Srishti Mishra
01FB15ECS310

[1]

Analysis of B-trees to store large number of indexes

Srishti Mishra
USN: 01FB15ECS310

Abstract

B-trees are efficient data structures designed to access, read and write data to disks and secondary storage devices. A B-tree is a balanced multi-way search tree whose properties, such as height and number of children, are bounded by a minimum degree. Since main memory is limited, data stored in blocks of information called pages is transferred between the backing store and memory as and when required. Using this data structure, the operations search, insert and delete on large datastores take logarithmic amortized time and are proportional to the height of the tree. Since the B-tree can have a large number of keys (defined by its minimum degree), the B-tree keeps its height low, hence improving its performance. Analysing the number of reads and writes while performing these operations with 1000s of keys, the average number of disk accesses per second stays low when using this data structure. Moreover, storage utilization is at least 50% at any time and does not have a problem in contracting and expanding through freeing or allocating pages.

I. Introduction

B-trees are balanced multi-way search trees with certain restrictions on the number of keys and children they can have, in order to allow searches, insertions, and deletions in logarithmic amortized time. B-trees generally have a large number of keys and a short height *h*, since most B-tree operations require O(*h*) accesses to the disk.

The data structure is designed to access disks and secondary storage memory efficiently. Since the main memory in a system is limited, a majority of data is stored in secondary storage or on backing store and the required pages of data must be moved into main memory when needed. A page is a block of data which can be moved in and out of main memory depending on when the data on the page needs to be accessed or modified. Since disk accesses are very slow, B-trees minimize the number of accesses which needs to be made by reading/writing to large chunks of data at a time (typically, the size of the page). This makes them a popular choice when implementing databases and file system storage.

II. Properties of B-trees

B-trees have the following properties:

1. Every node *x* contains
   a. Number of keys currently stored - *x.n*
   b. The values or keys - from *x.arr[0]* to *x.arr[n-1]*
   c. A boolean flag to indicate leaf nodes - *x.leaf*

2. The tree has a minimum degree - *t* .
   The number of keys that can be stored in a node is bounded by the minimum degree *t* as follows:
   a. Every node other than root must have at least *t - 1* keys. This indicates it must have at least *t* children.
   b. Every node may contain at most *2t - 1* keys. If a node contains *2t - 1* keys, it is a *full* node. Therefore, an internal node may have at most *2t* children.

3. Every internal node *x* contains *x.n+1* pointers to its children - $x.c_1, x.c_2, \ldots, x.c_{n+1}$

4. The keys $x.key_i$ indicate the ranges of the keys stored in each subtree. Keys less than $x.key_i$ are stored in the child node $x.c_i$ and keys greater than $x.key_i$ are stored in the child node $x.c_{i+1}$.

5. All leaves have the same depth - height of tree *h*

6. Each node is usually a disk block or a page.

A. Height of B-trees

Since, the cost of searching, inserting and deleting is proportional to the height of the tree, having a large number of children in the balanced tree ensures that the height remains low and hence the cost of these operations remain low too. The bounds imposed by the minimum degree *t* for *n* keys govern the height *h* of the tree

$$h \leq \log_t \frac{n+1}{2}$$

where $t \geq 2$ *and* $n \geq 1$.

Since *t* can be large, there are fewer disk accesses, as compared to other balanced search trees, while performing operations on the B-tree.

[1] 12/11/17
Advanced Algorithms - Assignment 3 on B-trees

Srishti Mishra
01FB15ECS310

B. *Number of nodes in B-trees*

Let N_{min} and N_{max} be the minimal and maximal number of nodes in a B-tree. Then

$$N_{min} = 1 + 2\,(t+l)^0 + (t+1)^1 + \ldots + (t+l)^{h-2})$$
$$= i + \tfrac{2}{t}(\,(t+1)^{h-1} - 1)$$

where minimum degree is t, and height $h \geq 2$.

$$N_{max} = \sum_{i=0}^{h-1} (2t-1)^i = \tfrac{1}{2t}\,(\,(2t+1)^h - 1)$$

where height $h \geq 1$.

