International Journal of Advances in Engineering and Management (IJAEM)
Volume 2, Issue 11, pp: 237-240 www.ijaem.net ISSN: 2395-5252

Query Optimization using B+ Trees

Paras Goyal, K C Tripathi, M L Sharma
Department of Information Technology, Maharaja Agrasen Institute of Technology, Delhi- India

Date of Submission: 05-12-2020 Date of Acceptance: 20-12-2020

ABSTRACT: This is a research-based project and the basic point motivating this project is learning and implementing Data Structures and Algorithms [1] that reduces time and space complexity [2]. In this project, we reduce the time taken to search a given record by using a B/B+ tree [3] rather than indexing and traditional sequential access. It is concluded that disk-access times are much slower than main memory access times. Typical seek times and rotational delays are of the order of 5 to 6 milliseconds and typical data transfer rates are of the range of 5 to 10 million bytes per second and therefore, main memory access times are likely to be at least 4 or 5 orders of magnitude faster than disk access on any given system. Therefore, the objective is to minimize the number of disk accesses and thus, this project is concerned with techniques for achieving that objective i.e. techniques for arranging the data on a disk so that any required piece of data, say some specific record, can be located in as few I/O's as possible.

KEYWORDS: B+ Trees - DBMS - Search Optimizations - File Systems

I. INTRODUCTION

B/B+ trees are extensively used in Database Management Systems [4] because search operation is much faster in them compared to indexing and traditional sequential access. Moreover, in DBMS, B+ tree is used more as compared to B-Tree. This is primarily because unlike B-trees, B+ trees have a very high fan out, which reduces the number of I/O operations required to find an element in the tree. This makes the insertion, deletion, and search using B+ trees very efficient [5]. However, the indexing of columns to be searched is also efficient but the downside of it is that when searching is to be done on large collections of data records, it becomes quite expensive, because each entry in B/B+ tree requires us to start from the root and go down to the appropriate leaf page. This operation takes only O(log n) time. Hence we would also like to implement the efficient alternative, B+ tree.

B-Tree is a self-balancing search tree [6]. In most of the other self-balancing search trees, it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to the main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, etc.) require O (h) disk accesses where h is the height of the tree. B-tree is a fat tree. Height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, the B-Tree node size is kept equal to the disk block size. Since it is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree [7], Red-Black Tree [8], etc.

II. RELATED WORK

The binary search tree is a well-known data structure. When the data volume is so large that the tree does not fit in main memory, a disk-based search tree is necessary. The most commonly used disk-based search trees are the B-tree and its variations. Originally invented by Bayer and McCreight [9], the B-tree may be regarded as an extension of the balanced binary tree, since a B-tree is always balanced (i.e., all leaf nodes are on the same level). It is the goal of all algorithms to consume as less time as possible in the computer and manage a tradeoff of time and space. B+ trees are used in a number of ways to manage this tradeoff by utilizing required space [10]. Efficient construction of indexes is very important in bulk-loading a database or adding a new index to an existing database since both of them should handle an enormous volume of data. Proposed algorithm for batch-constructing the B+-tree, the most widely used index structure in database systems. B+ Tree, a widely-used storage structure in database management systems, and propose its utilization for supporting the logging in databases [12]. Developing B+ trees with good performance on NVM, using Optane DCs, to study and analyze the influence factors of designing B+ trees on NVM.

III. METHODOLOGY

3.1 B+ Tree

Most file systems employ search-trees to index the stored data, and the B+ tree is a special search-tree with the following features: – It stores records: r = (k, d); k = key, d = data. The key is unique. – Data is stored only in leaves, inner-nodes are only index-nodes. – In an index-node there are x keys, and also x + 1 pointers, each pointing to the corresponding subtree. – The B+ tree has one main parameter, namely its order. If the order of a B+ is d, then for each index node there is a minimum of d keys, and a maximum of 2d keys, so there are a minimum of d + 1 pointers, and maximum of 2d + 1 pointers in the node. From the above, if a B+ tree stores n nodes, its height must not be greater than logd (n) + 1. The total cost of insertion and deletion is O (logd (n)) [13]. This data structure is used by some database systems like PostgreSQL and MySQL, and file systems like ReiserFS, XFS, JF2 and NTFS.

The leaves (the bottom-most index blocks) of the B+ tree are often linked to one another in a linked list; this makes range queries or an (ordered) iteration through the blocks simpler and more efficient (though the aforementioned upper bound can be achieved even without this addition). This does not substantially increase space consumption or maintenance on the tree. This illustrates one of the significant advantages of a B+ tree over a B-tree; in a B-tree, since not all keys are present in the leaves, such an ordered linked list cannot be constructed. A B+ tree is thus particularly useful as a database system index, where the data typically resides on disk, as it allows the B+ tree to actually provide an efficient structure for housing the data itself.

