TEMPORAL PERFORMANCE ANALYSIS OF BATCH OPERATIONS IN INDEX STRUCTURES

Project Report for

Database Management Systems (CSE2004)

submitted by

YASH GUPTA	15BCE2073	D1
NIKITA NEGI	16BCE2038	D1
KEVIN ABRAHAM	16BCE0983	D1

under the faculty

Dr. Swathi J. N.

in partial fulfillment for the award of the degree of

BACHELOR IN TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

VIT® UNIVERSITY (Estd. u/s 3 of UGC Act 1956) VELLORE ▪ CHENNAI www.vit.ac.in

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING

NOVEMBER, 2017

Temporal Performance Analysis of Batch Operations in Index Structures (November 2017)

Yash Gupta, *15BCE2073*, Nikita Negi, *16BCE2038*, and Kevin Abraham, *16BCE0983*

Abstract— **The most famous commercial database softwares are disk-based database systems, i.e. all create, read, update and delete operations have to reference the primary storage location for data storage and retrieval. The requirement for high-performance data access, coupled with the latest advances in hardware technology has led to the emergence of various main-memory database management systems such as SQLite, TimesTen, MeanSQL etc. It is now possible to accommodate entire databases in the main memory itself. This removes the need to reference the primary storage and results in upto 30 times faster performance. However, the software requirements for main-memory databases is still struggling to provide efficient designs and better algorithms to allow for optimization. In this paper, we have proposed the use of B-trees and red-black trees as the structure for main-memory database systems by demonstrating and analyzing the time taken for batch operations of insertion, deletion and updation. Since both of these are index data structures, they inherently provide faster performances compared to non-index structures.**

Index Terms—**index structures, main-memory database, performance analysis**

I. Introduction

IN recent years, the hardware industry has grown by leaps and bounds. Only five years ago, upto 1 terabyte of data storage space was considered to be more than enough for consumer purposes, the latest processors with two cores were considered to be top of the line, and hard disks were the pinnacle of storage technology. Now, however, terabytes of data are generated within weeks, the latest processors have upto 16 cores and the upcoming solid-state drive technology is rapidly replacing ordinary hard drives. Amidst these advances, the database systems industry is also making major breakthroughs. The most widely used database management systems, such as Oracle, MongoDB, MySQL, Microsoft Access, Firebird, Cassandra etc. are all disk-based systems. They have to reference the primary storage, i.e. hard drives, solid state drives etc. for all operations. However, primary storage references are slow, and to a large extent, have reached saturation point of optimization. Thus, the requirement of high-performance databases has led to the development of main-memory database systems.

This category of database systems makes it feasible to manipulate hundreds of megabytes or even several gigabytes of data, and store it entirely in volatile memory. Main memory references are much faster than primary storage references, and that has been the primary driving force behind this new system. Example applications include radio-frequency identification (RFID) readers to identify and track a large number of products, animals, vehicles, and even persons within a very short duration of time. Sometimes, the number of objects such as RFID tags per truck container, could be large, and the time duration, such as the passing time of a truck through a control gate, could be very short [1][2]. Many other applications also have the characteristics of high-rate data access, such as the process line of products manufacturing management [1][2][3], logistics management [4] goods retail management [2], and inventory management [2].

In this paper, we have talked about B-trees and red-black trees as potential designs for main-memory database systems. These are index structures, and while they involve additional writes take up additional space in memory, they inherently provide faster performances due to reduced look-up time. Researchers have explored different index structures such as the AVL tree, B^+-tree [5], T-tree [6][7], R-tree [19] and their derivatives [8]. B^+-trees [9][10][11][12], along with T-trees and AVL trees, would suffer from significant overheads in tree rebalancing when data are updated, inserted or deleted in a main-memory (and non-main-memory databases). Moreover, it has been observed that AVL trees are, in general, no better than T-trees in manipulation of data in short range even though each T-tree node could have more data than an AVL-tree node does (because each T-tree only has two pointers). While much research has been done on B^+-trees, T-trees and AVL trees, B-trees and red-black trees have remained relatively unexplored, and they are the focus of this paper. We have taken these two structures, and performed the operations of insertion, deletion and updation, in large batches. The time taken has been observed after fixed numbers of operations as well as after the completion of each operation, for both B-trees and red-back trees. Finally, these observations have been plotted on a graph and bar-chart, and the performance of these index structures for batch operations has been compared and analyzed.

