B$^{\text{point}}$-tree: An Indexing Structure for Efficient Search in Data Retrieval

Sufyan Almajali*, Ghazi Al-Naymat†, and Ayah Atiyah‡
King Hussein School of Computing Sciences, Princess Sumaya University for Technology
Amman, Jordan
Email: *s.almajali@psut.edu.jo, †g.naymat@psut.edu.jo, ‡aya20168009@std.psut.edu.jo

***Abstract*—The amount of cheap memory growing enables all data to be in main memory databases, this adds a critical performance advantage to the main memory databases. In order to keep and retrieve data effectively, indexing schemes/ systems have been proposed. However, existing indexing algorithms are poorly suited for effective search, not just because of the space efficiency, but also due to the fact that they are unable to execute per every query within a tight time budget. Satisfying such a standard requires a B-tree indexing algorithm to be capable of controlling its memory response time to provide superior search performance. The goal of this research is to present a new technique Bpoint-tree to enhance the effectiveness of indexing search with a new data structure to the conventional B-tree algorithm. The results of the Bpoint-tree performance have been compared to the conventional B-tree, they show that Bpoint-tree exceed the conventional B-tree. The results show that the Bpoint-tree is able to improve the indexing performance effectiveness.**

***Index Terms*—Indexing, query processing, B-tree, access methods, locking.**

I. Introduction

For many years, Main Memory has been a vital factor that adds a vital advantage to current and future Databases as discussed in [1], [2], [3]. In Main Memory Databases (MMDB), indexing schemes are expected to decrease the number of Input/ Output processes to the disk, which decrease the cost of file search and recovery. In database indexing schemes, B-tree is a beneficial indexing scheme that has received much attention for many years [4]. The B-tree allows users to have concurrent accesses on system databases [5].

Similar to the other indexing schemes, B-trees aim to map between information and the relevant search keys in databases and file systems. B-trees enhance queries and support query execution based on some sorting algorithms [6], [7]. A considerable amount of literature had been published on indexing techniques. [8], [9]. Indexes can be searched through either exact-match value or range-query lockup. In the latter, sort-based is allowed per each query execution such as join and explicit join. In the light of recent research, there is a considerable concern about multi-dimensional queries and data. For instance, [10], [11], [12] have addressed the question of multi-dimensional data lockups. Multi-dimensional data consist of *measures* and *dimensions*. A *hierarchy* is placed for each *dimension*. A dimension depends on some attributes that describe its elements.

978-1-5386-7167-2/18/$31.00 ©2018 IEEE

The basic idea of the B-tree structure is simple and easy to understand. However, the design and development of the tree nodes on disk storage seem to be challenging. Moreover, fixed-length and variable-length records require an additional implementation effort. A major difficulty flaw of B-tree index is in the concurrency control when many users are served using asynchronous I/Os. Therefore, concurrent threads are required to ensure asynchronous activities in database servers. Techniques to solve this issue are time consuming and subject to high overheads.

The B-tree is an indexing structure; its nodes have search keys. The B-tree holds a number of data pointers, left pointer pointing to the left subtree and right pointer pointing to the right pointer. Individually, a leaf B-tree points to the actual records (data instances). Also, the tree pointers in a B-tree mark the corresponding data indexes in the memory, since the data records can be accessed by the search keys who point to them [13], [14].

Despite the B-trees implemented earlier, implementing a B-tree that is able to solve the time issue in concurrency control of database users is so far a research material. Achieving an accurate and fast response to the user's lockups in a database system is the aim of our research by presenting a novel B-tree indexing algorithm. The paper is distributed as the following. The related topics are discussed in Section II. Section III is presents the proposed algorithm. Experiments and results are described in Section IV. The conclusion is done in Section V.

