B-Trees

Advanced Algorithms : Assignment 3 Report

Sruthi V
USN : 01FB15ECS311
Section F Semester 5
Department of Computer Science, PES University

***Abstract*—This document is a report on the implementation of the data structure B-trees using arrays and using files. B-trees are the most commonly used data structures in databases and file systems. They're self balancing and accommodate storage of large number of records. Implementation of B-trees can be either single pass or double pass, though single pass is used with respect to our method in this report.**

I. Introduction

B-trees have a tree like orientation with a unique specification known as the degree. Since a node in the B-tree can have more than one key, the degree of the B-tree sets a limit on this variability. If the degree of a B-tree is t, a B-tree node can have a minimum of ***t-1*** keys and a maximum of ***(2*t)-1*** keys. Since a node can have multiple keys, it is inferred that it can have multiple children. Thus the limit on the children are a minimum of ***t*** and a maximum of ***2*t***, where t is the degree if the B-tree.

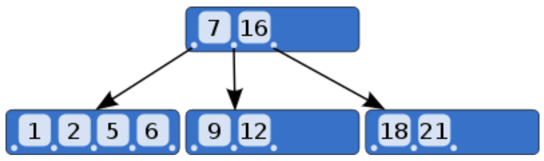


Fig. 1. Example of B-tree

The height of the B-tree grows logarithmically with the number of nodes it contains,ensuring that searching in a B-tree is extremely fast. This paves way for the usage of B-trees in databases.[2] In a B-tree application, the amount of data in it is so large that all the data cannot fit into main memory at once. Therefore, B-tree keeps only certain pages in the memory at a time. A disk read is issued to access a page that is not in memory. This also ensures that the size of the B-tree is not limited.

The reasons[1] for using B-trees in database systems are that they keep keys in sorted manner, they allow sequential traversal of keys, they keep the tree balanced, they provide partially full blocks for fast insertions and they minimize the number of disk reads. An important feature is that a B-tree does not waste space by making sure that internal nodes are atleast half full.

In this report we look at two implementations of a B-tree. The first is the array implementation, where keys in the B-tree are stored in nodes, which are actually arrays of keys. The second implementation tries to simulate the real world application of B-trees by storing them in files. Nodes of the B-tree are written and read from the file. The various aspects that are of key notice : the degree of the B-tree, the number of disk reads and disk writes to the B-tree and the time taken to construct and insert into it.

II. Preliminaries

The information to be stored in the B-tree is a record, which is monitored by a structure containing its fields.

A. Array Implementation

The original B-tree is implemented as a structure. It contains an array of nodes, or a node pool, from which a node is allocated. It also contains additional information regarding degree and maximum free nodes that can be allocated. The ListOfNodes member :

```
struct node *ListOfNodes;
```

A node is used to contain all it's records. It has an array of keys, an array of children and additional information. The array of children contain indexes, i.e, the child's index in the list of nodes in the B-tree. The start value in the node indicates it's index in the List of nodes. Leaf variable denotes if it is a leaf or not.

```
struct node{
        struct record * keys;
        int n; //current keys
        int leaf;
        int *children;
        int start;
        int c; //children count
}
```

B. File Implementation

The B-tree structure has a file pointer to store the file to which the nodes are written. It also has a member *next-pos* to store the next position to which a write can be issued to the file. The node structure has the additional variable *pos* that stores it's position in the file.

C. Disk Read and Write

- **Array implementation** : Disk read and write is simulated in the array by changing values in it.
  Read :
  ```
  struct node x=Btree.ListOfNodes[s.children[0]]
  //x is the 0'th child of node s
  ```
  Write
  ```
  Btree.ListOfNodes[st.start]=s;
  //Write s back to it's position in the array
  ```
- **File implementation** : Disk read and write is done in the array by changing values using *fseek*. *fseek* to a position will help us go to that particular line in the file.
  ```
  fseek(Btree.fp, pos * sizeof(struct node), 0);
  //fseek to position
  //pos * sizeof(struct node) as
  //the node is stored here.
  //Filepointer fp now pointes to this location
  ```
  To this position, fread or fwrite is issued.

