting is in the insertion and deletion operations and the manner of handling overflow and underflow conditions. In the following section we describe the group reorganisation when the group nodes overflow during insertion operation. A similar reorganisation procedure can be devised for dealing with group underflow condition.

3.2 Group reorganisation

Group reorganisation is required when a group overflow or group underflow takes place. There are several issues that must be considered when a group is reorganised. These include the following.

(*a*) Should the number of nodes after reorganisation be the same as before?

(*b*) How many keys should be placed in the overflow node as a result of reorganisation?

Let the number of keys in the group when it is being reorganised be n, which will be approximated to be the average of the minimum and maximum number of possible keys in a group. Let g be the number of leaf nodes in the reorganised group. It is clear that if n is close to n_{min} or n_{max}, another reorganisation could follow the current one. Since a group reorganisation is expensive, we should attempt to choose a value of g such that n is not close to the values n_{min} and n_{max}.

Also when the n keys are being distributed amongst the leaf nodes and the overflow node, it is desirable that the number of keys in each leaf node is not close to $M-1$ or $M-1-s$ since a subsequent overflow or an underflow of a leaf node would lead to a node reorganisation which incurs additional cost of reading and writing of the overflow node.

A simple scheme for computing the value of g during reorganisation is as follows. We have

$$n_{ave} = \frac{(n_{max}+n_{min})}{2} = g\left(M-1-\frac{s}{2}\right)+\frac{(M-1)}{2},$$

and therefore we could choose g to be

$$\frac{n}{(M-1-s/2)} - \frac{(M-1)}{2(M-1-s/2)} \approx \left\lfloor \frac{n}{(M-1-s/2)} \right\rfloor$$

We could then put $(M-1-s/2)$ keys in each of the g leaf nodes and put the remaining keys in the overflow node. The keys in the overflow node will be a uniform mix of keys from all g leaf nodes. If c is the number of keys that exist in the overflow node from a leaf node its value is such that

$$0 \leqslant c \leqslant \left\lceil \frac{(M-1-s/2)}{g} \right\rceil$$

or, substituting $M-1$ from the definition of s,

$$0 \leqslant c \leqslant \left\lceil \frac{s(G-1)}{2g} \right\rceil.$$

A group reorganisation would require reading of all the nodes in the group and writing of all the new nodes after redistribution of keys. This is likely to lead to changes in the parent node as well.

A group reorganisation may lead to a group split if the new $g > G$, or group merging if the computed value of

$g < G/2$. During group merging a brother group with the same parent is selected. Normally the right-hand group is chosen if it exists, otherwise the left brother group is selected and one of the following actions is taken.

(i) If number of leaf nodes in the brother group is more than $G/2$, the nodes in the two groups are equally divided into two groups.

(ii) If the number of leaf nodes in the brother group is $G/2$, the two groups are merged together to form a single group.

That completes the reorganisation algorithm.

4. PERFORMANCE ANALYSIS

This section evaluates the performance of the new data structure quantitatively. In particular, the expression for average storage utilisation and the cost of insertion and deletion will be derived. A random walk model will be used to analyse the behaviour of the proposed structure.

4.1 Storage utilisation

The average number of keys in a group is given by

$$n_{ave} \approx (M-1)g.$$

The storage utilisation of the new data structure therefore on the average would be

$$\eta_{ave} = \frac{g}{g+1},$$

where $G/2 \leqslant g \leqslant G$ is the average number of leaf nodes in a group. The number of leaf nodes in a group varies in the same way as the number of keys in a leaf node of a B-tree of order G. Therefore the average value of g is the same as the average number of keys in a leaf node of a B-tree of order G. That is,

$$g_{ave} \approx \frac{2G}{3}$$

Substituting g_{ave} in η_{ave}, we obtain the average storage utilisation of B-trees with overflow nodes for deferred splitting as

$$\eta_{ave} = \frac{2G}{2G+3}.$$

4.2 Cost of retrieval

The number of keys, on an average, in the overflow nodes of the new tree structure is equal to the number of vacancies in the leaf nodes. Therefore the overflow keys could be accommodated in the leaf nodes and a compact B-tree would be obtained. The depth of the leaf nodes of the tree would therefore be

$$h = \log_M (N+1)$$

where N is the total number of keys in the tree.

