Performance Comparison of Balanced Search Structures on Indexing of Database Files

Remya G
Assistant Professor
Department of Computer Science and Engineering
NSS College of Engineering, Palakkad, India

***Abstract*— Most of the relational database systems use indexing mechanism for efficient retrieval of database records. The data structure used for building such index has great impact on the performance of the index. Fundamentally the index should be a search structure which supports lookups and insertion in efficient time. The commonly used search structures like binary search tree, AVL tree, Red Black tree, B tree and B+ tree can serve this purpose. The proposed work is to perform a theoretical performance comparison of B+ Tree, AVL and Red Black tree on indexing of database records. Analysis results shows that B+ tree is a better structure for implementing database indexing when compared to other search structures. Results also show that the choice of order of B+ tree is another important factor which affects the performance of the system.**

Keywords-Database index, B+ tree, Search Strucutre.

I. INTRODUCTION

A single-level index [1] is an auxiliary file that makes it more efficient to search for a record in the data file .The index is usually specified on one field on several fields of the file. General form of an index is a file of entries <field data, pointer to record>, which is ordered by field value.The index is called an access path on the record field. Both index file and the data file will occupy storage but the index file requires less number of blocks. Also as the size of the data file increases the size of the index file also increases. When the size of the index file is very large, we need to use disk blocks to store the index file.Thus, when the data is so large that it can't be stored in main memory, the number of disk accesses becomes important. A disk access is very expensive compared to a typical computer instruction as one disk access is worth about 200,000 instructions. Many algorithms and data structures that are efficient for manipulating data in main memory are not suitable for manipulating data in disk memory because they are not designed to minimize the number of disk accesses.

II. SEARCH SRUCTURES

A. AVL Tree

The AVL Tree was defined by Georgy Adelson-Velsky and Evgenii Landis, two Soviet Mathematicians, in 1962 [2]. It is the first defined self-balancing binary search tree data structure. In order to avoid skewness, AVL tree uses balance factor, which is the difference between the height of its right sub tree and the height of its left sub tree. A node is said to be balanced it have a balance factor of either -1, 0, or 1. Nodes with balance factor greater than 1 or less than -1 are considered 'unbalanced', and different rotations are performed so as to make the tree rebalanced.

Inserting a value into an AVL tree often requires a look up for the location and the insertion may result in the loss of balance. In order to rebalance the tree we apply a re-balancing algorithm which rotates branches of the tree to ensure that the balance factors are kept in the range [-1,1]. Both the rebalancing and search take O(log n) time. Deletion is also like an insertion, a deletion from an AVL tree also requires the tree to be re-balanced. This operation also takes O (log n) time.Let N(h) = minimum number of nodes in an AVL tree of height h.We can apply induction method to find the height of an AVL tree. Basis is N(0) = 1, N(1) = 2 and by Induction $N(h) = N(h-1) + N(h-2) + 1$ The solution to this equation can be obtained by Fibonacci analysis and we will get $N(h) \geq \phi^h$ ($\phi \approx 1.62$)

Suppose we have n nodes in an AVL tree of height h.

1) $n \geq N(h)$ (because N(h) was the minimum)

2) $n \geq \phi^h$ hence $\log_\phi n \geq h$ (relatively well balanced tree)

3) $h \leq 1.44 \log_2 n$ (Find takes O (logn))

When the number of nodes in the AVL tree n becomes 2^{30} which is approximately equal to 10^9, the height of the tree lies between 30 and 43.When the AVL tree is stored on a disk, up to 43 disk access are made for a search. This takes approximately up to 3 seconds. As the height of an AVL tree increases, the number of disk accesses required to access a particular record also increases.

B. RedBlack Tree

A red–black tree [3][4] is also a self-balancing binary search tree. Each node of the RedBlack tree has an extra bit, which represents the color of the node. These colors are used to rebalance the tree during insertions and deletions. In addition to the conditions for a binary search tree the redblack tree need to satisfy the following constraints.

1. Each node can be either red or black.
2. All leaves nodes are black.
3. If a node is red, then both its children are black.
4. The number of black nodes from the root to a node is called the node's black depth. Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes.

Even if the balancing of the tree is not perfect, it is good enough to guarantee searching in O(log n) time, where n is the total number of elements in the tree. The insertion and removal operations requires either tree rotation or recoloring, which can be also performed in O(log n) time.