III. Creation

The initial creation of the B-tree involves allocating it's first node, which will go on to be it's root. Allocating a node is

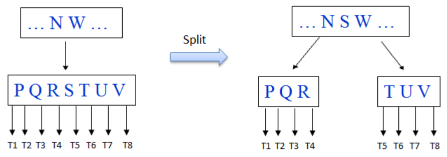
- **Array implementation** : extraction one from the node pool and increment the counter that keeps track of next free node.
- **File implementation** : Writing a node to the empty file at the first position.

We also set limits on the maximum keys a node can have based on it's degree.

IV. Insertion

Insertion here is implemented as a single pass procedure. Instead of waiting to see if a node is split on insertion, each full node on the way from the root to the position of insertion is split. The split is taken care by an additional function.

Fig. 2. Splitting a node



Splitting a node takes a non full node and an index of split. The index child o this node is split into two and the indexes are adjusted accordingly. Splitting a node issues 3 disk writes as all 3 nodes, i.e the 2 split ones and their parent, have to be written back. Thus disk operations are *O(1)* The CPU time taken by this function is *θ(t)* where t is the degree of the B-tree.

The number of disk accesses for insertion is *O(h)* = = $O(\log_t n)$,where h is the height of the tree. The below figure shows how the disk reads and disk writes vary with change in degree.

It can be seen how smaller degree B-trees have more disk reads as the number of insertions increase. This is because each node itself can accommodate fewer keys, and thus frequent disk accesses are required to create new nodes.

The CPU time[2] required for insertion is *O(th)* where t is the degree and h is the height. In insertion, if the root is full, a new root is created and the height of the tree is increased. The final insertion is done by an auxiliary function Insert Non Full that inserts into a node that is guaranteed to be non full by previously splitting the full nodes. It recurses down to the appropriate subtree, and goes through the node to find the appropriate position for the key.