The average cost in number of reads for a random retrieval is

$$\omega = \log_M (N+1) + \frac{c}{M-1},$$

where c is the average number of keys in the overflow node from a leaf node.

The height of a B-tree with overflow nodes for deferred splitting varies in the following range

$$\log_M (N+1) \leqslant h \leqslant 1 + \log_{M/2} \frac{N+1}{2}$$

4.3 Cost of insertion

An approximate mathematical analysis of the additional cost of maintaining the new data structure can be made by means of a random walk model. The random walk model to be used is described in Refs 1 and 2. Under the conditions of uniform distribution of key values in the tree and random insertions and deletions, all the groups and their leaf nodes are expected to behave in a similar fashion. A leaf node initially contains $M-1-c$ keys, where c is a positive integer such that $0 < c < s$. Let p, q and r be the probabilities of insertion, deletion and retrieval respectively.

We first consider the case when the tree is undergoing only insertions and no deletions. We are interested in finding out the number of insertions made before a leaf node, starting with $M-c$ keys in it, overflows. Let $T(M-1-c, M)$ be the number of insertions needed for the random walk, starting at height $M-c$, to reach height M. The mean value of $T(M-1-c, M)$ may be derived using the techniques presented in Ref. 2, and we obtain the following

$$E[T(M-1-c, M)] = \frac{c+1}{p},$$

where $E[T(M-1-c, M)]$ is the expected value of the number of insertions made before the random walk reaches the height M.

For the insertions directed to a group under consideration we can say that $p = 1/g$ and $q = 0$.

After a certain number of insertions (say I) the group will need group reorganisation. After this reorganisation the group comes back to the same configuration as it started before I insertions. However, the value of g would have gone up by one. We like to compute the average cost per insertion as

$$\omega = \frac{[\text{Additional cost incurred in } I \text{ insertions}]}{I}.$$

The additional cost we define as the i/o cost incurred during insertion, in addition to the usual cost of reading all nodes on the path from the root to the leaf node and writing of the leaf node that was updated. The additional cost in any of the I insertions can be incurred due to one of the following three factors.

(1) Additional cost incurred when an insertion is directed to the overflow node. This extra cost involves reading the overflow node.

(2) Additional cost incurred when an insertion is directed to an already full leaf node. This extra cost, because of this node reorganisation, involves reading and writing of the overflow node.

(3) Additional cost incurred due to the group reorganisation. This cost is incurred during the last of the I insertions. The additional cost incurred due to this is reading and writing of all other $g-1$ leaf nodes in the group and the overflow node.

In the sequel we will compute the upper bound for the cost of insertion.

The additional cost due to the first factor can be found if we know how many out of I number of insertions will be directed to the overflow node. The additional cost due to the second factor can also be similarly found out if we know the number of node reorganisations during these I insertions. We know that at the start of insertions, out of $M-1$ keys of a leaf node, c keys reside in the overflow node. If v is the probability that an insertion is directed to the overflow node, the value of v at the start of insertions is

$$v = \frac{c+1}{M}.$$

The minimum and maximum values of v during the course of I insertions are as follows:

$$v_{\min} = \frac{c+1}{M+c} \quad \text{and} \quad v_{\max} = \frac{2c+1}{M+c+1}.$$

The total additional cost incurred due to the first factor in I insertions will be Iv.

