iJRASET
International Journal For Research in Applied Science and Engineering Technology



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 9 **Issue:** VII **Month of publication:** July 2021

DOI: https://doi.org/10.22214/ijraset.2021.36971

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 9 Issue VII July 2021- Available at www.ijraset.com

Searching using B/B+ Tree in Database Management System

Anshita Garg[1], Anjali Rai[2], Aakash Verma[3], Anshika Srivastava[4], Aayush Aggarwal[5]
[1, 2, 3, 4]*Students,* [5]*Assistant Professor, Department of Computer Science and Engineering, AKTU, CSE, Dronacharya Group of Institutions, Greater Noida, U.P., India*

Abstract: ***This is a research-based project and the basic point motivating this project is learning and implementing algorithms that reduce time and space complexity. In the first part of the project, we reduce 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. 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. In the second part of the project, Dynamic Programming problems were solved with Recursion, Recursion With Storage, Iteration with Storage, Iteration with Smaller Storage. The problems which have been solved in these 4 variations are Fibonacci, Count Maze Path, Count Board Path, and Longest Common Subsequence. All 4 variations are an improvement over one another and thus time and space complexity are reduced significantly as we go from Recursion to Iteration with Smaller Storage.***
Keywords: Data Structures, Tree Scan, Index Scan, B Tree, DBMS

I. INTRODUCTION

B/B+ trees are extensively used in Database Management Systems because search operation is much faster in them compared to indexing and traditional sequential access. Moreover, in DBMS, the 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. 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 an efficient alternative, the B+ tree.

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees, it is assumed that everything is in the main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in the main memory. When the number of keys is high, the data is read from the 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 the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting the 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, Red-Black Tree, ..etc.

II. PROPOSED METHODOLOGY

A. Table Scan in a database

A table scan is a quite straightforward process. When your query engine performs a table scan it starts from the physical beginning of the table and goes through every row in the table. If a row matches the criterion then it includes that in the result set

A table scan is a scan made on a database where each row of the table is read in sequential order and the columns encountered are checked for the validity of a condition. Table scans are usually the slowest method of scanning a table due to the heavy amount of I/O reads required from the disk which consists of multiple seeks as well as the costly disk to memory transfers.

Normally, a full table scan is used when your query doesn't have a WHERE clause, or we can say when we want more or less every record from a table like the following query will use a full table scan: SELECT * from Employee;

©IJRASET: All Rights are Reserved

B. Index Seek in a Database

When your search criterion matches an index well enough that the index can navigate directly to a particular point in your data, that's called an *index seek*. It is the fastest way to retrieve data in a database. The index seeks are also a great sign that your indexes are being properly used.

This happens when you specify a condition in the WHERE clause like searching an employee by id or name if you have a respective index.

For example, the following query will use an index seek, you can also confirm that by checking the execution plan of this query when you run this on SQL server:

SELECT * from Employee where EmployeeId=3;

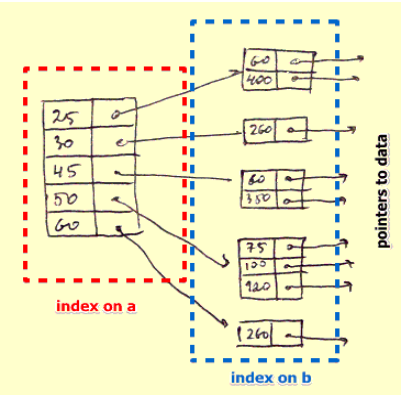
In this case, the Query Optimizer can use an index to directly go to the third employee and retrieve the data. If you look at the execution plan shown below, you can see that it uses an index seek using the index created on EmployeeId.

C. Difference Between A Table Scan And Index Seek

index seek is the fastest way to retrieve data and it comes into the picture when your search criterion is very specific. Normally, when you have a WHERE clause in your query and you are using a column that also has an index, then index seek is used to retrieve data as shown in the following query: select * from Employee where Id= 3;

D. Multilevel Index

Multilevel Indexing in a Database is created when a primary index does not fit in memory. In this type of indexing method, you can reduce the number of disk accesses to short any record kept on a disk as a sequential file and create a sparse base on that file.



E. B+tree structure

1) Every leaf node is at an equal distance from the root node. A B+ tree is of the order n where n is fixed for every B+ tree

2) Internal Nodes

a) Internal (non-leaf) nodes contain at least ⌈n/2⌉ pointers, except the root node.

b) At most, an internal node can contain n pointers.