Property of Red-Black-Trees [7]

A red-black tree with n internal nodes has height at most 2log(n + 1).Any node x with height h(x) has $bh(x) \geq h(x)/2$ where bh(x) represents the black height of node x. The sub tree rooted at any node x contains at least $2^{bh(x)}$ - 1 internal nodes. Therefore $N \geq 2^{bh} - 1 \geq 2^{h/2} - 1$.Adding 1 to either side and taking logs: $n + 1 \geq 2^{bh} \geq 2^{h/2}$, $\lg(n + 1) \geq h/2 \Rightarrow h \leq 2 \lg(n + 1)$.The complexity of search on a RedBlack tree is O(h) which is approximately O(log(n))
When the number of nodes in the RedBlack tree, n becomes 2^{30} which is approximately equal to 10^9 ,the height of the tree lies between 30 and 60. When the RedBlack tree is stored on a disk, up to 60 disk access are made for a search. This takes approximately up to 5 seconds. As the height of a RedBlack tree increases, the number of disk accesses required to access a particular record also increases.

C. B+ Tree

A B+ tree [5][6] is balanced n-ary tree with more than two children possible for each node. A B+ tree contain of a root node , internal and leaf nodes.The B+-Tree consists of two types of nodes namely (1) internal nodes and (2) leaf nodes
Internal nodes point to other internal and leaf nodes of the tree. Leaf nodes hold the data pointers which can point to the database records. Each leaf nodes also contain a sibling pointer, which helps in sequential access of records .All the nodes in a B+-Tree must be at least half full except the root node which may contain a minimum of two entries. The algorithms to insert and delete from a B+-Tree will guarantee that each node in the tree will be at least half full. Search for a value in the B+ tree starts from the root and moves downwards to the leaf .Each internal node in a B+-Tree contains a set of keys which are ordered and node pointers. The number of children possible for the internal nodes can be defined as the order P of the tree, and the number of data values possible on the leaf nodes is defined as the order of the leaf node P_{leaf} .

Analysis of B+ Trees

We can find the maximum number of values in a B+ tree of order p(assuming both internal and leaf nodes have same order) and height h:At root we have p– 1, at level 1 we have $p(p – 1)$, at level 2 we have $p^2(p – 1)$ and so on and at level h we have $p^h(p – 1)$. So, the total number of items is $(1 + p + p^2 + p^3 + \ldots + p^h) (p – 1) = [(p^{h+1} – 1)/ (p – 1)] (p – 1) = p^{h+1} – 1$
$N \approx p^h$, $h = O(\log_p n)$

A stricter analysis will give $h <= 1 + \log_{\lceil p/2 \rceil} ((n + 1) / 2)$

III. COMPARISON OF B+ TREE WITH AVL/REDBLACK TREE

The height h of a B-tree of order m, with a total of n keys, satisfies the inequality:
$h <= 1 + \log_{\lceil m/2 \rceil} ((n + 1) / 2)$

If m = 200 and n = 15,000,000 then $h \approx 3$.Thus, in the worst case finding a key in such a B-tree requires 2 disk accesses (considering the root is always in memory). For an AVL tree, the average number of comparisons with n keys is log n + 0.25 where n is large. If n = 15,000,000 the average number of comparisons is 23.Thus, in the average case, finding a key in such an AVL tree requires 23 disk accesses. To calculate the order p of a B+ -tree, suppose that the search key field is V = 10 bytes long, the block size is B = 1024 bytes, a record pointer is Pr = 8 bytes, and a block pointer is P = 4 bytes. An internal node of the B+-tree can have up to p tree pointers and p - 1 search field values and these must fit into a single block[1]. Hence, we have:

$$(p * P) + ((p - 1) * V) \leqq B$$
$$(p * 4) + ((p - 1) * 10) \leqq 1024$$
$$(14 * p) \leqq 1024$$

We can choose p to be the largest value satisfying the above inequality, which gives p = 73. Hence, the order Pleaf for the leaf nodes can be calculated as follows:

$$(Pleaf * (Pr + V)) + P \leqq B$$
$$(Pleaf * (8 + 10)) + 4 \leqq 1024$$
$$(18 * Pleaf) \leqq 1028$$

It follows that each leaf node can hold up to Pleaf = 57 key value/data pointer combinations, assuming that the data pointers are record pointers. Suppose that we construct a B+-tree on the field .To calculate the approximate number of entries of the B+ -tree, we assume that each node is 60 percent full. On the average, each internal node will have 73 * 0.60 or approximately 43 pointers, and hence 42 values. Each leaf node, on the average, will hold 0.60 * Pleaf = 0.60 * 57 or approximately 34 data record pointers. A B+-tree will have the following average number of entries at each level:

	Nodes	Entries	Pointers
Root	**1**	**42**	**43**
Level 1	**43**	**1806**	**1849**
Level 2	**1849**	**77658**	**79507**
Level 3	**79507**	**27,03248 (record pointers)**	