II. Literature Review

Much research has been done on the topic of main-memory database systems, the use of index structures to design them as well as efficient algorithms for data storage and retrieval. Some of the problems tackled in such work, and the solutions proposed are described in this section.

A. Models and Issues in Data Stream Systems

B. Babcock et. al. (2002) suggested that it is not feasible to simply load the arriving dot into a traditional DBMS as they are not designed for rapid and continuous loading of individual dot items, and they do not directly support the continuous queries that are typical of dot stream applications [13]. Decision trees are one form of synopsis used for prediction. They propose a query language and architecture for a DSMS query processor designed specifically to address the issues above.

B. The Universal B-Tree for Multi-Dimensional Indexing: General Concepts

R. Bayer (1997) suggested that even though almost all database systems use B-trees as their main access method, one of the main drawbacks of the classical B-tree is that it works well only for one-dimensional data [14]. In the paper, he presents a new access structure, called UB-tree (for universal B-tree) for multidimensional data. The UB-tree is balanced and has all the guaranteed performance characteristics of B-trees, i.e. it requires linear space for storage and logarithmic time for the basic operations of INSERT, FIND and DELETE. In addition, the UB-tree has the fundamental property, that it preserves clustering of objects with respect to Cartesian distance. Therefore, the UB-tree shows its main strengths for multidimensional data. It has very high potential for parallel processing.

C. Organization and Maintenance of Large Ordered Indexes

C. McCreight et. al. (1972) discussed the problem of organizing and maintaining an index for a dynamically changing random access file [15]. They assume that the index itself is so voluminous that only rather small parts of it can be kept in main store at one time. Thus, the bulk of the index must be kept on some backup store. Since the data file itself changes, it must be possible not only to search the index and to retrieve elements, but also to delete and to insert keys--more accurately index elements--economically. They propose an index organization scheme in this paper which always allows retrieval, insertion, and deletion of keys in time proportional to log k I or better, where I is the size of the index, and k is a device dependent natural number which describes the page size such that the performance of the maintenance and retrieval scheme becomes near optimal.

D. Implementation Techniques for Main Memory Database Systems

D. J. DeWitt et. al. (1984) discussed the availability of very large, relatively inexpensive main memories, it is becoming possible keep large databases resident in main memory [16]. In this paper, the changes necessary to permit a relational database system to take advantage of large amounts of main memory are considered. AVL vs B^+-tree access methods for main memory databases, hash-based query processing strategies vs sort-merge, and study recovery issues are evaluated when most or all of the database fits in main memory. As expected, B^+-trees are the most efficient method of data retrieval.

E. The Ubiquitous B-Tree

D. Comer (1979) argued that despite the presence of large amount of secondary memory, a computer must retrieve an item and place it in main memory before it can be processed [5]. A sequential request requires the searcher to examine the entire file, one folder at a time. Associated with a large, randomly accessed file in a computer system is in index which, like the labels on the drawers and folders of the file cabinet, speeds retrieval by directing the searcher to the small part of the file containing the desired item. They aim to introduce a general-purpose file access method using internal storage structure called the binary search tree because of their guaranteed low retrieval cost.

F. Autonomic Tracing of Production Processes with Mobile and Agent-Based Computing

M.G.C.A. Cimino et. al. (2011) discussed about the simplification of tracing items in a supply chain, across different enterprises and through the full processes scope by implementing a supply chain traceability system with a high level of automation [17]. The proposed system adopts an agent-based approach, in which cooperative software agents find solutions to back-end tracing problems by self-organization.

G. Issues in Data Stream Management

M. T. Özsu et. al. (2003) discuses about Traditional databases store sets of relatively static records with no pre-defined notion of time, unless timestamp attributes are explicitly added. Although this model adequately represents commercial catalogues or repositories of personal information, many current and emerging applications require support for on-line analysis of rapidly changing data streams.

H. Real-Time Access Control and Reservation on B-Tree Indexed Data

Tei-Wei Kuo et. al. (2000) proposed methodologies to control the access of B^+-tree-indexed data in a batch and firm real-time fashion. He discussed methodologies to reduce the number of disk I/O to improve the system performance without introducing more priority inversion. When the schedulability of requests with critical timing constraints is highly important, he proposed a mechanism for data reservation based on the idea of preemption level and the Stack Resource Policy.