The additional cost incurred because of the second factor can be calculated if we know the number of node reorganisations that occurred during the course of I insertions. The number of node reorganisations depend on the vacant key spaces available in the overflow node. At the start the overflow node has $M-1-gc$ vacancies. This vacant space is further reduced because of insertions directed to the overflow node. Therefore the number of vacancies in the overflow node available for leaf node reorganisations, in I insertions, will be

$$M-1-gc-Iv.$$

According to the random walk model, all the g leaf nodes would have overflowed or would be on the verge of overflowing after $g(c+1)$ insertions directed to the leaf nodes. A node reorganisation involves moving c keys from the overflowing leaf node to the overflow node. If the vacancies available in the overflow node are less than c, as many keys as the vacancies available are moved. The number of node reorganisations R in I insertions will therefore be

$$R = \frac{M-1-gc-Iv}{c}$$

or

$$R = \frac{M-1-gc-Iv}{c}+1 \quad \text{(at worst)}.$$

The value of I can be computed from the equation

I = Insertions into leaf nodes + Insertions into overflow node

$$I = g(c+1)+Iv$$

or

$$I = \frac{g(c+1)}{1-v}$$

and therefore

$$I_{\min} = \frac{g_{\min}(c+1)}{1-v_{\min}}.$$

The average additional cost incurred per insertion can now be evaluated by dividing the cumulative additional cost, due to all the three factors, over I insertions by I as follows

$$\omega = \frac{Iv+2R+2g}{I}.$$

Substituting R from above and after simplifying we have

$$\omega = v+\frac{2(M-1+c)}{Ic}-\frac{2v}{c}.$$

The worst value of ω can be obtained as follows

$$\omega_{\text{worst}} = v_{\max}+\frac{2(M-1+c)}{I_{\min}c}-\frac{2v_{\min}}{c}.$$

Substituting the values in the above equation we have

$$\omega_{\text{worst}} = \frac{2c+1}{M+c+1}+\frac{4(M-1+c)(M-1)}{cG(c+1)(M+c)}-\frac{2(c+1)}{c(M+c)}.$$

The expression for ω_{ave} can be computed using v_{ave} and g_{ave}. The final expression is as follows.

$$\omega_{\text{ave}} = \frac{(c+1)(c-2)}{Mc}+\frac{3(M-1-c)(M-1+c)}{MGc(c+1)}.$$

Note that ω is inversely proportional to G. That means higher values of G will give better performance for constantly growing B-tree with overflow nodes for deferred splitting.

4.4 Cost of insertions in B-tree

Yao has shown that

$$n(N) = \frac{N}{(M-1)\ln 2},$$

where M is the order of the B-tree and $n(N)$ is the average number of B-tree nodes obtained after N random insertions, starting with an empty tree and no deletions. The average number of splits after N random insertions is therefore $n(N)-1$. The additional cost incurred for a split operation on an average is writing the new leaf node and the parent node. The average additional cost incurred in building a B-tree after N random insertions is therefore

$$\omega = \frac{2(n(N)-1)}{N}$$

or

$$\omega = \frac{2}{(M-1)\ln 2} \quad \text{for} \quad N \gg M.$$

4.5 Cost of the new tree with insertions and deletions

We shall now consider the case when the tree is undergoing both insertions and deletions. We are interested in the expected number of transactions before a node, starting with $M-1-c$ keys, overflows or underflows. This problem is identical to the classical run problem with two absorbing barriers (see Ref. 1, pages 348–349). In our problem one of the barriers is when the node has M keys and the other barrier is when the node has $M-2-s$ keys. The values of c and s are such that $0 < c < s < M/2$. The expected duration of the random

walk hitting either barrier, when $p \neq q$, is given by the following expression:

$$E[T] = \frac{(c+1)}{(p-q)} - \frac{s+2}{(p-q)} \frac{1-(p/q)^{c+1}}{1-(p/q)^{s+2}}.$$

The expression simplifies to the following when $p = q$ (using L'Hospital's rule)

$$E[T] = (c+1)(s-c+1).$$

4.5.1. Case I: $p = q$

We now consider the environment in which the probabilities of insertion and deletion to the tree are same (namely $p = q$). Consider a leaf node with $(M-1-c)$ keys initially. We are interested in the expected number of transactions after which the node is either overflows or underflows. The expression for the expected duration of the random walk, on a leaf node, hitting either barrier when $p = q$ is given by the equation above.

