An Advanced Approach to Tree

Soumadip Biswas (10IT60R12)
Jyotirmay Dewangan (10IT60R04)
School of Information Technology
Indian Institute of Technology, Kharagpur

Abstract:

This paper is basically about proposing a new enhancement of tree in general. This concept can be used in any basic available trees e.g. binary tree, balanced tree, multi-way-tree etc. Basically it is about enhancing the no of relations between to elements, as for conceptual level we have considered on integer domain. The focus is to improve and find an efficient way-out for B+ tree finally to improve data access time.

Introduction:

Whenever talking about information technology, it is always our concern is concentrated on a particular aspect, which is nothing but ‘Information’, more specifically ‘data’. So we are always focused on data retrieval, access to data, and obviously the ease and efficient access of it. Here we will see the different access type and their efficiency, and then we will be proposing our model of data structure.

Problem Definition:

We have a lot of methods and data structure are available for maintain data base. They are efficient for most of the small data warehouse management system, but not for big one, for e.g. in GPS database takes more time to retrieval and process, because of big size of data base. All that existing method can further be improved, so that this type of big data base can be handled easily.

From the first of handling data the main problem is to access data efficiently. Continuing with that legacy we have different options, and now the question is can we have more? The main challenge in this context is to have a good design of new data structure which not only serves the purpose of giving a new data structure but also fits in the available technologies.

Existing Methods

As we can see, from the first we are been provided with different solutions in this prospect, for example the primitive way of storing multiple data is use of ‘Array’. It comes with some very useful solutions as well as problems like fixed size, difficulty in accessing when huge no of data is practiced. The next concept

‘Linked-list’ comes with solution to the memory problems, but still there are accessing problems.
In this concern the concept of tree came into the picture, which has the most impact on the approach to the data. Here we will see verities of them as per the hierarchy.

Binary Search tree:

A binary search tree is a special form of the binary tree in which all nodes satisfies the property that the value at any node n is greater than every node in the left sub-tree and less than every node in the right sub-tree.
Though this is a good structure for representing the data but it also comes with some problems like ‘skewed tree’, here the accessing time becomes O(n).

Height balance binary tree (AVL):

Height balance tree is such a data structure in which have complexity of any operation is O(log n). Balance factor is measured in the term of balance factor , and balance factor is basically difference of height of left sub tree and height of right sub tree. it is not efficient because required much more rotation and have to calculate balance factor for each node in each single operation.

Red Black Tree:

Red black tree is also one of the balance tree structures. In this color information is also stored, which helps to balance the tree structure. Here we don’t need to calculate balance factor of each node. it has following property :

a. root and all leaf node always is black.
b. node color are either red or black.
c. red node never be child of any other red node.
d. black depth property must be preserved at every level. Black depth of node is defined as no of black node in that particular node to leaf node.

B tree

Unlike a binary-tree, each node of a b-tree may have a variable number of keys and children. The keys are stored in non-decreasing order. Each key has an associated child that is the root of a sub tree containing all nodes with keys less than or equal to the key but greater than the preceding key. A node also has an additional rightmost child that is the root for a sub tree containing all keys greater than any keys in the node. A b-tree has a minimum number of allowable children for each node known as the minimization factor. If t is this minimization factor, every node must have at least t - 1 key. Under certain circumstances, the root node is allowed to violate this property by having fewer than t - 1 key. Every node may have at most 2t - 1 key or, equivalently, 2t children.

Since each node tends to have a large branching factor (a large number of children), it is typically necessary to traverse relatively few nodes before locating the desired key. If access to each node requires a disk access, then a b-tree will minimize the number of disk accesses required. The minimization factor is usually chosen so that the total size of each node corresponds to a multiple of the block size of the underlying storage device. This choice simplifies and optimizes disk access. Consequently, a b-tree is an ideal data structure for situations where all data cannot reside in primary storage and accesses to secondary storage are comparatively expensive (or time consuming), h =< log (n+1)/2. The worst case height is O(log n). Since the "branching" of a b-tree can be large compared to many other balanced tree structures, the base of the logarithm tends to be large; therefore, the number of nodes visited during a search tends to be smaller than required by other tree structures. Although this does not affect the asymptotic worst case height, b-trees tend to have smaller heights than other trees with the same asymptotic height.

B+ tree

A B+-tree in certain aspects is a generalization of a binary search tree (BST). The main difference is that nodes of a B+-tree will point to many children nodes rather than being limited to only two. Since our goal is to minimize disk accesses whenever we are trying to locate records, we want to make the height of the multi-way search tree as small as possible. This goal is achieved by having the tree branch in large amounts at each node.

A B+-tree of order m is a tree where each internal node contains up to m branches (children nodes) and thus store up to m-1 search key values -- in a BST, only one key value is needed since there are just two children nodes that an internal node can have. m is also known as the branching factor or the fan out of the tree.