I. B-Tree Indexes for High Update Rates

G. Graefe (2006) surveys some techniques that let B-trees sustain very high update rates, up to multiple orders of magnitude higher than traditional B-trees, at the expense of query processing performance [18].

J. R-Trees: A Dynamic Index Structure for Spatial Searching

Antonin Guttman (1984) describes a dynamic index structure called an R-tree [19] in order to handle spatial data efficiently as required in computer aided design and geo-data applications, and give algorithms for searching and updating it.

III. Problem Definition

Main-memory database management systems are the future of bulk data storage and access However, as discussed earlier, the advances in software have not been able to keep up with the leaps and bounds in hardware technology. Thus, there is still a lot of ground to cover in terms of the design of main-memory database systems, as well as algorithms for faster and more efficient data management operations.

In this context, index data structures pose as the most obvious choice when it comes to the design of database systems. The various advantages and improvements in performance that they provide are discussed in the next section. While much research has been done on B^+-trees, AVL trees, T-trees, R-trees and their derivatives, B-trees and red-black trees remain relatively unexplored.

In this paper, we explore the possibility of using B-trees and red-black trees to design main-memory databases. Using Python scripts, we perform batch operations on B-trees and red-black trees and record the time taken for the operations. This time is then analyzed, and conclusions are drawn based on the same.

IV. Proposed Methodology

An index data structure is different from an ordinary (non-indexed) data structure in the fact that it will require less time for data retrieval operations due to the presence of a data index, an additional structure that is used to quickly locate data without having to search every row in the structure (tree, graph, database table, etc.) every time a query is made. However, this comes at the cost of additional writes and storage space to maintain the index structure. Indexes can be created using one or more parameters (e.g. multiple rows of the database table), providing the basis for both rapid random lookups and efficient access of ordered data.

In database management systems, a number of records can only be sorted based on one field. Thus, it can be state that searching on a field that isn't sorted requires a linear search which requires N/2 block accesses (on average), where N is the number of blocks that the table spans. If that field is a non-key field (i.e. doesn't contain unique entries) then the entire table space must be searched at N block accesses. On the other hand, with a sorted field, a binary search may be used as this has log2N block accesses. Also, since the data is sorted given a non-key field, the rest of the table doesn't need to be searched for duplicate values once a higher value is found. Thus, the performance increase is quite substantial.

Both B-trees and red-black trees are self-balancing tree structures. The self-balancing nature implies that after every operation, the tree will attempt to keep its height (maximal number of levels below the root) small to optimize insertions and deletions. Such structures can also be used for other abstract data structures such as associative arrays, priority queues and sets.

