Efficient Medical Data Management in Electronic Health Record Systems Using a Modified B-Tree Indexing Algorithm for Scalable Clinical Information Retrieval

Dr. Sofia Marinova[1]*
Prof. Daniel Okafor[2]
[1] University of Vienna, Department of Biomedical Informatics and Clinical Data Engineering, Vienna, Austria
[2] University of Cape Town, School of Health Data Systems and Medical Informatics, Cape Town, South Africa

ABSTRACT

Main memory and secondary memory are different types of memory resources, as both of them have different properties and characteristics. It is often hard to load a large size of datasets into the main memory due to its cost and secondary memory can hold a large amount of data but the access time is comparatively slower than main memory. That's why datasets reside in secondary storage HDDs, Magnetic tapes, etc. To process such size of datasets, a required part of datasets is retrieved from secondary storage and is placed in the internal memory for processing with the help of data-structures like B-trees and B+ trees andother variants of B-trees. For a large dataset in primary memory CPU processing time and number of disk access time are important.In processing of B-tree, number of disk access depends on height of B-tree which is $O(\log_{blocksize/2} n)$, so no change can be done in the number of the disk access. In this paper we would use an efficient algorithm to modify CPU processing time from O(n)to O(log n).

Keywords: *B- Tree, B+ Tree, Data Management, Primary memory, secondary memory.*

I. INTRODUCTION

B-trees[1] and B+ trees[2] are disk based data structures that are stored and accessed from disk. The processing however is done in memory. If a change is done in it is immediately written on the disk. They are used in the implementation of databases like spatial databases. As the datasets are increasing in size it is getting habitual to use disk based data structures instead of placing whole dataset in main memory.

Main memory and secondary memory have different properties like, access time on main memory is faster than secondary memory and main memory is more costly so large datasets can be impractical to load. Data structures that are efficient on main memory may not work as same on secondary memory.

The data flow in disk based data structure is that the data is first fetched from the disk and then put into the memory to process, after processing the disk write operation is used to update the disk. Many data structures can be used to hold the processing in memory like, array, linked list, binary trees, B-trees and B+ trees. B-trees use nodes that are transferred to memory for processing.

In real- world, data is increasing due to exponential increase of structured and unstructured data. It is becoming impractical to process that amount of data as a data structure on the main memory. Efficient data structures like array, linked list, binary tree, B-trees and B+ trees are proposed but all of these have their pros and cons. Main memory is fast but less in comparison to secondary memory. So, secondary memory is used to store the datasets. So with increasing data, better retrieval speeds are also required. For that efficient algorithms are being created which focus to decrease the processing time, disk accesses.
In disk based data structures processing, storing and accessing are the three time complexity defining factors [4]. In this paper, the proposed structure and algorithm for the implementation of B-tree will focus to reduce the CPU processing time.

II. LITERATURE REVIEW

B-trees is mostly used as a standard in disk-based data structure as it is efficient and can have variants to improve the data handling process. Douglas Comer, in “The Ubiquitous B-tree” [1] published that why B-trees are so successful by mentioning its operations insertion, balancing, deletion and splitting. He wrote about various variants of B-trees i.e. B*trees and B+ trees and their properties. Virtual B-tree was an idea discussed in this paper which uses the concept of paging and addressing.Jan Jannik, in “Implementation and deletion in B+ trees” [2] proposed that an algorithm for implementation of B+ trees with the help of C programming libraries. In his paper he wrote that “deletion, due to its greater complexity and perceived lesser importance” and proposed algorithm for better deletion operation in B+ trees.In “Introduction to Algorithms”, by Thomas H. Cormen et al. [3] mentioned that the node size of the B-tree in secondary storage is depends on the page size. The page size is the size of unit data that is transferred from secondary storage to main memory.

III. METHODOLOGY

This paper proposed a modified structure of a node in a B-tree which is as follows:

```
struct node
 {        int value;
       Int nextindex;
          long pointer*c;    //c may be null pointer or disk pointer
}pds[blocksize];
```

Value	Index to next node	Child Pointer

Figure1: Representation of a Node in B- Tree

The structure contains an array of nodes. Each node consists an integer value to store the data. Pointer to the child node and next Index, will hold the index of the next node in the same pds. The field nextIndex is taken as an integer to save space.