3) Leaf Nodes

a) Leaf nodes contain at least ⌈n/2⌉ record pointers and ⌈n/2⌉ key values.

b) At most, a leaf node can contain n record pointers and n key values.

c) Every leaf node contains one block pointer P to point to the next leaf node and forms a linked list.

F. The Description Along With Enhanced User Efficiency

Tables are usually arranged in the file system in a particular manner and that hierarchy comprises of Extents, pages, rows, columns where the details of the tables are present, Now applying normal search that is table scan is not an optimum way of searching as in case of worst-case scenario that searching goes till the last page, Also this type of searching that is table scan can even be worst when the user is firing in between query So to avoid this type of searching index-based searching concept is used. In this type of searching all the data is arranged in the tree hierarchy that makes searching optimized. Here every node of the tree is treated as record pointers which means it contains the address of the other node where there is the possibility of data fetching.

©IJRASET: All Rights are Reserved

We know that data is stored in the form of records. Every record has a key field, which helps it to be recognized uniquely. Indexing is a storage structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done. Indexing in database systems is similar to what we see in books.

B+ Tree Representation for efficient searching

Node Type	Children Type	Min Number of Children	Max Number of Children	Example $b = 7$	Example $b = 100$
Root Node (when it is the only node in the tree)	Records	1	$b - 1$	1–6	1–99
Root Node	Internal Nodes or Leaf Nodes	2	b	2–7	2–100
Internal Node	Internal Nodes or Leaf Nodes	$\lceil b/2 \rceil$	b	4–7	50–100
Leaf Node	Records	$\lceil b/2 \rceil$	b	4–7	50–100

III. IMPLEMENTATION DETAILS

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 provide an efficient structure for housing the data itself (this is described in[4]:238 as index structure "Alternative 1").

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 that 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 processor's cache line size. 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. For string keys, space can be saved by using the following technique: Normally the i-th entry of an internal block contains the first key of the block. Instead of storing the full key, we could store the shortest prefix of the first key of the block that is strictly greater (in lexicographic order) than the last key of block i. There is also a simple way to compress pointers: if we suppose that some consecutive blocks are stored contiguously, then it will suffice to store only a pointer to the first block and the count of consecutive blocks.

All the above compression techniques have some drawbacks. First, a full block must be decompressed to extract a single element. One technique to overcome this problem is to divide each block into sub-blocks and compress them separately. In this case, searching or inserting an element will only need to decompress or compress a sub-block instead of a full block. Another drawback of the compression technique is that the number of stored elements may vary considerably from one block to another depending on how well the elements are compressed inside each block.

©IJRASET: All Rights are Reserved

A. Implementation Blocks

1) Insertion

a) Initialize x as root.

b) While x is not a leaf, do the following

- Find the child of x that is going to be traversed next. Let the child be y.
- If y is not full, change x to point toy.
- If y is full, split it and change x to point to one of the two parts of y. If k is smaller than the mid key in y, then set x as the first part of y. The other second part of y. When we split y, we move a key from y to its parent x.

c) 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.

2) Searching

a) Initialize x as root.

b) While x is not a leaf, do the following:

- Find the key-value which is equal to the value to search, if found return the index.
- Otherwise go to the node which may contain the 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.
- Recursively call the node.

c) The recursion in step 2 stops when x is leaf and we have not found a value to be searched. In that case, we return the key not found.

B. Code Structure

1) 2 classes are made for the BTree implementation: One is the client which is used to run all the functions of class BTree.

2) 4 data members are taken for class BTree which are M(for the degree of the BTree), a private class Node which stores an 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.

3) Private class Entry has 3 data members: key, value, and reference of type Node.

4) BTree constructor is used to initializing an empty BTree.

5) isEmpty() function –returns true if the symbol table is empty.

6) size() function returns the number of key-value pairs

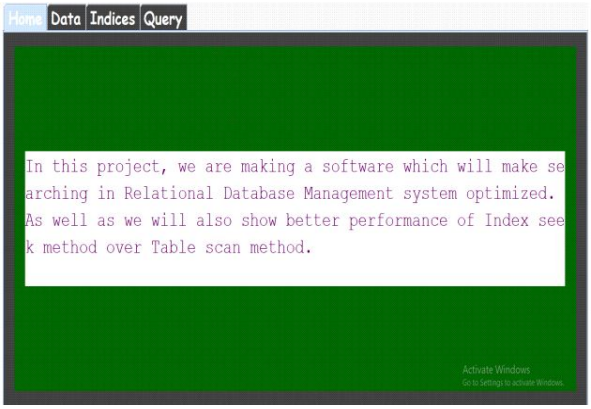
7) height() function returns the height of the tree(useful for debugging).