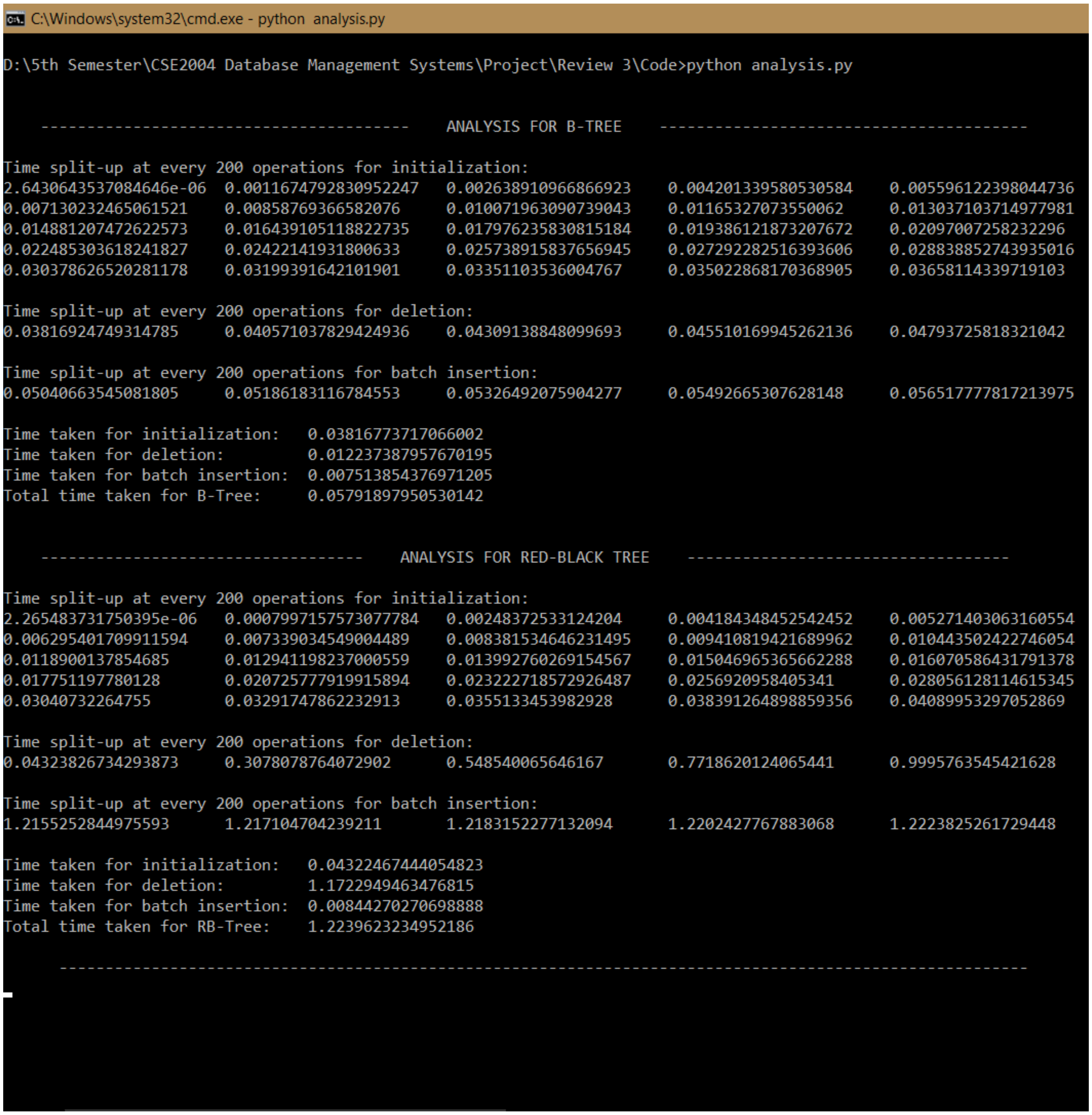
In a red-black tree, each node has an extra bit that is often interpreted as the colour of the node. Since there are only two colours, only one extra bit is needed. Moreover, since there is no more information pertaining to it being a red-black tree, its memory footprint is almost identical to a classic binary search tree. Balance is preserved by the maintenance of certain properties by the colours which collectively constrain how unbalanced the tree might get in the worst case. Even though the balancing of the trees is not perfect, it is good enough to allow the tree to perform all operations in O(log n) time.

Even though both the structures are self-balancing in nature, there is a vital difference between the type of data they are optimized for. A B-tree is a simple self-balancing tree structure, and is optimized for systems that read large blocks of data. On the other hand, a red-black tree is a binary search tree, meaning that apart from the self-balancing nature, the data is also arranged in order in the nodes of the tree. It is optimized for sparse operations of insertion and deletion.

Initially, we perform 5000 insertions to populate the database (initialization), followed by 1000 deletions and then 1000 insertions again. All of these are batch operations, i.e. they are done in bulk using Python scripts. For each type of tree, the time taken for each of the operations is recorded, and then added to calculate the total time for all operations. Next, a bar chart is plotted for time taken for each operation, for both tree types (the time for initialization is divided by 5 since there are 5000 initialization operations, but only 1000 deletions and insertions). This enables us to compare the performance of the two tree structures. Additionally, the time is recorded at every 200 operations, and a graph is plotted to observe and compare the progress of the operations, again for both tree types. All the operations have been done using Python, and graphs have been plotted using the matplotlib package in Python.

