International Journal of Advances in Engineering and Management (IJAEM)
Volume 2, Issue 11, pp: 250-252 www.ijaem.net ISSN: 2395-5252

RDBMS indexing using B+ trees

Karan Bali, Dr. Sunil Maggu
Student, Department of IT, MAIT (Rohini), affiliated to GGSIPU
(Mentor)

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

ABSTRACT: This is a research-based project and the point motivating this project is learning and implementing algorithms that reduces time and space complexity.In 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.

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, 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 column 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. 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 his 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, Red-Black Tree, ..etc.

II. METHODOLOGY

In this project I made a software which will make searching in RDBMS optimized . It will also show better performance of index seek method for searching over table scan method.
The project can be used to create references using any of the data information

.It can be on the basis of ID, name , username OR password. Storing the references in the form of a b tree helps optimize the search operation to a great extent. Once we want to create the indices on the basis of the name , we generate indices and create another file that stores indices . In order to create the indexes the names were sorted and then divided in groups of 2000. Then these were divided and segregated in the form of B tree which decided the number of levels the B tree will have. I am creating random data in tables / columns. Like in any database this is stored in the form of extents and pages. Create_data() function is used to create random data in Java by specifying the number of rows of data we need. Get_random_word function was used for this purpose. This was the function that used the inbuilt Math.random() function which is inbuilt in Java.

Then the ASCII value was used to create random names from randomly created numbers from the above function.
Considering normal storage of data in the backend the data is stored in extends. Each extend is made up of 64kb of data
.One extend is divided in 8 pages.
The data is generated in the form of ID / name / username / password.
Now to access any particular piece of data we have to access using the table scan method which is somewhat equivalent to linear search.
Now in order to reduce the time we create indexing and store them in the form of a b tree. Normally in a binary tree a node contains one data entry only but here as we are using a b tree we can enter multiple data references.
In order to reduce the page scan I have created indexes. This helps us create nodes in the form of a tree which makes searching easier. Let there be 8 x 10^9 kB data. The number of pages formed are 10^9 as each page contains 8 KB of data. So when we have to search normally in tables scan method , let on a 10^6 th page it takes a considerable amount of time.

Here , let if we were making indexes on the basis of ID 1 ID is 4 bits, one page can have 8kb data so one index page has 2000 ids. This means reference of 2000 rows can be stored in one page.

Thus starting from the root node let even if we have 8x10^9 data entries we will be able to cover them within 3 Page scans as we are covering only 3 B tree nodes starting from the root to the leaf node.

Normal SQL queries are used to search for a name. The leaf node of the B tree contains the references of the data page. On the data page is the actual data present. No inbuilt or self made trees were used in the project. Rather only the data was stored in the form of a b tree which gave the illusion of access of data like that in a b tree.

III. RESULT

The final output of the project begins with a start page having 4 panels at the top. The panels represent the home,data,indices and query page.
Once in the data page, the project asks us the number of rows of data that needs to be created.We enter 10000 rows. Once this is done we get data pages created of random data entries in a separate folder that represents a total of 10000 rows.
Then the indices page asks the parameter on which the indices need to be created. Once we enter NAME as the parameter. Further another index folder is created that holds the indices stored of the given data pages.
The indices, store the references as range of data in one index. Hence we need to scan only that particular range for the given name search.
Once this is done, search query is generated from the Query page. Here we pick a random name that we need to search for from the data page. Enter the name in the search query.
Here for the test case taken name in the system is “gezoqnkm”.
When considering normal tables scan method, the number of pages read in order to search for a name are 42. This includes a total of 6 extents. And scanning on the third page of 6th extent. The total time taken was 17 millisecond.
Once the indexes are created finally searching for the same name but using the index seek method we are able to do the same in 2 millisecond and then number of pages read are only two these involve starting the root then the leaf and finally reaching the data page.
Here whatever we did ,we did with the help of nonclustered indexing. We did not edit anything in the original data rather we created indexes for faster access.

IV. CONCLUSION

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

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 is Better for larger data whereas sparse indexing does well with smaller data

V. FUTURE SCOPE OF WORK:

1. Bull loading algorithm can be implemented to improve upon insertion in B/B+ tree which is O(log n). The conventional method is to

implement using a top-down fashion from the root node to leaf node.

2. 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.
3. One could then compare the statistics and decide which way would be better for bulk loading.
4. The great commercial success of database systems is partly due to the development of sophisticated query optimization technology. These techniques can be further applied to all the applications of Dynamic Programming.

REFERENCES

[1]. Data Structures and Algorithms in Java, Michael T.Goodrich, RobertoTamassia, Michael H. Goldwasser
[2]. Data Structures using C, Aaron M. Tenembaum, Yedidyah Langsam, Moshe J. Augenstein
[3]. https://medium.com/@info.gildacademy/time-and-space-complexity-of- data-structureand-sorting- algorithms-588a57edf495
[4]. https://en.wikipedia.org/wiki/B%2B_t ree
[5]. Database System Concepts taught in class and text reference textbook by Abraham Silberschatz, Henry F. Korth, and S. Sudarshan
[6]. https://www.javatpoint.com/b-plus- tree
[7]. https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree
[8]. https://en.wikipedia.org/wiki/AVL_tre e
[9]. https://en.wikipedia.org/wiki/Red%E2%80%93black_tree
[10]. Arbel-Raviv, M., Morrison, A., Trevor, B.: Getting to the root of concurrent binary search tree performance. In: ATC, Boston, MA, USA (2018)
[11]. R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious B+
-trees. In SIGMETRICS, 2003
[12]. Jiangkun Hu1 · Youmin Chen1 · Youyou Lu1 · Xubin He2 · Jiwu Shu1: Understanding and analysis of B+ trees on NVM towards consistency and efciency
[13]. Using the Structure of B+-Trees for Enhancing Logging Mechanisms of Databases: Peter Kieseberg Sebastian Schrittwieser Lorcan Morgan Martin Mulazzani Markus Huber Edgar Weippl SBA Research Vienna,Austria