III. Performance of B-trees on disk accesses

. Since main memory is limited, required data is moved between main memory and the backing store or secondary storage in the form of pages (or blocks of data read off secondary storage). We use indexes to reference these pages and minimize the number of reads and writes by ensuring that a page holds a considerable amount of information - usually a node of the B-tree - and by keeping track of the pages in the form of the special data structure described above, the B-tree.

B-trees tackle the program of organising and maintaining indexes for dynamically changing files in the system. Indexes consists of a key x and its associated information a stored in fixed size structures. The information a points to records or a collection of records in a file.

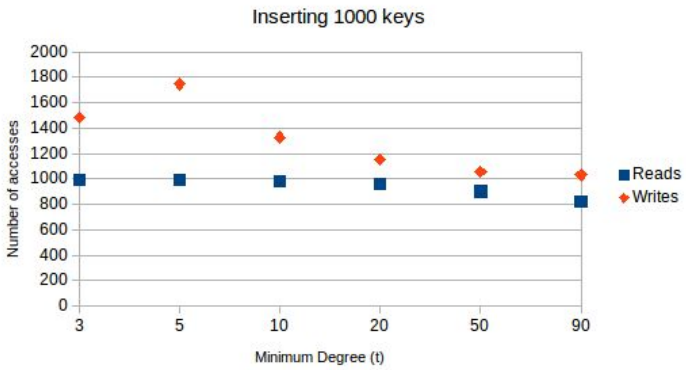
Since the number of indexes a B-tree keeps track of is large, only some of the indexes can be kept in main memory at a time. The rest needs to be transferred from the disk when required.. This transfer operation takes a long time, but can transfer a lot of data at once.

B-trees take advantage of this by identifying each node as a page. If this is used along with batches of inserts, retreivals and deletions, it can be optimized further to reduce disk accesses (since in many keys can be added in a single access to a page, or while that page is still in memory).

The number of disk-reads and disk-writes are counted while executing the operations in the array simulation of the B-tree.

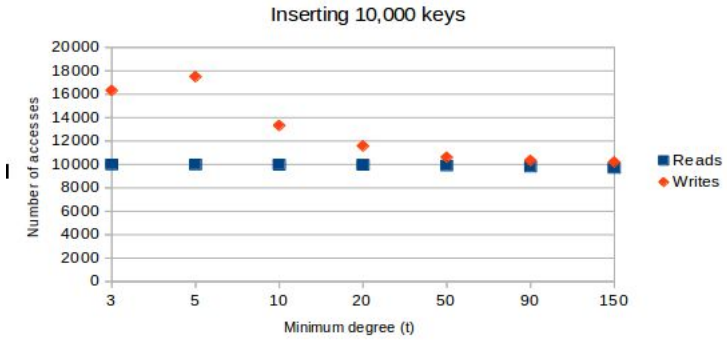
A. *Insertion of 1000 keys with varying t values*

For different values of *t*, namely, 3, 5, 10, 20, 50 and 90, the number of disk-accesses, both to read and write decreases as the *t* value increases. This indicates that, as the height decreases (since *t* value increases - more children in each node), the number of disk access reduce while inserting values.



B. *Insertion of 10000 keys with varying t values*

For different values of *t*, namely, 3, 5, 10, 20, 50, 90, and 150, the number of accesses, of both reads and writes reduce drastically after the t value starts increasing. This indicates that these operations are proportional to the height of the tree and as the height decreases (since t-value increases and there is more space for keys as well as more children nodes at every level), the performance of the B-tree improves.



IV. Insertions and Retrievals

A B-tree is designed such that the keys in the node specify a range of values for its children. The keys of in the node are arranged sequentially in ascending order $key_1 \leq key_2 \ldots \leq key_i$ and the values of the keys in the children lie in between the key values in the parent. While traversing the tree, the next node can be chosen from a minimum of *t* children and a maximum of *2t* children, making a $(x.n + 1)$ - way branching decision, where n is the number of keys. If the key is not found and the node is not a leaf node, a disk access takes place to read the corresponding child node (since each node is a page).

