B-Trees

Vaidehi Ghime
Dept. of Comp. Science&Engg.
IIIT, Nagpur
vaidehighime@gmail.com

Anushri Laddha
Dept. of Comp. Science&Engg.
IIIT, Nagpur
anushreeladdha07@gmail.com

Aayesha Bassy
Dept. of Comp. Science&Engg
IIIT, Nagpur
aayes1998@gmail.com

Abstract— **B-trees are by far the most important access path structure in database and file systems. This report discusses about B-trees, which are balanced search trees, it's properties, the different operations on the B-tree data structure, the time complexities of the operations performed and the essential importance of B-trees as a whole. Many database systems use B-trees, or variants of B-trees, to store information. We will also look into how data is accessed from the disks, and how B-trees make the process more efficient.**

I. INTRODUCTION

A B-tree is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree is a generalization of a binary search tree in that a node can have more than two children. In most of the other self-balancing search trees (like AVL and Red Black Trees), it is assumed that everything is in main memory. To understand use of B-Trees, we must think of huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses.

Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is height of the tree. B-tree is a fat tree. Height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red Black Tree,etc.

Computer systems take advantage of various technologies that provide memory capacity. The primary memory (or main memory) of a computer system normally consists of silicon memory chips. This technology is typically more than an order of magnitude more expensive per bit stored than magnetic storage technology, such as tapes or disks. Most computer systems also have secondary storage based on magnetic disks; the amount of such secondary storage often exceeds the amount of primary memory by at least two orders of magnitude. Although disks are cheaper and have higher capacity than main memory, they are much, much slower because they have moving mechanical parts. In order to amortize the time spent waiting for mechanical movements, disk access not just one item but several at a time. Information is divided into a number of equal-sized pages of bits that appear consecutively within tracks, and each disk read or write is of one or more entire pages. We measure the number of disk accesses in terms of the number of pages of information that need to be read from or written to the disk.

In a typical B-tree application, the amount of data handled is so large that all the data do not fit into main memory at once. The B-tree algorithms copy selected pages from disk into main memory as needed and write back onto disk the pages that have changed. B-tree algorithms keep only a constant number of pages in main memory at any time; thus, the size of main memory does not limit the size of B-trees that can be handled.

II. RELATED WORKS

A. *Origin of a B-tree*

Rudolf Bayer and Ed McCreight invented the B-tree while working at Boeing Research Labs in 1971 (Bayer & McCreight 1972).

Ed McCreight answered a question on B-tree's name in 2013:

"Bayer and I were in a lunchtime where we get to think [of] a name. And ... B is, you know ... We were working for Boeing at the time, we couldn't use the name without talking to lawyers. So, there is a B. [The B-tree] has to do with balance, another B. Bayer was the senior author, who [was] several years older than I am and had many more publications than I did. So there is another B. And so, at the lunch table we never did resolve whether there was one of those that made more sense than the rest. What really lives to say is: the more you think about what the B in B-trees means, the better you understand B-trees."

B. *Varients*

1) In the B+ tree, copies of the keys are stored in the internal nodes; the keys and records are stored in leaves.

2) The B* tree balances more neighboring internal nodes to keep the internal nodes more densely packed.

3) B-trees can be turned into order statistic trees to allow rapid searches for the Nth record in key order, or counting the number of records between any two records, and various other related operations.

III. BASIC CONCEPTS OF B-TREE

A B-tree T is a rooted tree (whose root is T.root) having the following properties:

- Every node x has the following attributes:
  - x.n: number of keys stored in the node x.
  - The x.n keys are stored in ascending order.
  - x.leaf: a boolean value that is TRUE if x is a leaf and FALSE is x is an internal node.

- Each node x also contains x.n + 1 pointers to its children.

- The keys separate the range of keys stored in each sub-tree.

- All leaves have the same depth, which is the trees height h.

- Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer t $\geq$ 2 called the minimum degree of the B-tree:

  - Every node other than the root must have at least t-1 keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

  - Every node may contain at most 2t - 1 keys. Therefore, an internal node may have at most 2t children. We say that a node is full if it contains exactly 2t - 1 keys.

- The number of disk accesses required for most operations is proportional to the height of the B-tree which is O(lg n).

IV. BASIC OPERATIONS ON A B-TREE

We model disk operations in our pseudo-code as follows:

1) x = a pointer to some object
2) DISK-READ(x)
3) operations that access and/or modify the attributes of x
4) DISK-WRITE(x) //omitted if no attributes of x were changed
5) other operations that access but do not modify attributes of x

Let x be a pointer to an object. If the object is currently in the computers main memory, then we can refer to the attributes of the object as usual: x.key, for example. If the object referred to by x resides on disk, however, then we must perform the operation DISK-READ(x) to read object x into main memory before we can refer to its attributes. Similarly, the operation DISK-WRITE(x) is used to save any changes that have been made to the attributes of object x.

Since in most systems the running time of a B-tree algorithm depends primarily on the number of DISK-READ and DISK-WRITE operations it performs, we typically want each of these operations to read or write as much information as possible. Thus, a B-tree node is usually as large as a whole disk page, and this size limits the number of children a B-tree node can have.Hence, height is reduced as the number of disk accesses are reduced.

A. Creating an empty B-tree

To build a B-tree T, we first use B-TREE-CREATE to create an empty root node and then call B-TREE-INSERT to add new keys. Both of these procedures use an auxiliary procedure ALLOCATE-NODE, which allocates one disk page to be used as a new node in O(1) time. We can assume that a node created by ALLOCATE-NODE requires no DISK-READ, since there is as yet no useful information stored on the disk for that node.The new node's leaf attribute is set to TRUE and the number of keys is set to 0. A DISK-WRITE is performed the save the attributes of x into the disk. x is made the root node.

B. Insertion

All insertions start at a leaf node. To insert a new element, search the tree to find the leaf node where the new element should be added. Insert the new element into that node with the following steps:

1) If the node contains fewer than the maximum allowed number of elements, then there is room for the new element. Insert the new element in the node, keeping the node's elements ordered.
2) Otherwise the node is full, evenly split it into two nodes so:

   - A single median is chosen from among the leaf's elements and the new element.

   - Values less than the median are put in the new left node and values greater than the median are put in the new right node, with the median acting as a separation value.

- The separation value is inserted in the node's parent, which may cause it to be split, and so on. If the node has no parent (i.e., the node was the root), create a new root above this node (increasing the height of the tree).

V. PSEUDOCODES AND TIME COMPLEXITIES

A. Creating an empty B-Tree

The pseudoecode is as follows:

```
B-TREE-CREATE(T)
1. x = ALLOCATE-NODE()
2. x.leaf = TRUE
3. x.n = 0
4. DISK-WRITE(x)
5. T.root = x
```

Time Complexity:
B-TREE-CREATE requires O(1) = disk operations and O(1) CPU time. O(1) disk operations for line number 4 and overall O(1) CPU time as each line in the code takes O(1) time.

B. Inserting In a B-Tree

The pseudocode for insertion is as follows:
B-TREE-INSERT(T,k)

```
1. r= root[T]
2. if n[r] = 2t - 1
3.    then s =ALLOCATE-NODE()
4.    root[T]= s
5.    leaf[s] =FALSE
6.    n[s] = 0
7.    c1[s] =r
8.    B-TREE-SPLIT-CHILD(s,1,r)
9.    B-TREE-INSERT-NONFULL(s,k)
10. else B-TREE-INSERT-NONFULL(r,k)
```

C. Helping Functions

1) **Split child**

```
B-TREE-SPLIT-CHILD(x,i,y)
1. z =ALLOCATE-NODE()
2. leaf[z] =leaf[y]
3. n[z] = t - 1
4. for j=1 to t - 1
5.    do $key^j[z] = key^{j+t}[y]$
6. if not leaf [y]
7.    then for j =1 to t
8.       do $c_j[z] = c_{j+t}[y]$
9. n[y] =t - 1
10. for j =n[x] + 1 downto i + 1
11.   do $c_{j+1}[x] = c_j[x]$
12. $c_{i+1}[x] = z$
13. for j =n[x] downto i
14.    do $key_{j+1}[x] = key_j[x]$
15. $key_i[x] = key_t[y]$
16. n[x]= n[x] + 1
17. DISK-WRITE(y)
18. DISK-WRITE(z)
19. DISK-WRITE(x) Time Complexity:
```

1) Lines 1 to 9 create node z and give it the largest t¡1 keys and corresponding t children of y.

2) Line 10 adjusts the key count for y. Finally, lines 11 to 17 insert z as a child of x, move the median key from y up to x in order to separate y from z, and adjust xs key count.

3) Lines 18 to 20 write out all modified disk pages. The CPU time used by B-TREE-SPLIT-CHILD is $\theta(t)$, due to the loops on lines 5 to 6 and 8 to 9. (The other loops run for O(t) iterations.) The procedure performs O(t) disk operations.

2) Insert Non Full

```
B-TREE-INSERT-NONFULL(x,k)
1. i =n[x]
2. if leaf[x]
3.  then while i= 1 and k ¡ keyi[x]
4.       do $key_{i+1}[x] = key_i[x]$
5.          i =i - 1
6 .      $key_{i+1}[x] = k$
7.       n[x] =n[x] + 1
8.       DISK-WRITE(x)
9 else while i =1 and k ¡ keyi[x]
10.          do i =i - 1
11.      i =i + 1
12.       DISK-READ($c_i[x]$)
13.       if n[$c_i[x]$] = 2t - 1
14.          then B-TREE-SPLIT-CHILD(x,i,ci[x])
15.             if k ¿ $key_i[x]$
16.                then i =i + 1
17   B-TREE-INSERT-NONFULL($c_i[x], k$)
```

Time Complexity:

1) Lines 3 to 8 handle the case in which x is a leaf node by inserting key k into x. If x is not a leaf node, then we must insert k into the appropriate leaf node in the subtree rooted at internal node x.

2) In this case, lines 9 to 11 determine the child of x to which the recursion descends. Line 13 detects whether the recursion would descend to a full child, in which case line 14 uses B-TREE-SPLIT-CHILD to split that child into two non full children.

3) Lines 15 to 16 determine which of the two children is now the correct one to descend to. (Note that there is no need for a $DISK - READ(x, c_i)$ after line 16 increments i , since the recursion will descend

in this case to a child that was just created by B-TREE-SPLIT-CHILD.)

4) The net effect of lines 13 to 16 is thus to guarantee that the procedure never recurses to a full node.

5) Line 17 then recurses to insert k into the appropriate subtree.

6) For a B-tree of height h, B-TREE-INSERT performs O(h) disk accesses, since only O(1) DISK-READ and DISK-WRITE operations occur between calls to B-TREE-INSERT-NONFULL. The total CPU time used is $O(th) = O(tlog_t n)$.

7) Since B-TREE-INSERT-NONFULL is tail-recursive, we can alternatively implement it as a while loop, thereby demonstrating that the number of pages that need to be in main memory at any time is O(1).

VI. **ADVANTAGES OF B-TREES**

Tha advantages of using B-trees for databases and files are as follows :

- Keeps keys in sorted order for sequential traversing.
- Uses a hierarchical index to minimize the number of disk reads.
- Uses partially full blocks to speed insertions and deletions.
- Keeps the index balanced with a recursive algorithm.
- In addition, a B-tree minimizes waste by making sure the interior nodes are at least half full.
- A B-tree can handle an arbitrary number of insertions and deletions.

VII. CONCLUSIONS

B-trees are balanced search trees specifically designed to be stored on disks. Because disks operate much more slowly than random-access memory, we measure the performance of B-trees not only by how much computing time the dynamic-set operations consume but also by how many disk accesses they perform. For each B-tree operation, the number of disk accesses increases with the height of the B-tree, but B-tree operations keep the height low.

TABLE I
TIME COMPLEXITIES

Creating an Empty B-Tree	O(1)
Insertion in a B-tree	O(log n)
Splitting A child	O(t)
Insert Non-Full	O(th) = O(t1ogn)

REFERENCES

[1] Bayer, Rudolf; McCreight, E. (July 1970), Organization and Maintenance of Large Ordered Indices, Mathematical and Information Sciences Report No. 20, Boeing Scientific Research Laboratories.
[2] MIT OpenCourseWare lectures.
[3] Introduction to Algorithms, third edition, by Thomas H. Cormen, Chapter 18.
[4] B Tree Visualisation (https://ysangkok.github.io/js-clrs-btree/btree.html)
[5] Merging and Splitting (http://techieme.in/b-trees-split-and-merge/)
[6] Insertion B-trees(https://www.geeksforgeeks.org/b-tree-set-1-insert-2/)