If a storage system has a block size of B bytes, and the keys to be stored have a size of k, arguably the most efficient B+ tree is one where. Although theoretically, the one-off is unnecessary, in practice there is often a little extra space taken up by the index blocks (for example, the linked list references in the leaf blocks). Having an index block which is slightly larger than the storage system's actual block represents a significant performance decrease; therefore erring on the side of caution is preferable. If nodes of the B+ tree are organized as arrays of elements, then it may take a considerable time to insert or delete an element as half of the array will need to be shifted on average. To overcome this problem, elements inside a node can be organized in a binary tree or a B+ tree instead of an array. B+ trees can also be used for data stored in RAM. In this case, a reasonable choice for block size would be the size of the processor's cache line.

3.2 Implementation Blocks

3.2.1 Insertion

1) Initialize x as root.
2) While x is not a leaf, do the following
a) Find the child of x that is going to be traversed next. Let the child be y.
b) If y is not full, change x to point toy.
c) If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as the first part of y. The else second part of y. When we split y, we move a key from y to its parent x.
3) The loop in step 2 stops when x is a leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

3.2.2 Searching

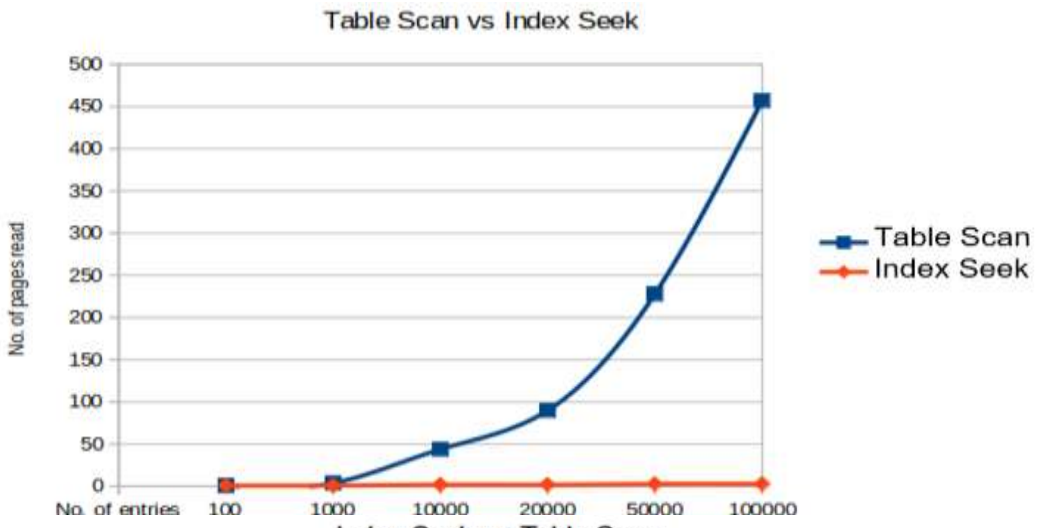
1) Initialize x as root.
2) While x is not a leaf, do the following:
a.) find the key value which equal to value to search, if found return the index.
b.) otherwise go to the node which may contain value to be searched i.e. if the value to be searched is valid, then go to the node which has key values x< val <y.
c.) Recursively call the node.
3) The recursion in step 2 stops when x is leaf and we have not found value to be searched. In that case, we return the key not found.

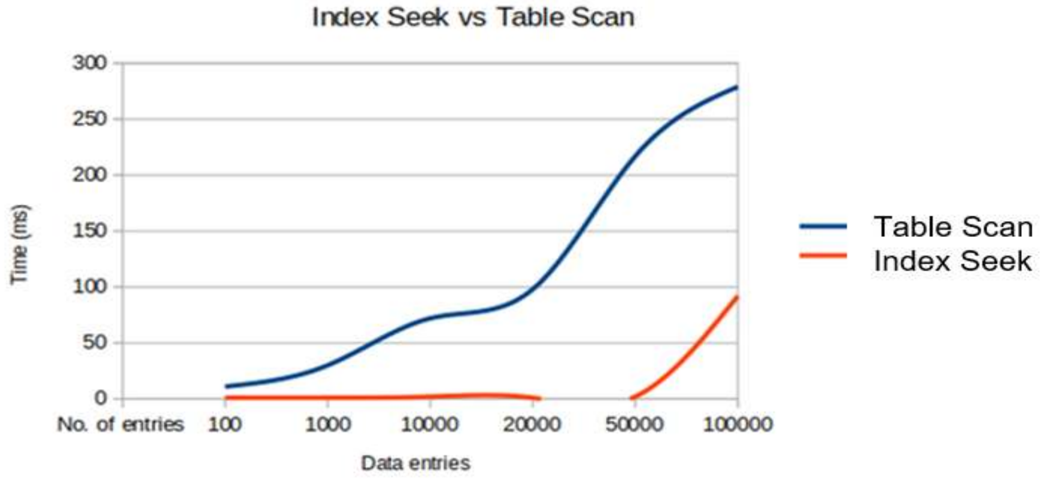
3.2.3 Code Structure

- 2 classes are made for the BTree implementation: One is the client which is used to run all the functions of class BTree.
- 4 data members are taken for class BTree which are M(for the degree of the BTree), a private class Node which stores array of Entry references which is another private class in BTree , n for the number of key-value pairs(key-value pair is counted as one data value), height for the height of BTree.
- Private class Entry has 3 data members: key, value, and reference of type Node.
- BTree constructor is used to initialize an empty BTree.