II. Related Topics

The primary topic connected to our research is B-tree. The base of B-tree indexing is a familiar transaction concept [15], [16]. B-tree indexing has been published in several researches [17], [18], [19]. In [20], the authors suggested an indexing technique that helps in improving the historical data relations. However, their work requires a time ordered data of the tuples to place a particular tuple. In [21], one of the first to examine the indexes only on magnetic disks. Author draws our attention to conventional multiple attribute indexes. Specifically, the indexes that are associated with the minimum and maximum transactions commit times. In spite of that, we argue that the use of indexes is of limited benefit when they have no place in optical disks. In other words, at least part of the indexes in optical disks should be taken into consideration.

Bayer [22] provided a balanced B-tree for what they previously called once-write. In contrast to [21], his proposal can be used completely on optical disks. Like [23], [24] presented an indexing scheme that entirely positioned on optical disks. His approach seems to be reliable as it focuses on a fast and efficient access of the last inserted elements in B-trees. Our algorithm bears a close to Vitter's efficient access property. The first investigations into R-tree was performed by [25]. R-tree is an indexing structure that allows multi-dimensionality in data indexing. While we investigate B-tree indexing performance, R-tree has been widely investigated in index structures [26], [27], [28], [29]. Thus, their works are completely different from ours.

In [30] and [31], they introduced multiple locking mechanisms for B*-tree in which all the data records are stored in the leaves, and the rest of the keys are stored in the other internal nodes. Figure 1 shows a sample node of B*-tree [31]. B*-tree supports high level concurrency and avoids deadlock. They argued that" The most efficient data structure for databases and file systems is B-tree". Regardless of the surprising number of studies that have been done on B-tree, their claims seem to be well-founded until now. [32] proposed a B$^{\text{link}}$-tree mechanism

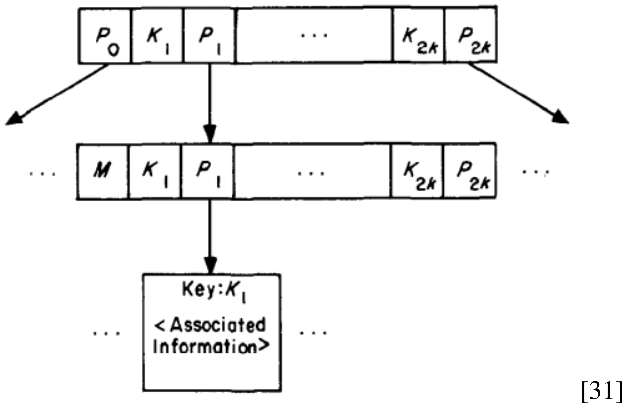

Figure 1. B*-tree Nodes

as a solution for the latching problem in the database systems. The tree structure not only has a parent-to-child pointer, but it also contains left-to-right pointers. Also, each level of the proposed b-tree is represented as a singly-linked-list. In the case of overflow, if a new node has to be inserted, the node with an overflow keeps both a pointer to the new node and a key value that is used to separate the key values that are kept between an old and new node. In [33], a multi-level transaction indexing scheme is presented. The optimistic approach in concurrency control have been hypothesized in several studies, such as [34].

In the field of open nested transactions, various techniques can be found [35], [36]. In [37], Lomet provided a new definition of database locking, in which he designed the key-range locking as a multi-granularity and hierarchal locking. A serious criticism of Lomet's approach is the extra lock-modes requirements to employ concurrency control. Some preliminary work was carried out several years ago by [38]. Their work is based on ARIES/KVL to investigate the lower level lockups. All of the previous related works are completely different from ours.

III. The Proposed Algorithm

For accelerating query lockups in MMDB, a new indexing algorithm is proposed. This proposed algorithm contains short-cut pointers added to the normal B-tree in aim of fast access of database records. In this algorithm, the number of memory accesses is reduced by reducing the depth of the search tree. In the presented algorithm, every single B-tree node has a lockup key, and every new data record is added to the tree as a new node. Also, counts are inserted into a B-tree node in aim of reading the repetition of any frequency group of all key indexes.