1. The B+-tree stores records (or pointers to actual records) only at the leaf nodes, which are all found at the same level in the tree, so the tree is always height balanced.
2. All internal nodes, except the root, have between Ceiling(m/2) and m children
3. The root is either a leaf or has at least two children.
4. Internal nodes store search key values, and are used only as placeholders to guide the search. The number of search key values in each internal node is one less than the number of its non-empty children, and these keys partition the keys in the children in the fashion of a search tree. The keys are stored in non-decreasing order (i.e. sorted in lexicographical order).
5. Depending on the size of a record as compared to the size of a key, a leaf node in a B+-tree of order m may store more or less than m records. Typically this is based on the size of a disk block, the size of a record pointer, etcetera. The leaf pages must store enough records to remain at least half full.
6. The leaf nodes of a B+-tree are linked together to form a linked list. This is done so that the records can be retrieved sequentially without accessing the B+-tree index. This also supports fast processing of range-search queries as will be described later.

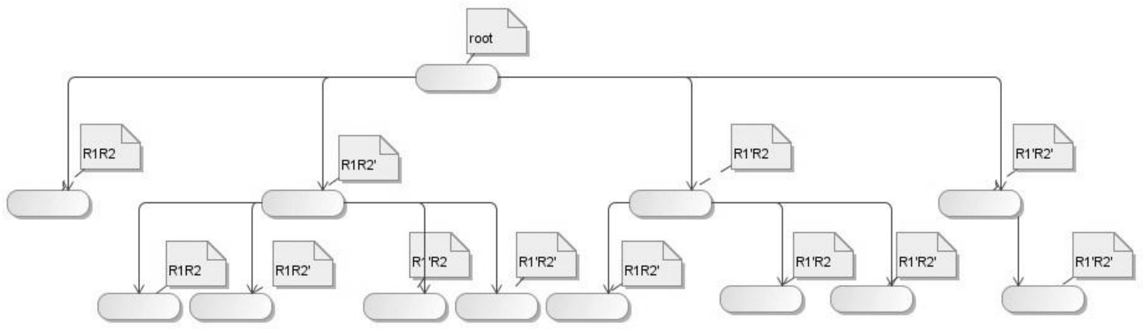
The simplest B+-tree occurs when m = 3: every internal node then has either 2 or 3 children. In practice, the size of a node in the B+-tree is chosen to fill a disk page. A B+-

tree node implementation typically allows between 100 to 200 children, the actual number depends on the memory requirements of the keys and memory addresses. It's been reported that the typical fill-factor of the nodes is 67%, giving an average branching factor or fan-out of 133 (i.e. each internal nodes points to 133 children nodes). In a B+-tree of height 4 that has an average fan out of 133 can be used to index a table of over 300million records (to be exact, that's 1334 = 312,900,700). A B+-tree of height 3 can index a table of 1333 = 2,352,637 records. By accessing a mere five disk pages (the number of levels traversed is one more than the height of the tree), the RDBMS can find one of the 300+ million records. Furthermore, it is common for the top levels of the B+-tree to be cached in main memory. The memory requirements are not that high. Using a typical page size of 8Kbytes, the first two levels of the B+-tree (up to 134 disk pages) can be cached in about 1Mbyte. Now the RDBMS needs to only access three disk pages to find one of the 300+ million records! This clearly illustrates why B+-trees have been so successful in their application to indexing data in databases.

Proposal:

We are proposing a new kind of a tree in this paper. As the concern of having a efficient usage of a tree is nothing but the links. A normal tree provides two links from it thus making the no of forward search node divided into two parts. The main idea is to use some relation to divide the data set. Now we are doing the same thing, but considering that whether we can make some improvement in case of number of relations. In abstract level it can be viewed as any relation R= {(a, b) | a is related to b}; on the basis of this relation we use the division of values, like truth values of the relation i.e. true or false. Here we can use for instance a relation which is anti-symmetric and transitive and this relation for our purpose to serve. Here we think that as we already use the relation '>' for choosing the direction of a new value to be inserted to a tree, and we always tried to increase the no of links from a node to increase efficiency by adding multiple values. Here if we can increase the no of links corresponding to a value, for instance R1 & R2 are two relations that hold on the entire domain of inputs. Then we have 4 options to choose e.g. R1R2, R1R2', R1'R2',R1'R2 . So we are proposing to use this idea in normal trees to have a better access efficiency.

Here’s a figures to illustrate our concept



This diagram shows a demonstration of our idea with teo relationshoips R1 & R2

We do think that this approach will give us an efficient way to handle data set available. As the no of links are 4 (in case of two relations) instead of 2, if the tree can be balanced then the no of forward scanning nodes will be ¼th every time, thus can give us O(log4n) instead of O(log2n).

Plan for Implementation

Our plan is to implement our new concept in the following steps. First we will be identifying some such relations. Then we will try to implement them in replacement of binary tree. If successful then we will extend this concept to B+ tree which in turn will save our data access time, thus query processing time also. For comparison we will be using some same sample data in both the old and new approach.

We are not providing any particular paper as reference for this proposal, as we see this can be applied as a general algorithm.
But after we can implement this algorithm, we will try to apply this to the only referring paper, which is on indexing and mining free trees.

References

Chi, Y.; Yang, Y.; Muntz, R.R.; , "Indexing and mining free trees," *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on* , vol., no., pp. 509- 512, 19-22 Nov. 2003