IV. RESULT AND ANALYSIS

We reduced the time taken to search a given record by using a B/B+ tree rather than indexing and traditional sequential access. It is concluded that disk-access times are much slower than main memory access times. Typical seek times and rotational delays are of the order of 5 to 6 milliseconds and typical data transfer rates are of the range of 5 to 10 million bytes per second and therefore, main memory access times are likely to be at least 4 or 5 orders of magnitude faster than disk access on any given system. If a storage system has a block size of B bytes, and the keys to be stored have a size of k, arguably the most efficient B+ tree is one where. Although theoretically, the one-off is unnecessary, in practice there is often a little extra space taken up by the index blocks (for example, the linked list references in the leaf blocks). Having an index block which is slightly larger than the storage system's actual block represents a significant performance decrease; therefore erring on the side of caution is preferable. If nodes of the B+ tree are organized as arrays of elements, then it may take a considerable time to insert or delete an element as half of the array will need to be shifted on average. To overcome this problem, elements inside a node can be organized in a binary tree or a B+ tree instead of an array. B+ trees can also be used for data stored in RAM. In this case, a reasonable choice for block size would be the size of the processor's cache line. The space efficiency of B+ trees can be improved by using some compression techniques. One possibility is to use delta encoding to compress keys stored into each block. For internal blocks, space saving can be achieved by either compressing keys or pointers.





V. CONCLUSION

It is concluded that disk-access times are much slower than main memory access times. Typical seek times and rotational delays [14] are of the order of 5 to 6 milliseconds and typical data transfer rates are of the range of 5 to 10 million bytes per second and therefore, main memory access times are likely to be at least 4 or 5 orders of magnitude faster than disk access on any given system. Therefore, the objective is to minimize the number of disk accesses and thus, this project is concerned with techniques for achieving that objective i.e. techniques for arranging the data on a

disk so that any required piece of data, say some specific record, can be located in a few I/O’s as possible.
1. From the above observations, it is very clear that B+ tree is better than normal indexing in every possible way.
2. Hence it is always desirable to implement B+ tree data structure to search data in an efficient manner.
3. Multilevel Indexing and is better for larger data whereas sparse indexing does well with smaller data.

REFERENCES

[1]. Data Structures using C, Aaron M. Tenembaum, Yedidyah Langsam, Moshe J. Augenstein
[2]. https://medium.com/@info.gildacademy/time-and-space-complexity-of-data-structureand-sorting-algorithms-588a57edf495
[3]. https://en.wikipedia.org/wiki/B%2B_tree
[4]. Database System Concepts taught in class and text reference textbook by Abraham Silberschatz, Henry F. Korth, and S. Sudarshan
[5]. https://www.javatpoint.com/b-plus-tree
[6]. https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree
[7]. https://en.wikipedia.org/wiki/AVL_tree
[8]. https://en.wikipedia.org/wiki/Red%E2%80%93black_tree
[9]. Bayer R. and McCreight E.M. Organization and maintenance of large ordered indices. Acta Inf., 1, 1972.
[10]. On batch-constructing B+-trees: algorithm and its performance evaluation Sang-WookKim Division of Computer, Information, and Communications Engineering, Kangwon National University, 192-1 Hyoja 2 Dong, Chunchon, Kangwon Do 200-701, Republic of Korea
[11]. Jiangkun Hu1 · Youmin Chen1 · Youyou Lu1 · Xubin He2 · Jiwu Shu1: Understanding and analysis of B+ trees on NVM towards consistency and efficiency.
[12]. Kieseberg, Peter & Schrittwieser, Sebastian & Morgan, Lorcan & Mulazzani, Martin & Huber, Markus & Weippl, Edgar. (2011). Using the structure of B + -trees for enhancing logging mechanisms of databases. International Journal of Web Information Systems. 9. 301-304. 10.1145/2095536.2095588.
[13]. Zhang D., Baclawski K.P., J. Tsotras V. (2009) B+-Tree. In: LIU L., ÖZSU M.T. (eds) Encyclopedia of Database Systems. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-39940-9_739
[14]. Kumar, M.R. & Rajendra, B.. (2015). An Improved Approach to Maximize the Performance of Disk Scheduling Algorithm by Minimizing the Head Movement and Seek Time Using Sort Mid Current Comparison (SMCC) Algorithm. Procedia Computer Science. 57. 222-231. 10.1016/j.procs.2015.07.468