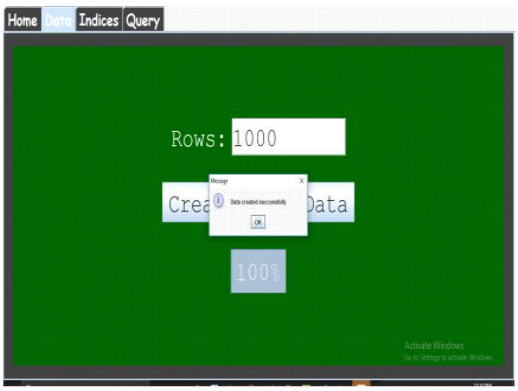
8) get() function returns the value associated with the key that is passed as the parameter.

9) get() function is implemented as the algorithm discussed earlier.

10) put() function inserts the key-value pair into the symbol table overwriting the old value with the new value if the key is already in the symbol table.

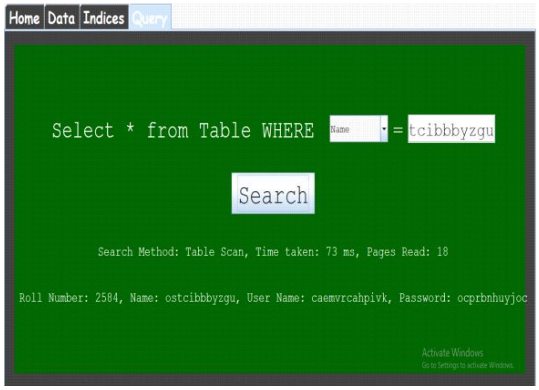
11) put() function uses function insert (for the node which is not full) and function split (for insertion in a node which is full) and is implemented as the algorithm discussed earlier.