V. Results and Analysis

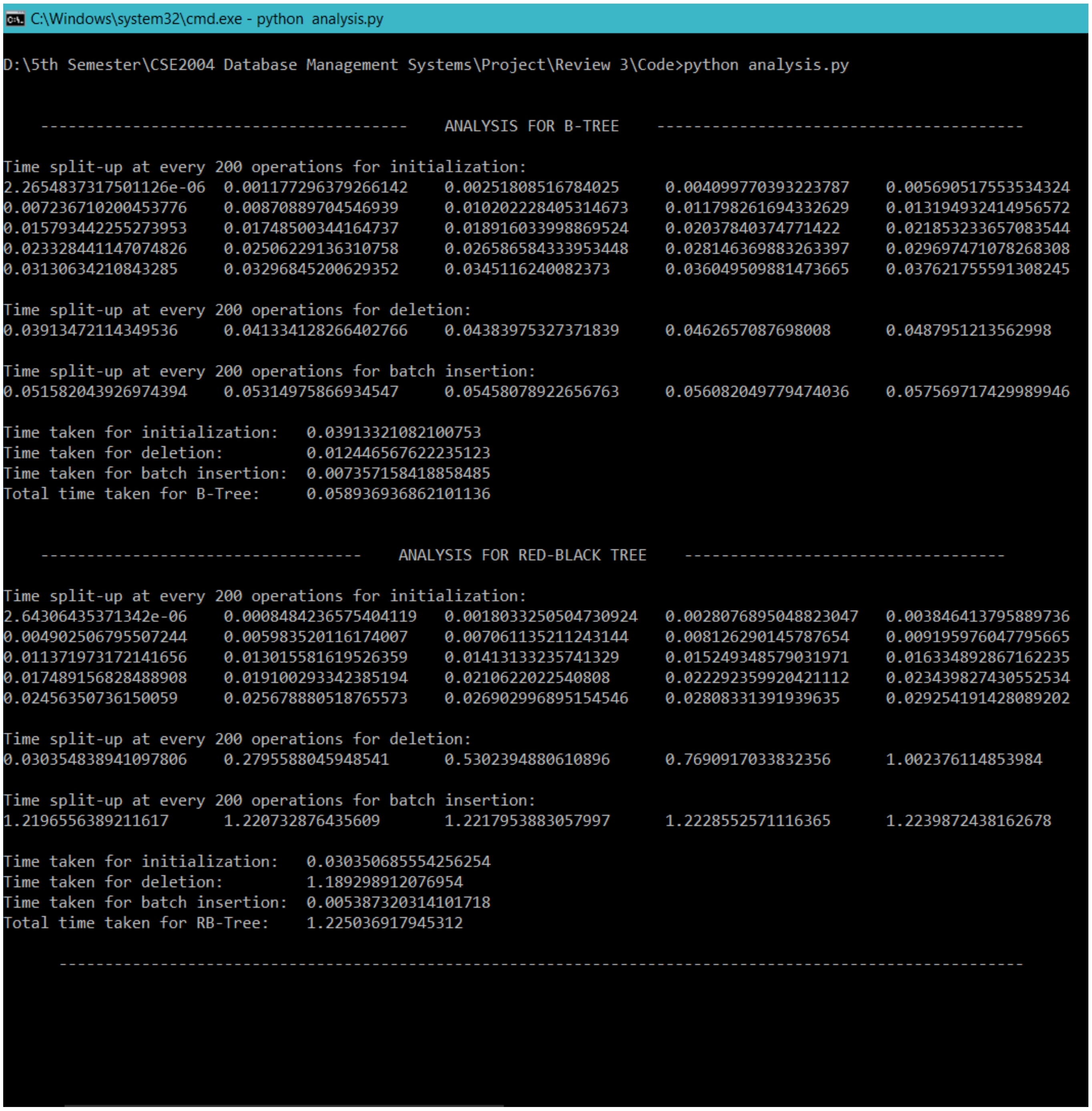
The B-Tree and red-black tree structures have been made to go through a batch insertion (initialization), deletion and a second insertion phase with the involvement of 5000, 1000 and 1000 records respectively. The experiments were simulated on a personal computer having Intel Core i5-7200U CPU @ 2.50GHz and 8GB RAM. Screenshots of the output obtained, along with the graphs and bar charts plotted, are described in this section.



```
C:\Windows\system32\cmd.exe - python  analysis.py

D:\5th Semester\CSE2004 Database Management Systems\Project\Review 3\Code>python analysis.py


    ----------------------------------------    ANALYSIS FOR B-TREE    ----------------------------------------

Time split-up at every 200 operations for initialization:
2.6430643537084646e-06  0.0011674792830952247   0.002638910966866923    0.004201339580530584    0.005596122398044736
0.007130232465061521    0.00858769366582076     0.010071963090739043    0.01165327073550062     0.013037103714977981
0.014881207472622573    0.016439105118822735    0.017976235830815184    0.019386121873207672    0.02097007258232296
0.022485303618241827    0.02422141931800633     0.025738915837656945    0.027292282516393606    0.028838852743935016
0.030378626520281178    0.03199391642101901     0.03351103536004767     0.035022868170368905    0.03658114339719103

Time split-up at every 200 operations for deletion:
0.03816924749314785     0.040571037829424936    0.04309138848099693     0.045510169945262136    0.04793725818321042

Time split-up at every 200 operations for batch insertion:
0.05040663545081805     0.05186183116784553     0.05326492075904277     0.05492665307628148     0.056517777817213975

Time taken for initialization:   0.03816773717066002
Time taken for deletion:         0.012237387957670195
Time taken for batch insertion:  0.007513854376971205
Total time taken for B-Tree:     0.05791897950530142


    -----------------------------------    ANALYSIS FOR RED-BLACK TREE    -----------------------------------

Time split-up at every 200 operations for initialization:
2.265483731750395e-06   0.0007997157573077784   0.00248372533124204     0.004184348452542452    0.005271403063160554
0.006295401709911594    0.007339034549004489    0.008381534646231495    0.009410819421689962    0.010443502422746054
0.0118900137854685      0.012941198237000559    0.013992760269154567    0.015046965365662288    0.016070586431791378
0.017751197780128       0.020725777919915894    0.023222718572926487    0.0256920958405341      0.028056128114615345
0.03040732264755        0.03291747862232913     0.0355133453982928      0.038391264898859356    0.04089953297052869

Time split-up at every 200 operations for deletion:
0.04323826734293873     0.3078078764072902      0.548540065646167       0.7718620124065441      0.9995763545421628

Time split-up at every 200 operations for batch insertion:
1.2155252844975593      1.217104704239211       1.2183152277132094      1.2202427767883068      1.2223825261729448

Time taken for initialization:   0.04322467444054823
Time taken for deletion:         1.1722949463476815
Time taken for batch insertion:  0.00844270270698888
Total time taken for RB-Tree:    1.2239623234952186

      ------------------------------------------------------------------------------------------------------
```

Fig. 1. Total and split-up of time taken for initialization, deletion and insertion in B-Tree and RB-Tree.

Fig. 1 shows an instance of the time recorded for batch initialization, deletion and insertion in B-Tree and red-black tree. The time has also been recorded at intervals of every 200 operations for plotting of the graph.



```
C:\Windows\system32\cmd.exe - python  analysis.py

D:\5th Semester\CSE2004 Database Management Systems\Project\Review 3\Code>python analysis.py


    ----------------------------------------    ANALYSIS FOR B-TREE    ----------------------------------------

Time split-up at every 200 operations for initialization:
2.2654837317501126e-06  0.001177296379266142    0.00251808516784025     0.004099770393223787    0.005690517553534324
0.007236710200453776    0.00870889704546939     0.010202228405314673    0.011798261694332629    0.013194932414956572
0.015793442255273953    0.01748500344164737     0.018916033998869524    0.02037840374771422     0.021853233657083544
0.023328441147074826    0.02506229136310758     0.026586584333953448    0.028146369883263397    0.029697471078268308
0.03130634210843285     0.03296845200629352     0.0345116240082373      0.036049509881473665    0.037621755591308245

Time split-up at every 200 operations for deletion:
0.03913472114349536     0.041334128266402766    0.04383975327371839     0.0462657087698008      0.0487951213562998

Time split-up at every 200 operations for batch insertion:
0.051582043926974394    0.05314975866934547     0.05458078922656763     0.056082049779474036    0.057569717429989946

Time taken for initialization:   0.03913321082100753
Time taken for deletion:         0.012446567622235123
Time taken for batch insertion:  0.007357158418858485
Total time taken for B-Tree:     0.058936936862101136


    -----------------------------------    ANALYSIS FOR RED-BLACK TREE    -----------------------------------

Time split-up at every 200 operations for initialization:
2.64306435371342e-06    0.0008484236575404119   0.0018033250504730924   0.0028076895048823047   0.003846413795889736
0.004902506795507244    0.005983520116174007    0.007061135211243144    0.008126290145787654    0.009195976047795665
0.011371973172141656    0.013015581619526359    0.01413133235741329     0.015249348579031971    0.016334892867162235
0.017489156828488908    0.019100293342385194    0.0210622022540808      0.022292359920421112    0.023439827430552534
0.02456350736150059     0.025678880518765573    0.026902996895154546    0.02808331391939635     0.029254191428089202

Time split-up at every 200 operations for deletion:
0.030354838941097806    0.2795588045948541      0.5302394880610896      0.7690917033832356      1.002376114853984

Time split-up at every 200 operations for batch insertion:
1.2196556389211617      1.220732876435609       1.2217953883057997      1.2228552571116365      1.2239872438162678

Time taken for initialization:   0.030350685554256254
Time taken for deletion:         1.189298912076954
Time taken for batch insertion:  0.005387320314101718
Total time taken for RB-Tree:    1.225036917945312

      ----------------------------------------------------------------------------------------------------
```