In B-tree indexing structure, all the tree nodes contain a lockup data, and each node points to the left and right subtrees. Moreover, the traversal of costs $\mathcal{O}(\log n)$ time, (N denotes the number of the tree nodes). However, the proposed algorithm aim to decrease the time complexity of the B-tree to become $\mathcal{O}(1)$, while lowering the total number of tree rotations, that to exceed the conventional B-tree method. The normal B-tree has been modified to the new algorithm by adding a data structure in aim of increasing the efficiency. The proposed data structure contains a number of shortcut pointers point to the most frequently used data records in the database (see Figure 2).

The proposed B$^{\text{point}}$-tree contains an array that has most used N nodes that are modified constantly in every query execution. Moreover, the array's nodes has pointers to the records that have the maximum number of counts. The array represents a shortcut in query lockup operations. Compared to other B-tree lockup techniques, the added value of B$^{\text{point}}$-tree is the lower number of data traversals for every query execution. In this research, the presented technique is able to handle many users in a database system and improve the system efficiency. We expect the new algorithm to lower the time complexity to a constant time which may consequently improve the query lockup efficiency. Bpoint-tree technique has been applied in the previous works [39], [40]

In the conventional B-tree index, every node contains a lockup value which represents the matched index for a data query, and group of indexes are traced using counts. Most frequently used nodes are continuously adjusted. Also, the array is added to sort them and track the counts instantly, the required data entries are shown using the maximum numbers