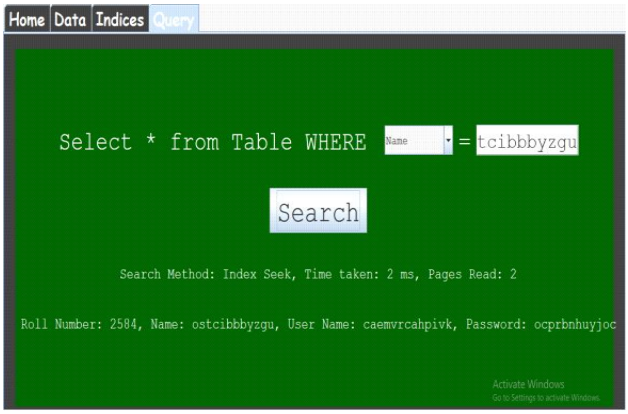




©IJRASET: All Rights are Reserved







IV. CONCLUSIONS

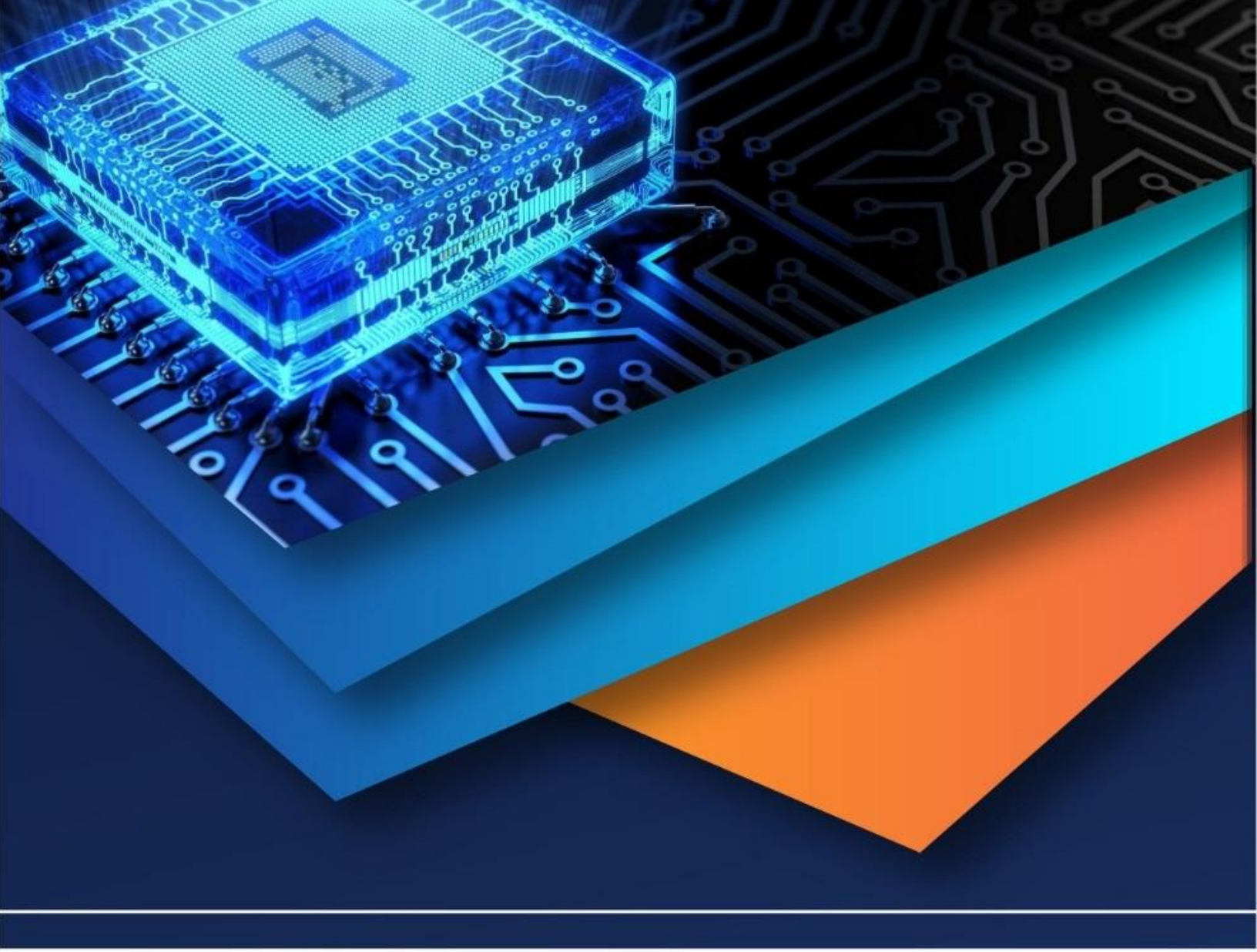
It is concluded that disk-access times are much slower than main memory access times. Typical seek times and rotational delays are of the order of 5 to 6 milliseconds and typical data transfer rates are of the range of 5 to 10 million bytes per second and therefore, main memory access times are likely 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 a few I/O's as possible

A. From the above observations, it is very clear that a B+tree is better than normal indexing in every possible way.
B. Hence it is always desirable to implement a B+ tree data structure to search data efficiently.
C. Multilevel Indexing is Better for larger data whereas sparse indexing does well with smaller data
D. Bull loading algorithm can be implemented to improve upon insertion in the B/B+ tree which is O(log n). The conventional method is to implement using a top-down fashion from the root node to the leaf node.
E. Bulk loading can also be implemented using a bottom-up fashion from the leaf node to the root accessing only one level at a time.
F. One could then compare the statistics and decide which way would be better for bulk loading.
G. The great commercial success of database systems is partly due to sophisticated query optimization technology development. These techniques can be further applied to all the applications of Dynamic Programming.

REFERENCES

[1] Database System Concepts taught in class and text reference textbook by Abraham Silberschatz, Henry F. Korth, and S. Sudarshan
[2] Bulk loading in https://en.wikipedia.org/wiki/B
[3] B+Tree in https://en.wikipedia.org/wiki/B%2B_tree
[4] Dynamic Programming: https://en.wikipedia.org/wiki/Dynamic_programming
[5] Donald Kossmann and Konrad Stocker, "Iterative Dynamic Programming: A New Class of Query Optimization Algorithms".
[6] Data Structures and Algorithms in Java, Michael T.Goodrich, RobertoTamassia, Michael H. Goldwasser

©IJRASET: All Rights are Reserved



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)