3.1 Proposed algorithm for searching an item:

Instead of linear search in original algorithm, proposed algorithm is performed binary search on nodes array as follow:

```
Step1: Fetch the nodes array i.e pds from secondary storage
          Disk_read(pds)
Binary_search_node(pds,item)
Step2: Intialise variables lower_bound(lb),upper_bound (ub), location(loc), middle (mid)
      lb->1        ub->n[pds]        loc->1   mid=(lb+ub)/2
Step3: Compare the value of middle node element with "item" and proceed accordingly
          If (pds[mid].value==item)
                    loc->mid
                    return (loc)
          else
                    If (pds[mid].value>item)
                              ub->mid-1
                    lb->mid+1
Step4:  Iterate till lb==ub
          if (lb==ub)       loc→lb
Step5:  read the child node from secondary memory
          If (leaf(pds))
                    Return -1;
          Else
                    If(pds[loc].value>item)
                              pds=Disk_read(pds[loc-1].c)
                    else      pds=Disk_read(pds[loc].c)
                    call the function recursively
Step6: END
```

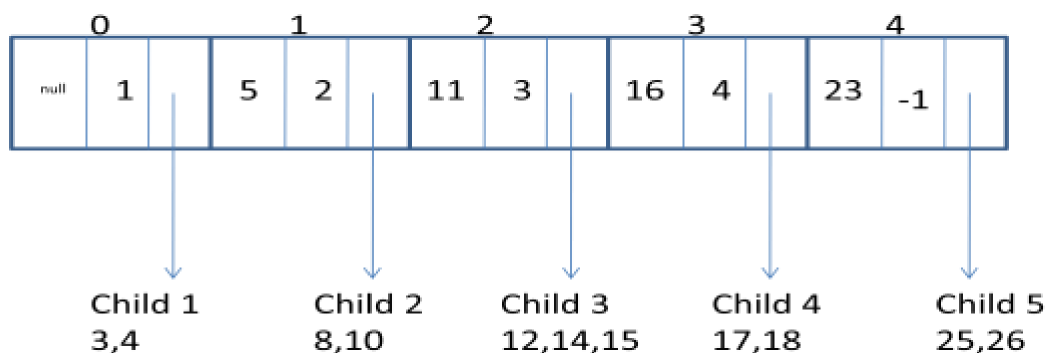
3.2 Proposed Algorithm for Inserting an Item

In this algorithm we are first doing binary search to reach the leaf node where we have to insert the item. If the leaf node is already filled, then we would split the node into 2 and send the median value to the parent. We will repeat

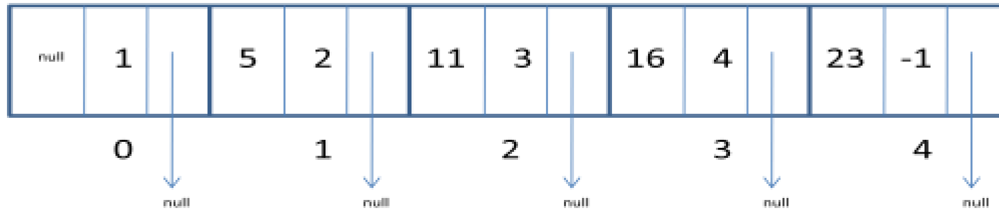
this process until all nodes are completely filled or less than block size. Our algorithm does not require shifting of values as we are storing them with the help of the variable "nextIndex".