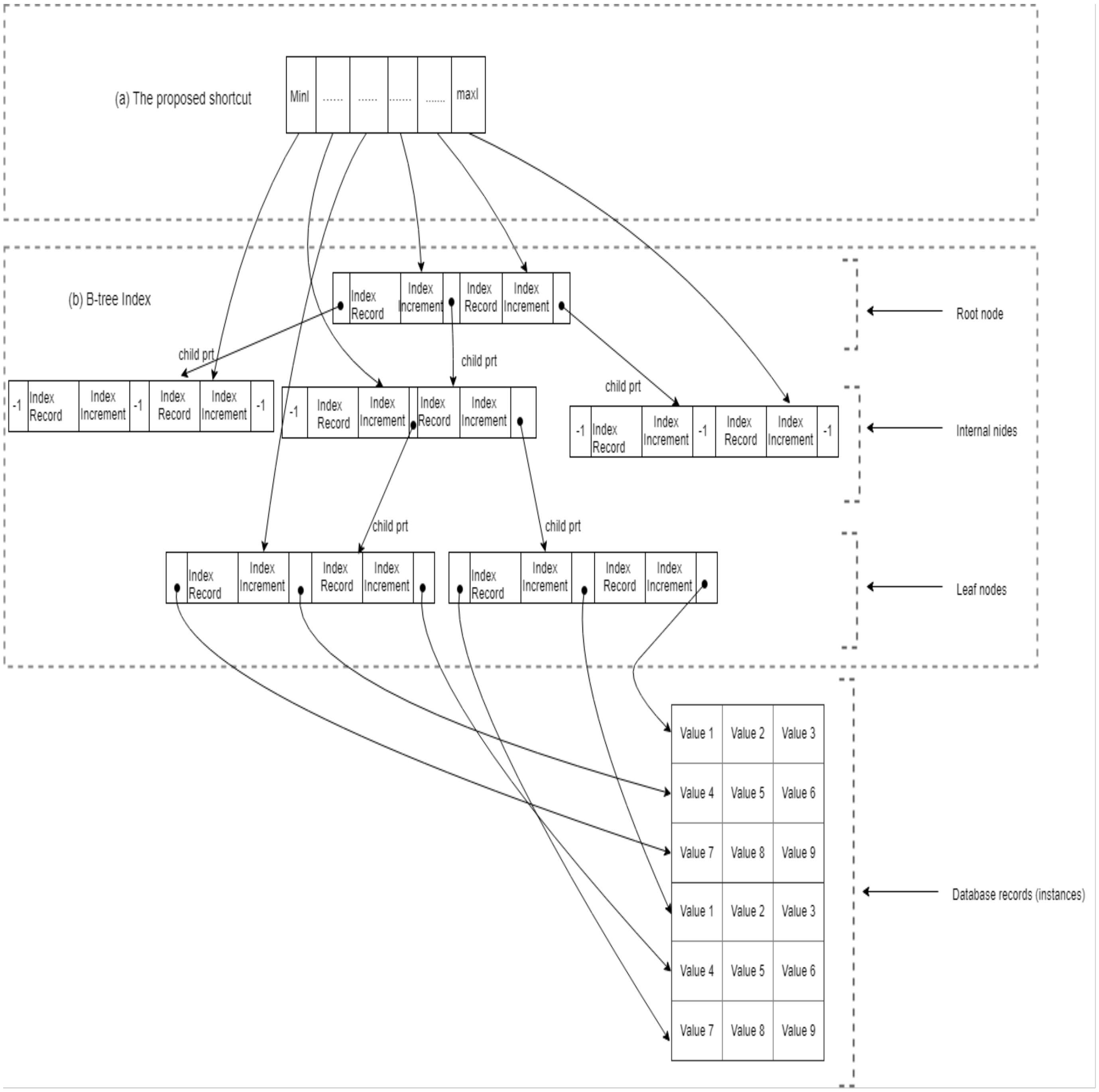

Figure 2. The modified B-tree with Shortcut Pointer Structure

that should be tracked in this sorted array. This process could lead to an increase in the number of memory accesses, due to the need for the statistics to be collected on each B-tree traversal, an array is therefore should be added to be used to insert and sort the nodes. The proposed method contains of 4 main sections, which are interpreted as the following:

A. Search

The pointer's value is constantly being modified in the search module. The last updated index should be looked up and matched with the last inserted value into the shortcut array. We call this value the shortcut index. If the key is found, it will add the count, after that the pointer's key is looked for a larger value than the last one traced. Finally, a swap is performed as the array is sorted.

B. Update

Update section performs tracking of every repeated index group inserted, when calling the index, if the new index group matched the last one tracked, it performs modifications to the record counts instead of performing a tree traversal which is costly. Shortcut index refers to a non-replicated index group.

C. Insertion with Shortcut

The sorted array in this section contains the most frequently used N nodes that are modified constantly for every data query. By tracking every index, if it was one of the existing indexes, it increases the count with no need to perform a complete traversal for the tree such as in the case of the conventional B-tree index. If the index will match a node key, the count is increased by one, otherwise, it is added to the tree as a new node. The pseudo code of the proposed shortcut algorithm is shown below.

Algorithm 1 Shortcut Insert

```
1: procedure INSERT WITH SHORTCUT
2: Require: $S \leftarrow \{\}$ where $S$ is shortcut array
3:   While New Index Group Added do
4:      $R \leftarrow$ New Inserted Index Group
5:      Read index NDX
6:      Match NDX to last index LNDX
7:      if $NDX = LNDX$ then
8:        $Entry \leftarrow LastEntry$
9:        Entry Increment $\leftarrow$ Entry Increment + 1
10:       if $Entry \notin S$ then
11:         if EntryIncrement $\geqslant$ S Entry Min Increment then
12:           $S \leftarrow$ Add Entry
13:           Entry Increment $\leftarrow$ Entry Increment + 1
14:         end if
15:       end if
16:       $LastEntry \leftarrow Entry$
17:     else
18:       Do Normal B-tree Insert
19:     end if
20:   end while
```

$$NDX := Index$$

$$LNDX := LastIndex$$

$$S := Shortcut$$

IV. EXPERIMENTS AND RESULTS

The proposed algorithm has been implemented using C++ programming language. The experiments have been conducted on the Bpoint-tree using different parameters: shortcut array sizes, number of index groups, number of indexes in each group, and percentage of repeated indexes in each group. As forecast, our experiments prove that B$^{\text{point}}$-tree has the potential to outperform the normal B-tree. This is by cause of that the B-Tree requires more memory accesses and recovery operations than B$^{\text{point}}$-tree. Moreover, the normal B-tree costs higher locking overhead. Two indexing approaches over the B-Tree are presented, the results of the simulation study indicate that the B$^{\text{point}}$-tree outperforms the B-Tree performance.