Fig. 3. Disk Reads for Insertion

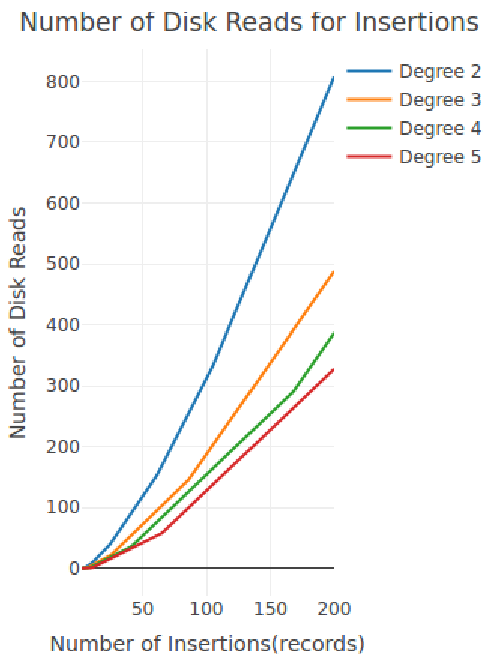


Fig. 4. Disk Writes for Insertion

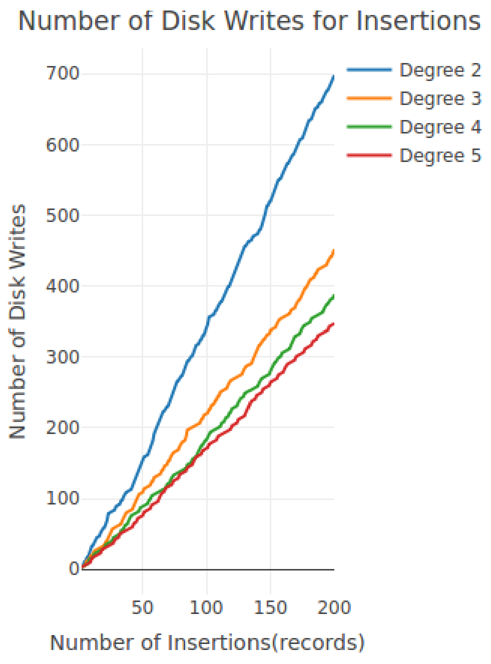
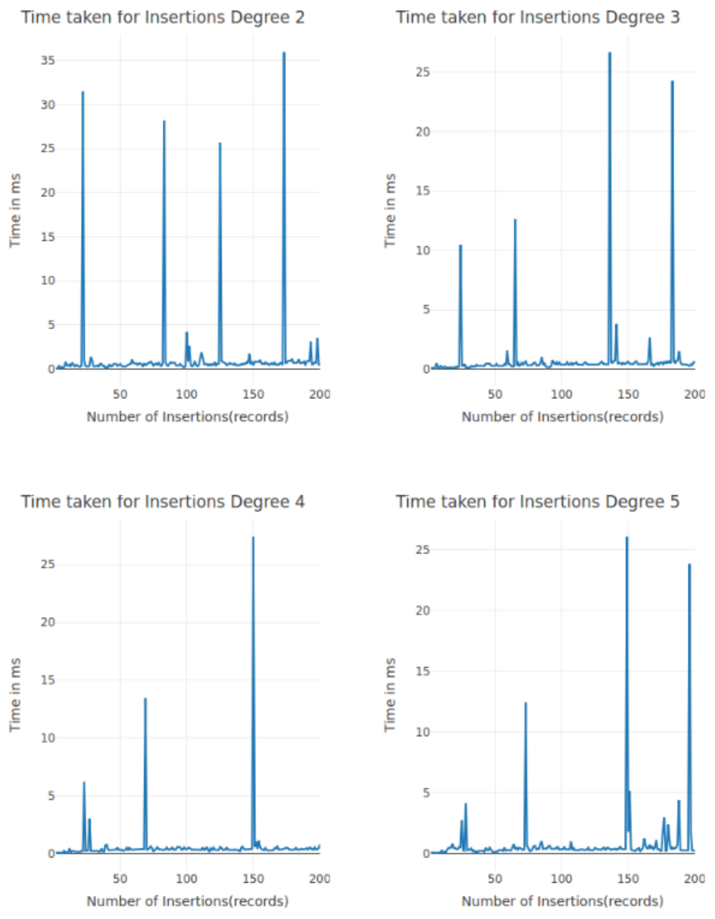


Fig. 5. Insertion time



V. Search

Searching in a B-tree takes a total CPU time of O(th) = O(t $\log_t n$) where t is the degree of the B-tree and h is it's height. The Search process runs through the keys of a node sequentially and finds the appropriate child to traverse to if the key is not present in this node. Searching through the file implementation of the B-tree consumed more time than that of the array implementation as disk read takes more time in the former case. For example, for a degree of 3, search for a key in the array implementation took 0.0023 ms and the same in file implementation to 0.065 ms. The disk reads done by Search is of order *O(h)*.

VI. Deletion

Deletion is implemented by having a logical field known as the valid bit. A key is deleted by setting this bit to 0. Since in deletion the key is first searched for, it os of the same order as search, i.e *O(h)*.

VII. Problems Faced

The issues faced, though currently fixed, while implementing this are :

- Insertion of large number of records
- Traversal through the array without the use of pointers
- The method of using *fseek* with respect to the nodes

VIII. Conclusion

Results extracted show that the :

- File implementation consumes more time than array implementation with respect to disk reads and disk writes as they involve shifting of a file pointer.
- The number of disk reads and disk writes in both implementations are the same.
- The implemented version of the algorithms [2]tb show almost the similar orders with respect to their theoretical values.

References

[1] Wikipedia : B trees.
[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms.