ABSTRACT: This is a proposal for a new data-structure called chained B-trees (CB-trees). CB-trees exhibit a superior access cost curve compared to B-trees. They provide the same amount of space utilisation as B-trees and are not any more expensive to build. In this paper we define CB-trees and study their performance vis-a-vis B-trees through extensive simulation studies. Simulations were done through a novel technique which allows large random trees to be simulated in core.

1. INTRODUCTION: B-trees proposed by Bayer & McCreight [Bayer 72] are widely used for organizing large indices. Simple maintenance algorithms, reasonable space utilisation and logarithmic access cost are the main reasons for their popularity. Several variants of the B-tree data-structure appeared in the literature, (e.g. B*-Trees, B+ Trees [Knuth 73, COMER 79], Digital B-trees [Lomet 81], Signature Trees [PRABHAKAR 83] etc.). The main thrust of these variants is in improving space utilisation and/or access costs. We propose here yet another data-structure based on the B-tree model. We call it a Chained-B-tree or a CB-tree for short. CB-trees exhibit an access cost curve (number of keys vs. average height) superior to that of a B-tree, provide the same amount of space utilisation and comparable construction cost. Nodes in a CB-tree might span over two disk pages and accessing all keys in such a node requires two page fetches. Thus all leaf nodes nodes are not at the same number of page fetches away from the root. However, they are different from height-balanced-multiway trees [Pateli 79].

In the next section we explain the motivation behind modifying the basic B-tree definition. Section 3 defines and discusses CB-trees. Here we examine them in relation to other data structures proposed in the literature. A novel technique to simulate in core large randomly built tree structures is used to examine the performance of CB-trees vis-a-vis B-trees. In section 4 we briefly describe this technique and present simulation results. Section 5 presents a summary and conclusions to the paper.

2. MOTIVATION: A B-tree of order n has the following properties:
1: Every node except the root will have at least n keys and at most 2n keys.
2: A node with i keys will have (i+1) descendants.
3: The root may have as few as 2 descendants.
4: All leaves are at the same distance from the root.

A new key to be inserted is accommodated in a leaf node. If the leaf node overflows (contains more than 2n keys) it is split into two nodes each with n keys and the centre key is pushed up to be accommodated at the parent node.
The tree grows in height when the root splits, the average height going up by one. Then the average height stays almost constant till the next split of the root. Variation of average height with number of keys can be seen in fig.3.1.

We observe that when the root is split to accommodate an upcoming key, all sub-trees emanating from the root are penalised by increasing their height. This is necessitated by the restriction that all leaves be at the same distance from the root. If we relax this constraint, one way to accommodate an upcoming key is to chain a new node to the root and absorb it there as shown in fig.1.

Notice that only sub-trees $m$ and $(m+1)$ have their height increased by 1 and the rest of the sub-trees retain their earlier height. This results in a tree whose average height is less than if the node has been split. With this in mind we define the CB-tree.

3.1 CB-tree: In a CB-tree of order $n$:
1. Every node is either a simple node (encompasses one disk page) or a compound node (encompasses two disk pages).
2. Every node except the root is at least half full. That is, a simple node will have a minimum of $n$ keys and a maximum of $2n$ keys. A compound node will have at least $(2n+1)$ keys and almost $4n$ keys.
3. The root may have only one key (2 descendants).
4. All leaf nodes have the same number of keys (either simple or compound) in the path to the root.

In a compound node the two pages are referred to as the main and twin. Disk pages for these two pages need not be contiguous and can be assigned from anywhere on the disk. The twin can be reached only via the main page. Hence to access entries on a twin an extra page fetch is necessary. Leaf nodes are equi-distant from the root in node count but can differ by a ratio of 1:2 in page fetches.

A key insertion is done as in a B-tree observing the following rules:
Rule 1: Do not split an overflowing simple node if doing so will cause the root to split.
Rule 2: Always split an overflowing compound node.

Rule 3: Split a compound node even if it is not full when doing so will not cause the root to split.

When all the nodes in the path to the root are saturated a split at the leaf also splits the root. Rule 1 indicates, at such a time do not split the leaf but chain a new page converting it into a compound node. We observe that this operation retains the average height of keys in a CB-tree close to the height before insertion whereas a B-tree would have grown in height by 1. Thus in a CB-tree first the leaf nodes acquire chains and become compound nodes.

According to rule 2, a compound node is always split if it overflows. This might result in the root splitting if it is already a compound node which is full.

It is possible to have compound nodes in the tree even when their ancestor nodes are not saturated. This can occur as follows: Consider a node "a" with "x" and "y" as two of its descendants. Node "a" is saturated and so are its ancestors. Now "x" and "y" overflow and are converted to compound nodes "x-x'" and "y-y'. If the compound node "x-x'" again overflows it is split into two simple nodes. A key be pushed up and node "a" is converted to a compound node "a-a'". One can see that the compound node "a-a'" can accommodate more keys but it has a descendant which is also a compound node. In such a situation when the next key insertion occurs in "y-y'", it is split into two simple nodes nodes with the upcoming key accommodated at the compound node "a-a'". Fig.2 illustrates this sequence of events.

Among the data-structures reported in the literature of related interest are height-balanced-multiway trees [Papadimitriou 79], Symmetric Binary B-trees [Bayer 73, Wirth 77] and Digital B-trees [Lomet 81]. Height-balanced-multiway trees are like AVL trees [Knuth 73] where the height of all sub-trees at any node may differ by one. Symmetric binary B-trees have been proposed to avoid re-allocation of space to nodes which are anyway generally not full. Hence a 2-3 tree node with 2 keys and 3 pointers is constructed out of 2 nodes with 1 key and 2 pointers. Digital B-trees have nodes which can span over

Proceedings of the Tenth International Conference on Very Large Data Bases.

Singapore, August, 1984
CB-trees are different from all these data-structures. Unlike in a height-balanced-multiway tree the longest path length in a CB-tree can be twice as much as the longest path length in a neighbouring sub-tree. CB-trees are not higher order generalisations of symmetric binary B-trees either where an order 2n node could be composed by 2 order n nodes. This is so because when an order n node sets full in a symmetric binary B-tree it automatically gets chained to another order n node to provide a full node of order 2n. In a CB-tree such chains are made only under certain conditions. Digital B-trees are clearly different since they use prefixes (digital searching) and are not comparison based. But like in a CB-tree an overflowing node in a Digital-B-tree can be split or compound. A compound digital B-tree node occupies two consecutive disc pages. After doubling entries in its ancestor are modified so that only one of these two disc pages need be fetched at any time. The choice between doubling or splitting a node is decided by considering the utility of its ancestor and siblings. Note that in a CB-tree the decision to split or chain is arrived at by consulting the utility of its ancestors.

3.2 CB-tree Algorithms. Nodes in a CB-tree are composed of the following record structure:
```plaintext
cnode = record
  TWIN : boolean;
  KEYSHERE: 1..2n;
  KEYS : array [1..2n] of key-type;
  P : array [0..2n] of cnode
end;
```

The 'TWIN' field on a node will indicate if it is a compound node. For a compound node the TWIN flag on main is set to 'true' and P[2n] points to its twin. On a twin node 'TWIN' is always set to false. KEYSHERE is a count for the number of keys on the node.

3.2.1 SEARCH: Searching for a key in the tree in similar to searching in a B-tree. Fig.3 gives a function to search for a key in a CB-tree.

The main advantage of a CB-tree over a B-tree is in the average page fetches required for a search. In both the trees, a search is completed in O (log N) time, for N keys in the tree. In the worst case a B-tree search will take \( \log_2 N \) page fetches. For a CB-tree—the furthest node in the worst case may be \( 2 \log_2 (N+1) \) pages away. At the same time there will be leaf nodes nearer to the root than this worst case, reducing the average height. The average height is sometimes better than the average height of a B-tree. We do not yet have an expression for the expected average height of a CB-tree. Experiments were conducted to compare the average heights of random B-trees and CB-trees. The next section gives details of these experiments.

3.2.2 INSERTION: Key insertion in a CB-tree is handled in such a way that overflowing nodes penalize as little as possible the rest of the tree. As explained before, an overflowing simple node sets split or sets converted to a compound node depending on its ancestors' occupancy. An overflowing compound node is always split. A compound node may sometimes set split even when it does not overflow. Fig.4 sketches the algorithm for inserting a key in a CB-tree. Here the predicate THERE_IS_SPACE_HERE is true for a node if it is not full. That is, for a simple node THERE_IS_SPACE_HERE is true if it has less than 2n entries and for a compound node less than 2n entries on its twin (the main will always have 2n entries). Similarly THERE_IS_SPACE_ABOVE is true for any node if for any of its ancestors THERE_IS_SPACE_HERE is true.

Like in a B-tree an insertion will take \( O(\log n) \) pages fetches. Accessing the entries on a twin page of a compound node requires an extra page fetch. If a key were to be inserted on the main page of a compound node the other node also need be fetched and rewritten.

3.2.3 Key Deletion: Deleting a key is very similar to key deletion in a B-tree. When a node underflows it is merged with its siblings to maintain 50% occupancy. An underflowing compound
node gets converted to a simple node.

4.1 SIMULATION RESULTS: In the absence of analytical results to evaluate CB-trees, we decided to examine their performance through simulation studies. Simulations were done using a novel technique which allows random trees of large size to be built in the main memory. (See [Yao 78] for the definition of random trees). This method of simulation does not store the key values in the tree. Only the structural information (number of keys at each node, in each sub-tree, pointers to the sub-tree) is retained. A key insertion is done by navigating through this to a page containing a random number in the range 1 to (Nh1) and updating its key count (There are N keys in the tree). This way an order n tree with N keys can be simulated for structural information in O(N/n) memory locations. For example a million key tree of order 50 can be accommodated in under 100k bytes.

Our objective in these simulation studies was to examine CB-tree vis-a-vis B-trees for the following performance measures:

1. Average height
2. Storage utilisation
3. Construction cost

Average height of a tree is the average number of links traversed (including the link to reach the root) to execute a successful search operation. Storage utilisation is the ratio between the space occupied and space allotted. Construction cost is measured in nodes read and written to build the tree.

Trees of medium order (15, 20, 25, 30) were examined for their performance over an operating range extending to 100000 keys. Each tree was studied over 20 instances of randomly generated trees.

4.2 Average Height: Figures 5.1 to 5.4 give the average access cost curves for these trees. Also shown is the access cost curve of an 'ideal tree'. An ideal tree gives the best possible average height for a multiway tree. In here, the leaves may differ in height by one and only the unsaturated nodes are the farthest nodes.

The average height of a B-tree grows in sharp steps at the points where the root splits. After a root split the height stays constant for a large number of insertions (till the next split). CB-trees exhibit a more gradual increase in height with increase in tree size. They are like B-trees (in fact, they 'are' B-trees) till the point where the root of a B-tree would have been split. After that they grow differently for some time and again acquire the shape of B-trees. When the root splits in a B-tree, a CB-tree leaf acquires a chain becoming a compound node. Height of the CB-tree grows gradually as more and more nodes become compound nodes. When the CB-tree root sets saturated as a compound node it splits. This is the point where its height overshoots that of a B-tree (This overshoot is too small to observe in trees of order 20 and 25). But quickly after that it loses all its compound nodes and continues to grow like B-tree till one of the leaf nodes become a compound node.

Areas under the B-tree and CB-tree access cost curves were computed for comparison. Table 1 indicates the same along with the standard deviation recorded for each of these areas. If each of them is assumed to be normally distributed about their means, the random variable

\[ T = \frac{\bar{X} - \bar{Y}}{s^2/n_1 + s^2/n_2} \]

where \( s^2_i = \sum (x_i - \bar{X})^2 \) and

\( s^2_i = \sum (y_i - \bar{Y})^2 \)

is approximately distributed as a Student-t with I degrees of freedom where I is calculated as [Green 78]

\[ I = \frac{\sigma_i^2/n_1}{\sigma_i^2/n_1 + \sigma_i^2/n_2} \]

The computed significance level for

Proceedings of the Tenth International Conference on Very Large Data Bases.

Singapore, August, 1984
the hypothesis: Area under the CB-tree access cost curve is less than Area under B-tree access cost curve came out to negligibly small.

4.3 Storage Utilisation: No marked differences were observed among the number of pages that each tree occupies. Table 2 indicates the average storage utilisation for trees of order 25 at various points of their growth. Also shown are the standard deviations for these samples and the confidence level for the hypothesis:

\[
0.95 \times \text{Mean}(\text{B-tree}) \leq \text{Mean}(\text{CB-tree}) \\
\leq 1.05 \times \text{Mean}(\text{B-tree})
\]

These are computed using the t-statistic described earlier.

4.4 Construction Cost: This parameter does not particularly appear to be in favour of any of the two trees. Table 3 shows a sample of the page accesses (reads and writes) incurred per insertion for constructing these trees. Even though CB-trees required less page accesses at the maximum tree size simulated, it is more likely that the oscillatory behaviour indicated in the earlier readings to continue.

5. CONCLUSIONS: A data-structure (Chained B-trees) for index organisation has been proposed. It is obtained by modifying the B-tree definition to allow leaf nodes at different distances (measured in page fetches) away from the root. Simulation studies were conducted to examine its performance as compared to B-trees. The following conclusions were reached:

1) CB-trees exhibit a more gradual growth in their average height with increase in number of keys. Compared to a B-tree this growth curve is nearer to the best possible by a multiway tree.

2) CB-trees provide the same amount of space utilisation as B-trees.

3) It is no more expensive to construct a CB-tree than a B-tree.

6. REFERENCES:


[GREEN 78] : Green, J.R., Marderison, D., 'Statistical Treatment of Experimental Data', Elsevier/North-Holland Inc., First Revised Reprint


[LOMET 81] : Lomet, D.R. 'Digital B-trees', Proc. 7 Intnl. Conf. on Very Large Data Bases


Fig. 1 Chaining Instead of Splitting

(a)

(b)

(c)

(d)

Figure 2
function SEARCH(K:KEY-type ; X : ^ to A NODE ):boolean
{ SEARCH FOR K IN THE TREE POINTED BY X. SEARCH IS TRUE IF KEYS IS FOUND OR FALSE OTHERWISE }
begin
  if X=nil then SEARCH := false
  else with X^ do
    begin
      case K of
      K in {K[1],K[2],...K[KEYSHERE]} : SEARCH := true ;
      K < K[1] : SEARCH := SEARCH(K,[P[0]]) ;
      K > K[KEYSHERE] : SEARCH := SEARCH(K,[P[KEYSHERE]])
      end end end

FIG.3 Function SEARCH

Procedure INSERT ;
{ TO INSERT A KEY K IN THE SUB-TREE ROOTED AT X^ }
begin
  if X=nil then PUSH THE KEY UP
  else with X^ do
    begin
      if TWIN then
      begin
        FIND THE APPROPRIATE SUB-TREE in THE
        COMPOUND NODE THROUGH A BINARY SEARCH;
        INSERT KEY in THE SUB-TREE;
        if SUB-TREE SPLITS
        then
          if (THERE IS SPACE HERE) and (not THERE IS SPACE ABOVE)
          then ABSORB THE UPCOMING KEY
          else
            begin
              SPLIT THE COMPOUND NODE
              ABSORB THE UPCOMING KEY
              PUSH A KEY UP
            end
        else if (THERE IS SPACE ABOVE) then begin
          SPLIT THE COMPOUND NODE
          PUSH A KEY UP
        end
      end
    end
end

Fig.4 Procedure to Insert a key in a CB-tree
FIG. 5.3 ACCESS COST CURVES

FIG. 5.4 ACCESS COST CURVES
<table>
<thead>
<tr>
<th>ORDER</th>
<th>MEAN</th>
<th>S.D</th>
<th>MEAN</th>
<th>S.D</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>367559.9200</td>
<td>1009.8646</td>
<td>336640.3100</td>
<td>2086.7847</td>
</tr>
<tr>
<td>20</td>
<td>364900.9800</td>
<td>1422.8034</td>
<td>312586.8800</td>
<td>1139.0156</td>
</tr>
<tr>
<td>25</td>
<td>345847.6600</td>
<td>105.2140</td>
<td>301769.3300</td>
<td>311.9264</td>
</tr>
<tr>
<td>30</td>
<td>281530.5200</td>
<td>1515.3457</td>
<td>276414.7400</td>
<td>189.1412</td>
</tr>
</tbody>
</table>