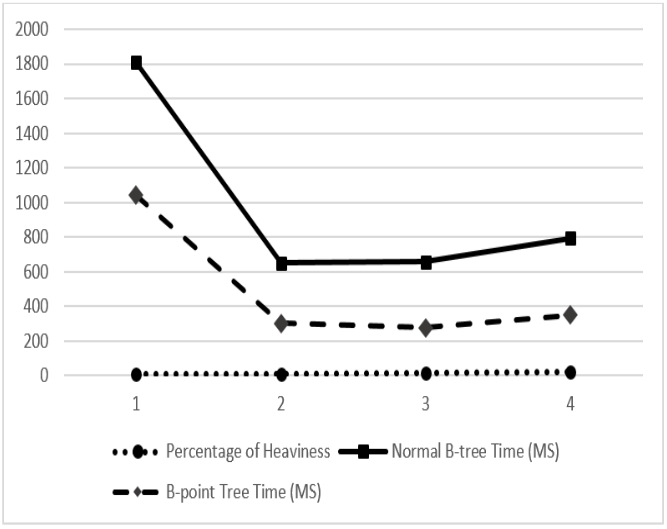

Figure 3. Experiments with 10000 indexes, 100000 index groups and shortcut size of 5

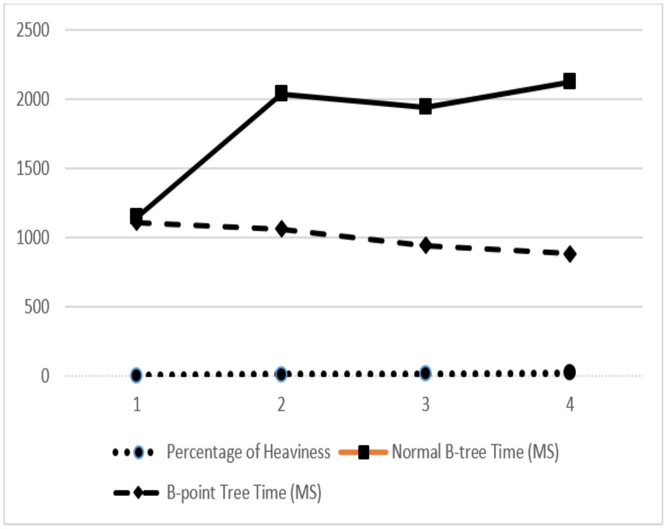

Figure 4. Experiments with 10000 indexes, 100000 index groups and shortcut size of 7

In Figures 3, 4 and 5, the results in Bpoint-tree starts to exceed the conventional one when the number of indexes is increased to 10000. However, Bpoint-tree costs almost 50 percent of the time spent by the conventional tree when the size is 5. Also, Bpoint-tree takes much less time when percentages of heaviness are 15 or 20 when rising the array is 7. When it is equals to 5 or 10, the same time is spent approximately. The

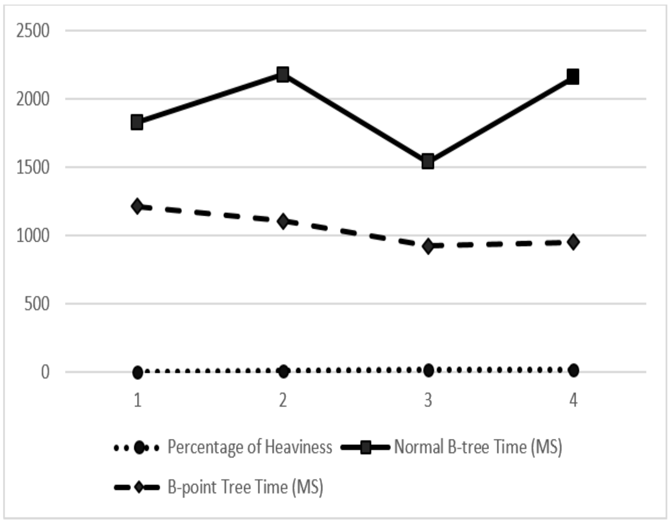

Figure 5. Experiments with 10000 indexes, 100000 index groups and shortcut size of 10

results shown in 5 and 7 are the same whenever the size is 10. This finding suggests that when the percentage in this certain shortcut array size is increased, the performance is improved in the Bpoint-tree. This was the most surprising finding to emerge from the data.