INSERT: 3,26,4,25,5,23,8,18,10,17,11,16,12,15,14

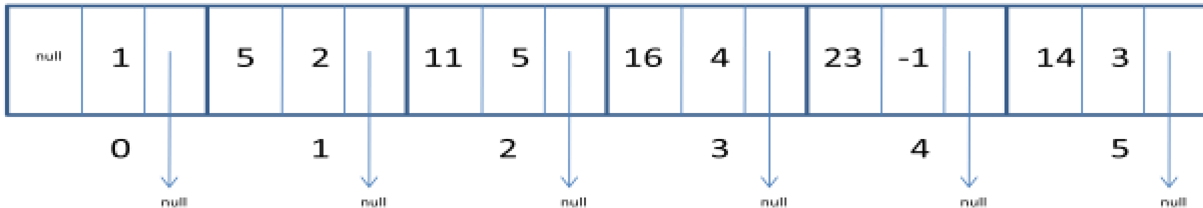
```
Step1: Fetch the nodes array(root) i.e pds from secondary storage
        pds=Disk_read(pds)
Step2: Intialise variables lower_bound(lb), upper_bound(ub) and location(loc)
step3: initialize variable mid
        mid=(lb+ub)/2
Step4: Compare the value of middle node element with "item" and proceed
accordingly
        If (pds[mid].value==item)
                loc->mid
        else
                If (pds[mid].value>item)
                        ub->mid-1
                lb->mid+1
Step5:  Iterate till lb==ub
        if (lb==ub)     loc→lb
Step6: check type of node and insert accordingly
        If leaf(pds)
                If node is full, Split the node
                Else    Insert accordingly
        Else
                Read the child node from secondary memory
                If(pds[loc].value>item)
                        pds=Disk_read(pds[loc -1].c)
                else    pds=Disk_read(pds[loc].c)
                call the function recursively
                B-tree_insert(pds,item)
Step7: END
```



Pds (leaf)



Insert 14
After traversal, loc = 2, n[pds] = 4
Pds[loc].nextIndex = n[pds] +1
Pds[n[pds]+1].nextIndex = loc + 1
Pds[n[pds]+1].value = 14



Disk_write() is done according to the next index.

3.3 Proposed Algorithm for Splitting a Node:
In the proposed algorithm, the child node would be split in to 2 parts. The median value of the node will be sent to the parent node. We pass 3 arguments, node which we want to split, its parent node and location in the parent node where value should be placed.

3.4 Algorithm Analysis
There are 3 ways calculating recursion:

- Substitution Method
- Recursive tree method
- Master Method

Complexity calculation using Master Method:
$T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$
There are following three cases:
1. If $f(n) = \Theta(n^c)$ where $c < Log_b a$ then $T(n) = \Theta(n^{Log_b a})$
2. If $f(n) = \Theta(n^c)$ where $c = Log_b a$ then $T(n) = \Theta(n^c Log\ n)$
3.If $f(n) = \Theta(n^c)$ where $c > Log_b a$ then $T(n) = \Theta(f(n))$

Binary Search – $T(n) = T(n/2) + O(1)$
$a = 1$, $b = 2$, $k = 0$ and $p = 0$
$b^k = 1$. So, $a = b^k$ and $p > -1$ [Case 2.(a)]
$T(n) = \theta(n^{log_b a} \log^{p+1} n)$
$T(n) = \theta(logn)$

Table 1: Time complexity of searching and inserting a key in proposed algorithm and general B-Tree algorithm

Data Structure used	SEARCHING OPERATION	INSERTION OPERATION
B-TREE	$\Theta(nlogn)$	$\Theta(n^2 logn)$
PROPOSED DATA STRUCTURE	$\Theta(logn*logn)$	$\Theta(logn*logn)$

SEARCHING COMPLEXITY: Θ(logn*logn); as the time complexity for the traversal of tree is Θ(logn) and time complexity of doing binary search in a node is also Θ(logn), therefore total time complexity is Θ(logn*logn).
INSERTION COMPLEXITY: Θ(logn*logn); insertion complexity would be multiplied with the searching complexity calculated above. Our insertion part would have a time complexity of 1 because we are inserting at the last position while updating 2 fields of proposed structure i.e NextIndex.

IV. CONCLUSION

This paper provided an efficient algorithm for management of data using B-trees thus giving us reduced time complexity and less number of CPU cycles. This algorithm can be applied to a wide variety of systems such as managing of VM images in cloud, local PC storage, inventory management system.

REFERENCES

1. *Comer D., "The Ubiquitous B-tree", ACM Computer Survey, Vol. 11, No. 2, pp. 121-137, June 1979.*
2. *Jannink J., "Implementing deletion in B+-trees", Proc. ACM SIGMOD Int. Conf. Manag. Data, vol. 24, no. 1, pp.33–38, 1995.*
3. *Thomas H. Cormen et.al "Introduction to Algorithms", Third Edition 2009 pp. 486-488.*
4. *Arora, N., Tamta, V. K., and Kumar, S. 2012. Modified Non-Recursive Algorithm for Reconstructing a Binary Tree. International Journal of Computer Applications. Vol 43. No 10. 25-28*