VISVESWARAYA TECHNOLOGICAL UNIVERSITY

BELGAUM

SRI JAYACHAMARAJENDRA COLLEGE OF ENGINEERING, MYSORE-570006

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Report on

IMPLEMENTATION OF B^+-TREE

Guidance of

N.R. PRASHANTH
Professor
Department of CS&E, SJCE, Mysore.

Done By:

VIKRAM T.V.
5th Semester,
Computer Science and Engineering,
S.J.C.E, Mysore

Contents

1 Introduction

A B^+-tree (Bplus Tree) is a type of tree which represents sorted data in a way that allows for efficient insertion, retrieval and removal of records, each of which is identified by a key. It is a dynamic, multilevel index, with maximum and minimum bounds on the number of keys in each index segment (usually called a "block" or "node"). The primary value of a B^+-tree is in storing data for efficient retrieval in a block-oriented storage context in particular, filesystems as B^+-trees have very high fanout (typically on the order of 100 or more), which reduces the number of I/O operations required to find an element in the tree.

Each nonleaf node in the tree has between $\lceil n/2 \rceil$ and n children, where n is fixed for a particular tree. Although there is a overheads in insertion, deletion operations and space, the performance benefits of B^+-tree allows it to be used in most of the applications.

2 Structure of B^+-Tree

A typical B^+-tree node contains n pointers P_1, P_2, ..., P_n and n - 1 search-key values K_1, K_2, ..., K_n. The search-key values within a node are in sorted order, that is if i < j, then $K_i < K_j$. The typical node of B^+-tree is as shown in figure 1.

Figure 1: Typical B^+-Tree Node

2.1 Structure of a Leaf Node

For i = 1, 2, . . . , n 1, pointer P_i points to either a file record with search-key value K_i or to a bucket of pointers, each of which points to a file record with search-key value K_i. The last pointer points to the next leaf node in the tree. Figures 2 and 3 taken from [Silberschatz-Korth] illustrates the usage of leaf nodes and the typical tree.

During implementation, there should be a cast of the pointers while use in leaf nodes as those pointers point to records. Figure 3 clearly shows the n^th pointer pointing to the next leaf node and this allows for efficient sequential processing of the file. Each

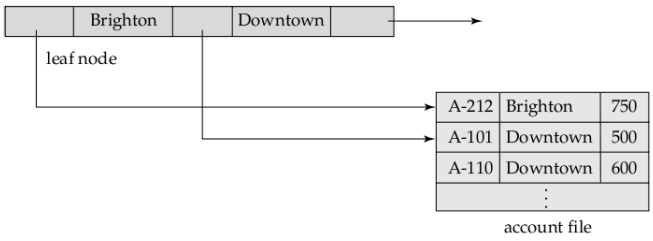

Figure 2: Leaf Node with pointers pointing to records

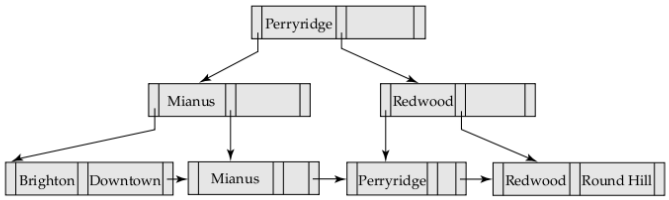

Figure 3: B^+-Tree

leaf node can hold upto n - 1 values and can contain as few as $\lceil (n - 1)/2 \rceil$ values. Also, if L_i and L_j are leaf nodes and i < j, then every search-key value in L_i is less than every search-key value in L_j .

2.2 Structure of non-Leaf nodes

The structure of nonleaf nodes is the same as that for leaf nodes, except that all pointers are pointers to tree nodes. A nonleaf node may hold up to n pointers, and must hold at least $\lceil n/2 \rceil$ pointers. Consider a nonleaf node containing m pointers. For i = 2, 3, . . . , m 1, pointer P_i points to the subtree that contains search-key values less than K_i and greater than or equal to K_i1 . Pointer P_m points to the part of the subtree that contains those key values greater than or equal to K_m1, and pointer P_1 points to the part of the subtree that contains those search-key values less than K_1.

Also, unlike other nonleaf nodes, the root node can hold fewer than $\lceil n/2 \rceil$ pointers, but it should hold at least two pointers, unless the tree consists of only one node. Figure 3 shows a complete B^+-Tree with leaf and nonleaf nodes and $n = 3$. It is a B^+-Tree (Balanced plus tree) for an account file arranged alphabetically with path length 2.

3 Querying B$^+$-Tree

In order to search find all records with a search-key value of V, we first examine the root node, looking for the smallest search-key value greate than V and suppose this value is K_i, we follow the pointer P_i to reach another node. If there is no such value, then k $\geq$ K_m−1, where m is the number of pointers in the node and hence follow P_m to another node. In the reached node, we again search for a smallest search-key value greater than V, and follow the corresponding pointer and eventually reach a leaf-node. In the leaf-node, if search-key value K_i equals V, then pointer P_i directs to desired record or bucket of pointers. If value of V is not found in leaf node, no record with key value V exists.

The algorithm to query a B$^+$-Tree is given in algorithm 1.

Time Complexity: If there are K search-key values in the file, the path is no longer than $\lceil \log_{n/2}(K) \rceil$.

Algorithm 1 Querying a B$^+$-Tree

```
1: procedure find (value V)
2: set C = root node
3: while C is not a leaf node do
4:    Let K_i = smallest search-key value, if any, greater than V
5:    if there is no such value then
6:       Let m = the number of pointers in the node
7:       set C = node pointed to by P_m
8:    else
9:       set C = the node pointed to by P_i
10:   end if
11: end while
12: if there is a key value K_i in C such that K_i = V then
13:   pointer P_i directs us to the desired record or bucket
14: else
15:   no record with key value k exists
16: end if
```

4 Insertion to a B$^+$-Tree

Insertion is a bit more complicated than finding a search-key. We need to split a node that becomes too large as the result of an insertion. Furthermore, when a node is split, we must ensure that balance is preserved. Using the same technique as for lookup, we find the leaf node in which the search-key value would appear. If the search-key value already appears in the leaf node, we add the new record to the file and, if necessary, add to the bucket a pointer to the record. If the search-key value does not appear, we insert the value in the leaf node, and position it such that the search keys are still in order. We then insert the new record in the file and, if necessary, create a new bucket with the appropriate pointer.

Splitting a node: We take the n search-key values (the n - 1 values in the node plus the value being inserted), and put the first $n/2$ in the existing node and the remaining values in a new node. The new node must also be inserted to the tree such that the new node has the same parent as that of the existing node otherwise the parent node also needs to be split to make room for the new node. The split of root makes the tree deeper by one level.

Consider the insertion of a key-value, pointer pair (V, P) to the B$^+$ index. The general idea is to identify the leaf-node l that holds the pair, and if a split results, insert the new node into the parent of node l. If this insertion too results in a split, proceed recursively up the tree, until no new split occurs or new root is created. Algorithms 2 and 3 give the pseudocodes for *insert*, *insert_in_leaf* and *insert_in_parent* respectively. In the pseudocode, L, N, P and T denote pointers to nodes, with L being used to denote a leaf-node. L.K_i and L.P_i denote the ith value and the ith pointer in node L, respectively. 'T' is a temporary that holds the contents of the node to be split. In the case of leaf nodes, the pointer to an entry actually precedes the key value, so the leaf node actually stores P before V . For internal nodes, P is stored just after V.

The above alogrithms were directly implemented with the necessary routines.

5 Sample Run

A sample run of the B$^+$ tree with size 5 and 10 elements resulted in the tree as shown in figure 4. The sentinel value 999999 is used to generate random numbers for insertion. Another run with size 5 and 10 elements is shown in figure 5. Clearly, as the size decreases, the height of the tree increases and thus the search time becoming more.

Algorithm 2 Insertion into B^+-Tree - main routine

procedure insert (*value* K, *pointer* P)
find the leaf node L that should contain key value K
if L has less than n - 1 key values **then**
    insert_in_leaf (L, K, P)
else
    /* L has n - 1 key values already, split it */
    Create node L'
    Copy L.P_1, ..., L.K_{n-1} to a block of memory T that can hold n (pointer, key-value) pairs
    insert_in_leaf (T, K, P)
    Set L'.P_n = L.P_n; Set L'.P_n = L'
    Erase L.P_1 through L.K_{n-1} from L
    c = $\lceil n/2 \rceil$ /* c holds the ceil value of $n/2$ */
    Copy T.P_1 through T.K_c from T into L starting at L.P_1
    Copy T.P_{c+1} through T.K_n from T into L' starting at L'.P_1
    Let K' be the smallest key-value in L'
    insert_in_parent (L, K', L')
end if

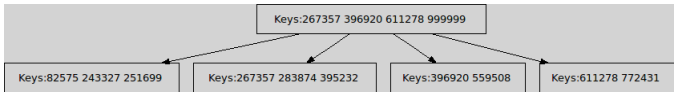

Figure 4: B^+ Tree with size 5

References

[Silberschatz-Korth] Abraham Silberschatz, Korth H.F., Sudarshan S., *Database System Concepts*, Fifth Edition, McGRAW-HILL, 2006.

Algorithm 3 Pseudocodes for insertion in leaf and parent

procedure insert_in_leaf (*node* L, *value* K, *pointer* P)
if K is less than L.K_1 **then**
    insert P, K into L just before L.P_1
else
    Let K_i be the least value in L that is less than K
    insert P, K into L just after T.K_i
end if

procedure insert_in_parent (*node* N, *value* K', *pointer* N')
if N is the root of the tree **then**
    create new node R containing N, K', N' /* N and N' are pointers */
    make R the root of the tree
    return
end if
Let P = parent (N)
if P has less than n pointers **then**
    insert (K', N') in P just after N
else
    /* Split P */
    Copy P to a block of memory T that can hold P and (K', N')
    Insert (K', N') into T just after N
    Erase all entries from P; Create node P'
    c = $\lceil n/2 \rceil$ /* c holds the ceil value of $n/2$ */
    Copy T.P_1 ... T.P_c into P
    Let K" = T.K_c
    Copy T.P_{c+1} ... T.P_{n+1} into P'
    insert_in_parent (P, K", P')
end if

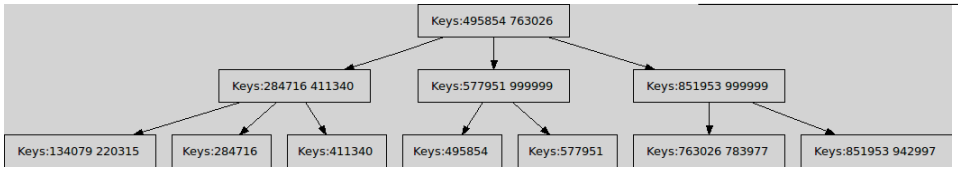

Figure 5: B^+ Tree with size 3