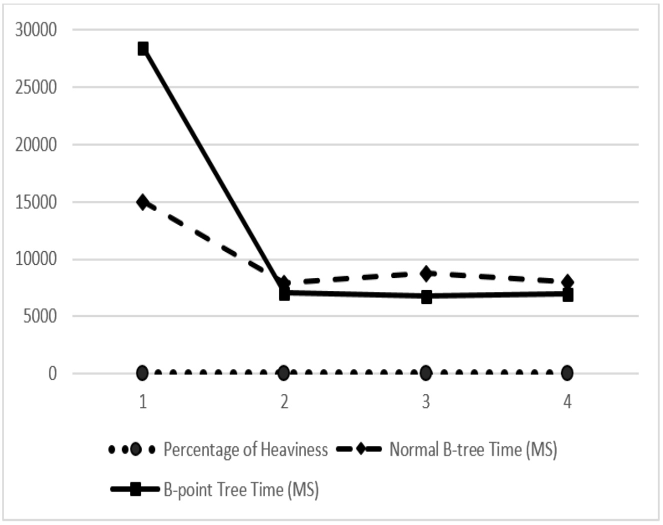

Figure 6. Experiments with 10000 indexes, 1000000 index groups and shortcut size of 5

The observed decrease in Bpoint-tree performance could be attributed to the increasing number of groups to 1000000. Also, except for some cases such as the percentage of 20, Bpoint-tree and the conventional B-tree performances are similar. In this experiment, Bpoint-tree represents a revolutionary back-up for the conventional B-tree index by 20% 30% as shown in figures 6, 7, and 8.

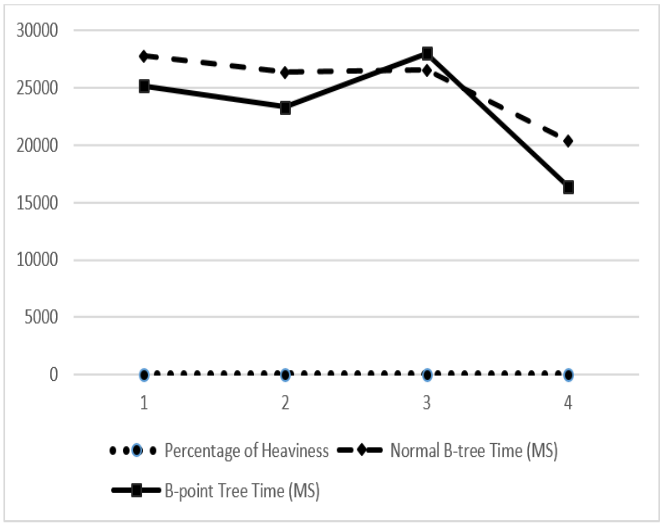

Figure 7. Experiments with 10000 indexes, 1000000 index groups and shortcut size of 7

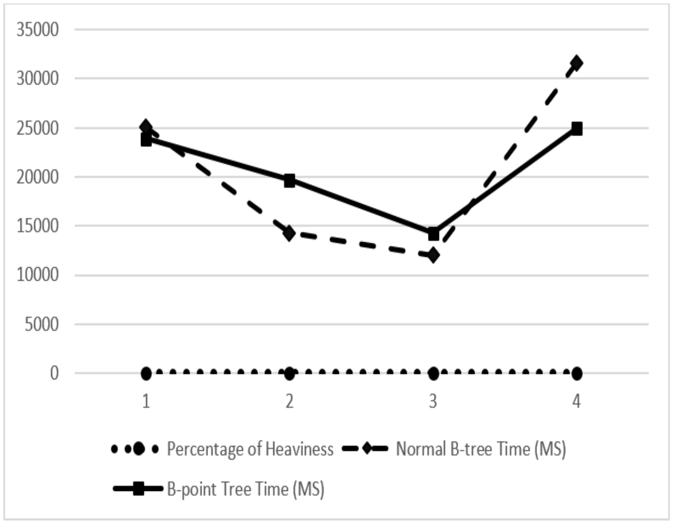

Figure 8. Experiments with 10000 indexes, 1000000 index groups and shortcut size of 10

V. Conclusion

B-tree indexing schemes are expected to decrease the number of Input/ Output processes to the disk. Developing a B-tree that is able to Retrieve and keep data efficiently of database users seemed to be a challenging task. This paper proposed a technique called B^{point}-tree to accelerate the lockup process using an additional data structure that holds shortcut pointers point to the normal B-tree index. The findings of the experiments show that the B^{point}-tree exceeds the conventional B-tree index. This indicates that applying the shortcut algorithm manages to lower the lockup operation time, therefore enhances the performance efficiency of the B-tree index.

References

1 Lee, S. K., Lim, K. H., Song, H., Nam, B., and Noh, S. H., "Wort: Write optimal radix tree for persistent memory storage systems." in *FAST*, 2017, pp. 257–270.

2 Marathe, N. and French, B., "Append-only b-tree cursor," Mar. 14 2017, uS Patent 9,594,786.

3 Heman, S. A., Boncz, P. A., Zukowski, M., and Nes, N. J., "Methods of operating a column-store database engine utilizing a positional delta tree update system," Feb. 13 2018, uS Patent 9,892,148.

4 Sharma, S., Sharma, M., Jain, R., and Khatri, S. K., "Nlcs based string approximation for searching indexing keywords in b-tree," in *2017 2nd International Conference on Telecommunication and Networks (TEL-NET)*. IEEE, 2017, pp. 1–4.

5 Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J., Hunter, J., and Barnett, M., "Faster: an embedded concurrent key-value store for state management," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1930–1933, 2018.

6 Silberschatz, A., Korth, H. F., Sudarshan, S. *et al.*, *Database system concepts*. McGraw-Hill New York, 1997, vol. 4.

7 Bernstein, P. A., Hadzilacos, V., and Goodman, N., "Concurrency control and recovery in database systems," 1987.

8 Bertino, E., "An indexing techniques for object-oriented databases," in *Data Engineering, 1991. Proceedings. Seventh International Conference on*. IEEE, 1991, pp. 160–170.

9 Zobel, J., Moffat, A., and Sacks-Davis, R., "An efficient indexing technique for full-text database systems," in *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*. INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS (IEEE), 1992, pp. 352–352.

10 Idris, F. and Panchanathan, S., "Review of image and video indexing techniques," *Journal of visual communication and image representation*, vol. 8, no. 2, pp. 146–166, 1997.

11 Bayer, R., "The universal b-tree for multidimensional indexing: General concepts," in *International Conference on Worldwide Computing and Its Applications*. Springer, 1997, pp. 198–209.

12 Ramsak, F., Markl, V., Fenk, R., Zirkel, M., Elhardt, K., and Bayer, R., "Integrating the ub-tree into a database system kernel." in *VLDB*, vol. 2000, 2000, pp. 263–272.

13 Wirth, N., *Algorithms+ Data Structures= Programs Prentice-Hall Series in Automatic Computation*. Prentice Hall, 1976.

14 Bentley, J. L., "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

15 Johnson, T. and Shasha, D., "A framework for the performance analysis of concurrent b-tree algorithms," in *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, 1990, pp. 273–287.

16 Jensen, C. S., Lin, D., and Ooi, B. C., "Query and update efficient b+-tree based indexing of moving objects," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 2004, pp. 768–779.

17 Lanin, V. and Shasha, D., "A symmetric concurrent b-tree algorithm," in *Proceedings of 1986 ACM Fall joint computer conference*. IEEE Computer Society Press, 1986, pp. 380–389.

18 Mohan, C., *ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes*. Citeseer, 1989.

19 Li, Y., He, B., Luo, Q., and Yi, K., "Tree indexing on flash disks," in *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. IEEE, 2009, pp. 1303–1306.

20 Lum, V., Dadam, P., Erbe, R., Günauer, J., Pistor, P., Walch, G., Werner, H., and Woodfill, J., "Designing dbms support for the temporal dimension," in *ACM Sigmod Record*, vol. 14, no. 2. ACM, 1984, pp. 115–130.

21 Ahn, I., "Performance modeling and access methods for temporal database management systems," Ph.D. dissertation, Citeseer, 1986.

22 Bayer, R. and McCreight, E., "Organization and maintenance of large ordered indices," in *Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*. ACM, 1970, pp. 107–141.

23 Easton, M. C., "Key-sequence data sets on indelible storage," *IBM Journal of Research and Development*, vol. 30, no. 3, pp. 230–241, 1986.

24 Vitter, J. S., "An efficient i/o interface for optical disks," *ACM Transactions on Database Systems (TODS)*, vol. 10, no. 2, pp. 129–162, 1985.

25 Guttman, A., *R-trees: A dynamic index structure for spatial searching*. ACM, 1984, vol. 14, no. 2.

26 Kothuri, R. K. V., Ravada, S., and Abugov, D., "Quadtree and r-tree indexes in oracle spatial: a comparison using gis data," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 2002, pp. 546–557.

27 Yu, S., Cai, G., Li, W., and Xie, J., "Top-k frequent spatial-temporal words query based on r-tree," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*. IEEE, 2018, pp. 423–428.

28 Sharrma, R. and Gupta, A., "Implementing and evaluating r-tree techniques on concurrency control and recovery with modifications on non-spatial domains," in *Quality, IT and Business Operations*. Springer, 2018, pp. 203–211.

29 Hadjieleftheriou, M., Manolopoulos, Y., Theodoridis, Y., and Tsotras, V. J., "R-trees: A dynamic index structure for spatial searching," in *Encyclopedia of GIS*. Springer, 2017, pp. 1805–1817.

30 Bayer, R. and Schkolnick, M., "Concurrency of operations on b-trees," *Acta informatica*, vol. 9, no. 1, pp. 1–21, 1977.

31 Lehman, P. L. *et al.*, "Efficient locking for concurrent operations on b-trees," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 4, pp. 650–670, 1981.

32 Lehman, T. J. and Gottemukkala, V., "The design and performance evaluation of a lock manager for a memory-resident database system." 1996.

33 Weikum, G., "Principles and realization strategies of multilevel transaction management," *ACM Transactions on Database Systems (TODS)*, vol. 16, no. 1, pp. 132–180, 1991.

34 Kung, H.-T. and Robinson, J. T., "On optimistic methods for concurrency control," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 2, pp. 213–226, 1981.

35 Weikum, G. and Schek, H.-J., "Concepts and applications of multilevel transactions and open nested transactions," 1992.

36 Ni, Y., Menon, V. S., Adl-Tabatabai, A.-R., Hosking, A. L., Hudson, R. L., Moss, J. E. B., Saha, B., and Shpeisman, T., "Open nesting in software transactional memory," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2007, pp. 68–78.

37 Lomet, D. B., *Key range locking strategies for improved concurrency*. Digital Equipment Corporation, Cambridge Research Laboratory, 1993.

38 Gray, J. and Reuter, A., *Transaction processing: concepts and techniques*. Elsevier, 1992.

39 Atiyah, A. and Almajali, S., "A traffic tracking algorithm for a fast detection of active network sources," in *Proceedings of the 2nd International Conference on Future Networks and Distributed Systems*. ACM, 2018, p. 16.

40 Atiyah, A., Jusoh, S., and Almajali, S., "An efficient search for context-based chatbots," in *Proceedings of the 8th International Conference on Computer Science and Information Technology*. IEEE, 2018.