Suppose that the search key field is V = 10 bytes long, the block size is B = 1024 bytes, a record pointer is Pr = 8 bytes, and a block pointer is P = 4 bytes Each avl/redblack tree node can have at most 2 tree pointers, 1 data pointers, and 1 search key field values. These must fit into a single disk block if each AVL/RedBlack tree node corresponds to a disk block. Suppose that the search field is a non ordering key field, and we construct a AVL on this field. Each node, on the average, will have 2 tree pointers and, hence, 1 search key field values. We can start at the root and see how many values and pointers can exist, on the average, at each subsequent level:

	Nodes	Entries	Pointers
Root	**1**	**1**	**3**
Level 1	**2**	**2**	**6**
Level 2	**4**	**4**	**12**
Level 3	**8**	**8**	**24**

Hence, for the given block size, pointer size, and search key field size, a three-level avl/redblack tree holds 8+4+2+1=15 entries. A three-level B+ tree holds up to 27,03248 record pointers, on average. When compared to the 15 record pointers for the corresponding avl/redblack tree the B+ definitely seems to a better option to create a multi level index. The fundamental difference is that the B+ tree benefits from the disk access property that within a single access of the disk, a fixed number of bytes(in the example it is 1024 bytes) will be loaded into the memory. Later we can apply a binary search on the node to locate the actual record value.

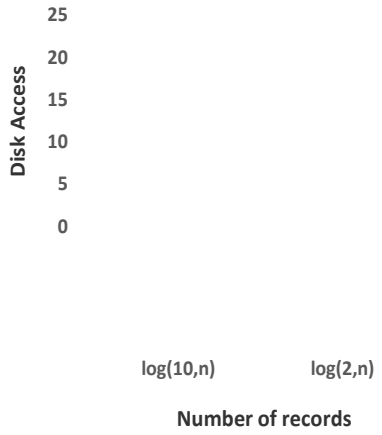

Fig 1. B+ vs AVL - Height Comparision

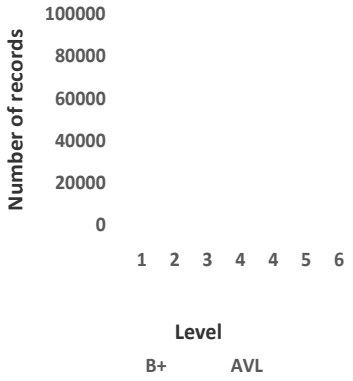

Fig 2. B+ vs AVL – Efficiency

Nodes of Avl/RedBlack tree may be placed at different blocks in the disk that may really make their performance worse in storing indexes A binary search is to be performed within the node after disk access in B+ tree but still it incomparable because a disk access takes about the same time as executing 200,000 instructions The choice of order P and pleaf of B+ tree also benefits from the property of disk access. The choice Of order p in B+ tree is a deciding factor which influences the number of disk accesses and there by the performance of the index. The worst-case search time can be computed as (time to fetch a node + time to search node) * height ,ie (a + b*p + c * log2p) * h where a, b and c are constants .The search time decreases with the increase in order of the B+ tree, but beyond a particular value for order, the search time will increase.

IV CONCLUSION

A database index is a search structure which is used to efficiently retrieve records from the database files based on

some indexing field. Most of the dictionary based search structures can serve this purpose. This work aims to compare the efficiency of these structures when used for database indexing. Analysis shows that B+ trees are well suited when considered as a disk based structure for database indexing compared to AVL and RedBlack trees. It is widely used as an index structure in most of the database management systems because of its versatility.

REFERENCES

[1]. Navathe, Ramez Elmasri, Shamkant B. (2010). Fundamentals of database systems (6th ed.). Pearson Education. pp. 652–660.

[2]. Georgy Adelson-Velsky, G.; Evgenii Landis (1962). "An algorithm for the organization of information". Proceedings of the USSR Academy of Sciences (in Russian) 146: 263–266. English translation by Myron J. Ricci in Soviet Math. Doklady, 3:1259–1263, 1962

[3]. R. Bayer. Symmetric binary b-trees: Data structures and maintenance algorithms. Acta Informat., 1:290–306, 1972.

[4]. L. J. Guibas and R Sedgewick. A dichromatic framework for balanced trees. IEEE Symposium on Foundations of Computer Science, pages 8–21, 1978.

[5]. Rudolf Bayer and Edward M. McCreight, Organization and Maintenance of Large Ordered Indices. Acta Informatica 1: 173–189 (1972)

[6]. Douglas Comer: "The Ubiquitous B-Tree", ACM Computing Surveys 11(2): 121–137 (1979)

[7]. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Red–Black Trees". Introduction to Algorithms (second ed.). MIT Press. pp. 273–301.

[8]. Ramakrishnan Raghu, Gehrke Johannes - Database Management Systems, McGraw-Hill Higher Education (2000), 2nd edition (en) page 267

[9]. Donald Knuth. The Art of Computer Programming, volume 3. Addison-Wesley Publishing Co., Philippines, 1973.