Srishti Mishra
01FB15ECS310

```
searchTree(Tree *tree, int ind, int key){
    int i = 0;

    //Initialize a node to search in
    bNode *tempNode = tree->arr[ind];
    printf("\nsearching node %d containing elements :", ind);

    while( (i < tempNode->n) && (key > tempNode->keys[i]) )
        i++;

    if((i <= tempNode->n) && (key == tempNode->keys[i])){
        printf("\n Found at node %d key %d ", ind, i);
    }
    else if(tempNode->leaf == 1){
        printf("Not found!");
    }
    else{
        bNode *newChild = malloc(sizeof(bNode));
        read_file(tree, newChild, key);
        searchTree(tree, newChild->pos, key);
    }

}
```

Due to searching through all keys in a node and then branching, this operation takes O(*th*) time.

While inserting a key into the B-tree, a number of checks need to be performed. This is because the upper bound on the number of keys a node can hold is *2t-1* and the lower bound is *t-1*. The code checks for a full node and splits the node, whenever necessary, before inserting the new element. The node is split around its median value, into two nodes having *t-1* children each and the median moves up to the parent of the originally full node. However, if the parent is also full, another split occurs and in the worst case, this could happen all the way up to the root of the tree. If the root of the tree is split, it results in an increase in the height of the tree.

There is a helper function which then inserts the key into the non-full node which was selected, after splitting, if necessary.

Hence, insertion into the B-tree takes O(*th)* time.

V. Deletion from B-Trees

Deleting from a B-Tree is more complicated than insertions and searches since the deletion of the key can cause the tree to become unbalanced or can cause a node to have less than t-1 keys. The nodes and their keys need to be rearranged appropriately, so that all the properties of the B-Tree are adhered to. The code uses the one-pass algorithm, so that a backwards traversal need not occur and is therefore more efficient. Deletion can reduce the height of the tree.

There are several cases in deletion :

Case 1:

If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.

Key k is in node x and x is a leaf, simply delete k from x.

Case 2 : If key k is in node x and x is an internal node :

a) If the child y that precedes k in node x has at least t keys (more than the minimum), then find the predecessor key k' in the subtree rooted at y. Recursively delete k' and replace k with k' in x.

b) Symmetrically, if the child z that follows k in node x has at least t keys, find the successor k' and delete and replace as before.
Note: Finding k' and deleting it can be performed in a single downward pass

c) Otherwise, if both y and z have only t−1 (minimum number) keys, merge k and all of z into y, so that both k and the pointer to z are removed from x. y now contains 2t − 1 keys, and subsequently k is deleted.

Case 3 :

If key k is not present in an internal node x, determine the root of the appropriate subtree that must contain k. If the root has only t − 1 keys, execute either of the following two cases to ensure that we descend to a node containing at least t keys. Finally, recurse to the appropriate child of x.

a) If the root has only t−1 keys but has a sibling with t keys, give the root an extra key by moving a key from x to the root, moving a key from the roots immediate left or right sibling up into x, and moving the appropriate child from the sibling to x.

b) If the root and all of its siblings have t−1 keys, merge the root with one sibling. This involves moving a key down from x into the new merged node to become the median key for that node.

VI. Conclusions

The B-Tree is an efficient data structure which takes advantage of the memory-management techniques of the operating system in order to provide faster operations even when there is a lot of data to be stored. The challenge lies in efficient implementations, and several improvements over ordinary B-Trees have already been found. One of them are B+ Trees, where only keys are stored in the nodes, and a layer of linked leaves are added at the bottom, saving on storage and allowing faster access of all nodes of the tree. Choosing an

optimal Minimum Degree *t* is important, so that disk accesses can be kept at a minimum while being storage efficient. The minimum degree may also be bounded by the size of the page, which can vary from system to system.

B-Trees are a huge improvement from binary search trees, since their heights are extremely low, and most of their operations are proportional to their height. Also, at any point in time, at least 50% of the storage in the B-tree is used (dependent on the implementation) and is generally higher on an average, making it a little more storage efficient too.

VII. References

[1] R.Bayer, E.M. McCreight. *Organisation and Maintenance of Large Ordered Indices. Available :* http://www.dtic.mil/get-tr-doc/pdf?AD=AD0712079

[2] J. K. Cormen, Leiserson, Rivest, Stein, "Ch 18. B-trees" in *Introduction to Algorithms, 3rd* ed.

[3] R.Bayer. *Binary B-Trees for Virtual Memory. Available :* https://dl.acm.org/citation.cfm?id=1734731

[4] http://scanftree.com/Data_Structure/deletion-in-b-tree