As noted earlier, the additional cost of maintaining the B-tree with overflow nodes for deferred splitting results mainly when a leaf node overflows or underflows. Therefore the additional cost will be least if $E[T]$ is maximum. In the above expression the value of $E[T]$ is maximum when $c = s/2$.

We shall now derive the expression for the additional cost of the tree per transaction. A transaction in this case is either insertion or deletion. The factors affecting the additional cost are the same as listed in the previous section. In this case the evaluation of the additional cost due to the first and the third factors remain the same. For the second factor we make a worst-case assumption here. That is, the group will need reorganisation after all the g leaf nodes have hit either barrier. Some of them may overflow and the others will underflow. In other words the number of node reorganisations will be

$$R = g.$$

From the random walk model, the mean number of transactions directed to the leaf nodes, after which all the g leaf nodes would have hit either barrier, with $c = s/2$, is

$$(c+1)^2 g.$$

The expected total number of transactions (say I), directed to the group, after which the group will need reorganisation, can therefore be obtained from the following equation

$$I = Iv + g(c+1)^2 \quad \text{or} \quad I = \frac{g(c+1)^2}{1-v},$$

where v is the probability that a transaction will be directed to the overflow node. It starts with value $(c+1)/M$ and its minimum and maximum values in this case are as follows

$$v_{\min} = 0, \quad v_{\max} = \frac{2c+1}{M+c+1} \quad \text{and} \quad v_{\text{ave}} = \frac{c+1}{M}.$$

The expression for ω in this case is as before:

$$\omega = \frac{Iv + 2R + 2g}{I}.$$

Substituting the value of R we get

$$\omega = v + \frac{4(1-v)}{(c+1)^2}.$$

The worst value of ω in this case will be

$$\omega_{\text{worst}} = v_{\max} + \frac{4(1-v_{\min})}{(c+1)^2}.$$

Substituting the values from above we get

$$\omega_{\text{worst}} = \frac{2c+1}{M+c+1} + \frac{4}{(c+1)^2}.$$

The final expression for the average additional cost per transaction, using v_{ave}, therefore is

$$\omega_{\text{ave}} = v_{\text{ave}} + \frac{4(1-v_{\text{ave}})}{(c+1)^2}.$$

Substituting the values we have

$$\omega_{\text{ave}} = \frac{c+1}{M} + \frac{4(M-c-1)}{M(c+1)^2}.$$

4.5.2 Case II: $p \neq q$ and $p > q$

We shall now consider the case when $p \neq q$ and both p and q are non-zero. If the value of p is sufficiently greater than q, the barrier at $M-s-2$ can be ignored. As a result the expected value expression can be simplified[2] to

$$E[T] = \frac{c+1}{p-q}.$$

The derivation of the expression for the additional cost in this case is identical to the case when $q = 0$. The only difference is that the expression for the mean expected value of number of transactions after which the random walk reaches height M now becomes

$$E[T(M-1-c, M)] = \frac{c+1}{p-q}.$$

The total number of transactions, directed to the group under consideration, after which the group will need reorganisation can be obtained from the following equation:

$$I = Iv + \frac{c+1}{p-q} \quad \text{or} \quad I = \frac{c+1}{(p-q)(1-v)}.$$

The final expression for ω thus becomes

$$\omega = v + \frac{2(p-q)(1-v)(M-1+c)}{c(c+1)} - \frac{2v}{c}.$$

The final expression for the worst value of ω will be

$$\omega_{\text{worst}} = v_{\max} + \frac{2(p-q)(1-v_{\min})(M-1+c)}{c(c+1)} - \frac{2v_{\min}}{c}.$$

Substituting the values we have

$$\omega_{\text{worst}} = \frac{2c+1}{M+c+1} + \frac{2(p-q)(M-1)(M-1+c)}{c(c+1)(M+c)} - \frac{2(c+1)}{c(M+c)}.$$

The final expression for the average additional cost, using v_{ave}, therefore becomes

$$\omega_{ave} = v_{ave} + \frac{2(p-q)(1-v_{ave})(M-1+c)}{c(c+1)} - \frac{2v_{ave}}{c}$$

or

$$\omega_{ave} = \frac{c+1}{M} + \frac{2(M+c-1)(M-c-1)(p-q)}{M^2 c(c+1)} + \frac{2(c+1)}{Mc}.$$

5. SIMULATION RESULTS

In order to evaluate the correctness of our mathematical model, we performed some simulation studies on the proposed data structure, and the results are summarised in Tables 2–4. We now present the results of the simulation and compare them with the theoretical results.

The number of keys used in the simulation for a given order is large enough for at least three levels in the tree to be obtained. For instance, 20,000 keys were used to build a tree of order 91. To simulate the new tree with $p = q$, the insertions and deletions were performed alternately. The number of insertions and deletions was three times the number of keys present in the tree.

Table 2. Space utilisation

B-tree with deferred splitting					B-tree	
			η		η	
G	$s = 2c$	M	Simulation	Theoretical	Simulation	Theoretical
10	18	91	0.87	0.87	0.68	0.69
8	16	65	0.84	0.84	0.68	0.69
6	14	43	0.80	0.80	0.68	0.69
4	12	25	0.73	0.73	0.68	0.69

Table 3. Average additional cost per insertion with no deletion ($q = 0$)

B-tree with deferred splitting					B-tree	
			ω		ω	
G	$s = 2c$	M	Simulation	Theoretical	Simulation	Theoretical
10	18	91	0.32	0.37	0.03	0.03
8	16	65	0.36	0.42	0.04	0.04
6	14	43	0.44	0.48	0.07	0.07
4	12	25	0.56	0.57	0.12	0.12

Table 4. Average additional cost per transaction when insertion and deletions are done with equal probability ($p = q$)

B-tree with deferred splitting					B-tree	
			ω		ω	
G	$s = 2c$	M	Simulation	Theoretical	Simulation	Theoretical
10	18	91	0.12	0.13	0.04	—
8	16	65	0.14	0.16	0.06	—
6	14	43	0.18	0.21	0.10	—
4	12	25	0.24	0.31	0.20	—

6. CONCLUSION

B-tree is considered as the *de facto* standard access path for providing random access to databases. However, conventional B-trees and their variants are not space-optimal; those that are space-optimal are also nearly time-optimal. With that result in mind, this paper proposed a method to improve the space utilisation of B-trees. A new data structure is proposed where overflow nodes are attached to the leaf nodes of the conventional B-trees. The data structure is formalised and its performance is analysed both mathematically and by simulation.

From the results it can be seen that the proposed data structure can achieve high storage utilisation compared to the conventional B-tree even for moderate orders. Trees of the order 91 can achieve a storage utilisation as high as 89 per cent. However, this increase in storage utilisation does come with a penalty in the form of the effort required during the reorganisation of the tree structure. The analysis shows that the cost of insertion is much higher than that of conventional B-trees for applications which involve only insertion. However, the performance of the new data structure is comparable to that of conventional B-trees in situations where the insertions and deletions are intermixed and occur with near-equal probabilities. Such situations are more practical. Because of higher storage utilisation the proposed data structure can provide improved response time to retrieval queries without degrading the performance of index maintenance.

Acknowledgements

The author thanks Professor Gupta for introducing the problem of deferred splitting in B-trees, Professors Wallace and Deogun for their help in the mathematical analysis and Mr Bajwa for allowing him to use his B-tree simulation programs. Finally, the author would like to thank the referees for their comments and suggestions.