**TABLE 1. AREAS UNDER THE ACCESS COST CURVE**

<table>
<thead>
<tr>
<th>KEYS</th>
<th>MEAN</th>
<th>STANDARD DEV.</th>
<th>CONF.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B-TREE</td>
<td>CB-TREE</td>
<td></td>
</tr>
<tr>
<td>1037</td>
<td>0.683582</td>
<td>0.682783</td>
<td>0.042821</td>
</tr>
<tr>
<td>2500</td>
<td>0.689363</td>
<td>0.707390</td>
<td>0.021834</td>
</tr>
<tr>
<td>4957</td>
<td>0.691083</td>
<td>0.695623</td>
<td>0.012508</td>
</tr>
<tr>
<td>10841</td>
<td>0.694090</td>
<td>0.689499</td>
<td>0.009957</td>
</tr>
<tr>
<td>21497</td>
<td>0.696491</td>
<td>0.691716</td>
<td>0.006805</td>
</tr>
<tr>
<td>28827</td>
<td>0.688875</td>
<td>0.690629</td>
<td>0.006490</td>
</tr>
<tr>
<td>42628</td>
<td>0.693079</td>
<td>0.691429</td>
<td>0.005045</td>
</tr>
<tr>
<td>63036</td>
<td>0.685374</td>
<td>0.690657</td>
<td>0.004829</td>
</tr>
<tr>
<td>76655</td>
<td>0.690161</td>
<td>0.694073</td>
<td>0.002767</td>
</tr>
<tr>
<td>84530</td>
<td>0.692911</td>
<td>0.695727</td>
<td>0.003168</td>
</tr>
<tr>
<td>93215</td>
<td>0.695055</td>
<td>0.697747</td>
<td>0.003496</td>
</tr>
<tr>
<td>102793</td>
<td>0.693278</td>
<td>0.699631</td>
<td>0.003372</td>
</tr>
</tbody>
</table>

**TABLE 2. STORAGE UTILIZATION**

**ORDER OF THE TREE = 25**

CONF. = 1.0 - SIGNIFICANCE LEVEL OF THE HYPOTHESIS:

0.95*MEAN(B-TREE) <= MEAN(CB-TREE) <= 1.05*MEAN(B-TREE)

<table>
<thead>
<tr>
<th>KEYS</th>
<th>B-TREE</th>
<th>N-TREE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MEAN</td>
<td>ST.DEV.</td>
</tr>
<tr>
<td>1037</td>
<td>3.00376</td>
<td>0.00357</td>
</tr>
<tr>
<td>2500</td>
<td>3.30908</td>
<td>0.03165</td>
</tr>
<tr>
<td>4957</td>
<td>3.68015</td>
<td>0.01623</td>
</tr>
<tr>
<td>10841</td>
<td>3.88472</td>
<td>0.00729</td>
</tr>
<tr>
<td>21497</td>
<td>3.97033</td>
<td>0.00386</td>
</tr>
<tr>
<td>28827</td>
<td>3.99311</td>
<td>0.00285</td>
</tr>
<tr>
<td>42628</td>
<td>4.01374</td>
<td>0.00194</td>
</tr>
<tr>
<td>63036</td>
<td>4.07762</td>
<td>0.01610</td>
</tr>
<tr>
<td>76655</td>
<td>4.25146</td>
<td>0.01290</td>
</tr>
<tr>
<td>84530</td>
<td>4.32637</td>
<td>0.01166</td>
</tr>
<tr>
<td>93215</td>
<td>4.39433</td>
<td>0.01061</td>
</tr>
<tr>
<td>102793</td>
<td>4.45627</td>
<td>0.00974</td>
</tr>
</tbody>
</table>

**TABLE 3. PAGES READ:WRITES/INSERTION**

**ORDER OF THE TREE = 25**

Proceedings of the Tenth International Conference on Very Large Data Bases.

Singapore, August, 1984