REFERENCES

1. W. Feller, *An Introduction to Probability Theory and its Applications*, 3rd edn, John Wiley and Sons, New York (1968).
2. P. Heyman Daniel, Mathematical models of database degradation. *ACM Transactions on Database Systems* 7 (4), 615–631 (1982).
3. R. Bayer and E. McCreight, Organization and maintenance of large ordered indexes. *Acta Informatica* 1, 173–189 (1972).
4. R. Bayer and K. Unterauer, Prefix B-trees. *ACM Transactions on Database Systems* 2, 11–26 (1977).
5. D. Comer, The ubiquitous B-tree. *ACM Computing Surveys* 11, 121–137 (1979).
6. K. Culik II, T. Ottmann and D. Wood, Dense multiway trees. *ACM Transactions on Database Systems* 6, 486–512 (1981).
7. G. K. Gupta and K. J. McDonell, B-trees with deferred splitting. *Proceedings of the 7th Australian Computer Science Conference* 6 (1), 26.1–26.9 (1984).

8. G. Held and M. Stonebraker, B-trees reexamined. *Communications of the ACM* **21**, 139–143 (1978).
9. D. G. Keehn and J. O. Lacy, VSAM data set design parameters. *IBM Systems Journal* **3**, 186–212 (1974).
10. D. E. Knuth, *The Art of Computer Programming*, Volume 3: *Searching and Sorting*, esp. chap. 6. Addison-Wesley, London (1973).
11. R. E. Miller, N. Pippenger, A. L. Rosenberg and L. Snyder. Optimal 2-3 trees. *SIAM Journal of Computing* **8**, 42–59 (1979).
12. A. L. Rosenberg and L. Snyder, Minimal comparison 2,3 trees. *SIAM Journal of Computing* **7**, 465–480 (1978).
13. A. L. Rosenberg and L. Snyder, Time and space-optimality in B-trees. *ACM Transactions on Database Systems* **6**, 174–183 (1981).
14. M. Scholl, New file organizations based on dynamic hashing. *ACM Transactions on Database Systems* **6**, 194–211 (1981).
15. K. H. Quitzow and M. R. Klopprogge, Space utilisation and access path length in B-trees. *Information Systems* **5**, 7–16 (1980).
16. A. C. Yao, On random 2-3 trees. *Acta Informatica* **9**, 159–170 (1978).

Correspondence

Sir,

The algorithm in Fig. 1 is the square-move version of the cubic generating algorithm given in this Journal, vol. 11, p. 120. The algorithm is similar to the conic generating algorithm (see *Fundamental Algorithms for Computer Graphics*, 1985, vol. 17, pp. 219–237) in that it permits the plotter to use square moves only. (It is useful in rendering applications where it is necessary to visit, in turn, all the pixels intersected by a given cubic curve.)

Initial conditions for the cubic form are:

$$k+2vx-2uy-\alpha y^2-\beta x^2-2\gamma xy - \frac{rx^2}{3}-\frac{sy^2}{3}-px^2y-qxy^2=0.$$

(We are here assuming that u and v are positive, but, if they are not, the algorithm responds by searching through the quadrants.)

$$L_1 = 2r$$
$$L_2 = 2p$$
$$L_3 = 2q$$
$$L_4 = 2s$$
$$K_1 = 2\beta - r + p$$
$$K_2 = 2\gamma$$
$$K_3 = 2\alpha - s + q$$
$$a = 2u + \gamma + \frac{s}{12} + \frac{p}{4}$$
$$b = 2v - \gamma - \frac{r}{12} - \frac{q}{4}$$
$$d = k - \frac{\alpha}{4} - \frac{\beta}{4} + \frac{\gamma}{2} - \frac{a}{2} + \frac{b}{2}.$$

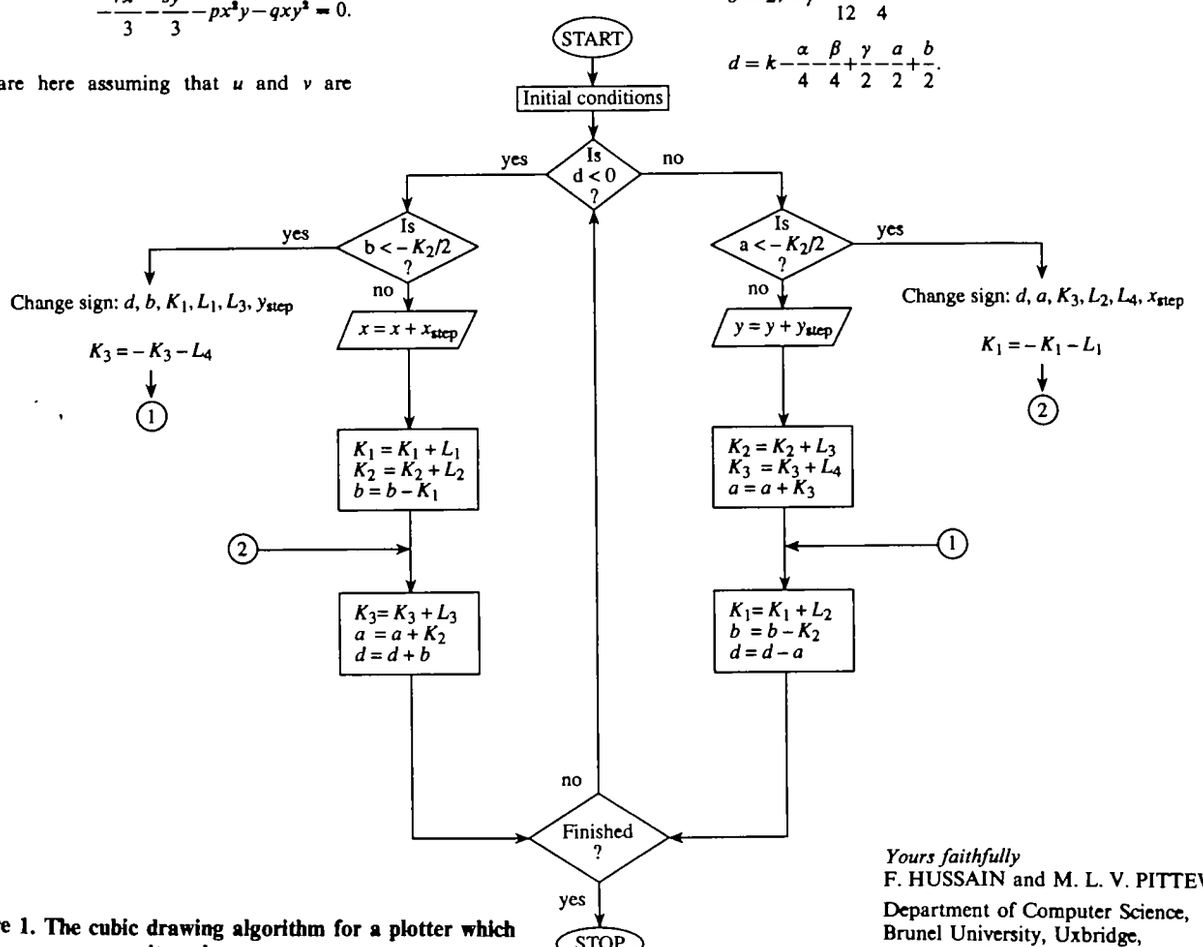


Figure 1. The cubic drawing algorithm for a plotter which permits only square moves.

Yours faithfully

F. HUSSAIN and M. L. V. PITTEWAY

Department of Computer Science,
Brunel University, Uxbridge,
Middlesex, UB8 3PH.