Fig. 2. Total and split-up of time taken for initialization, deletion and insertion in B-Tree and RB-Tree.

Fig. 2 shows another instance of the time recorded for batch initialization, deletion and insertion in B-Tree and red-black tree. Again, the time has also been recorded at intervals of every 200 operations. When compared to Fig. 1, it is observed that the recorded times are consistent in both the instances.

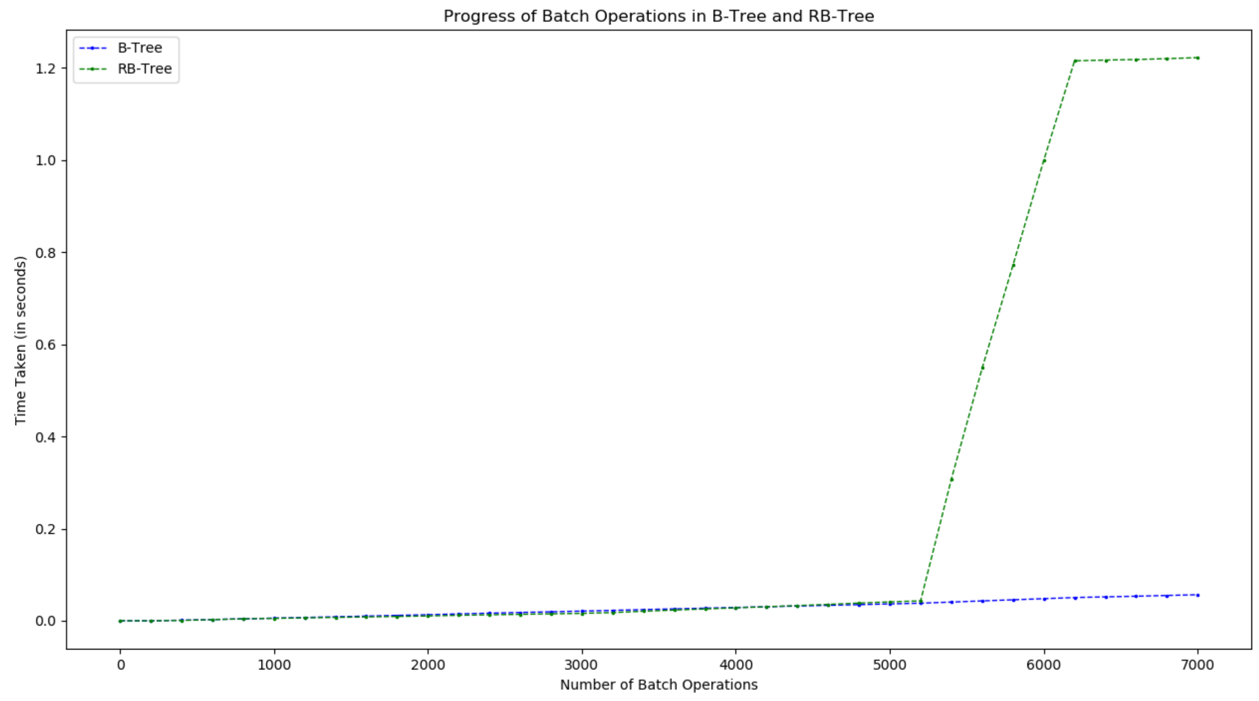


Fig. 3. Progress of batch operations (5000 insertions, 1000 deletions and 1000 insertions again) in B-Tree and red-black tree.

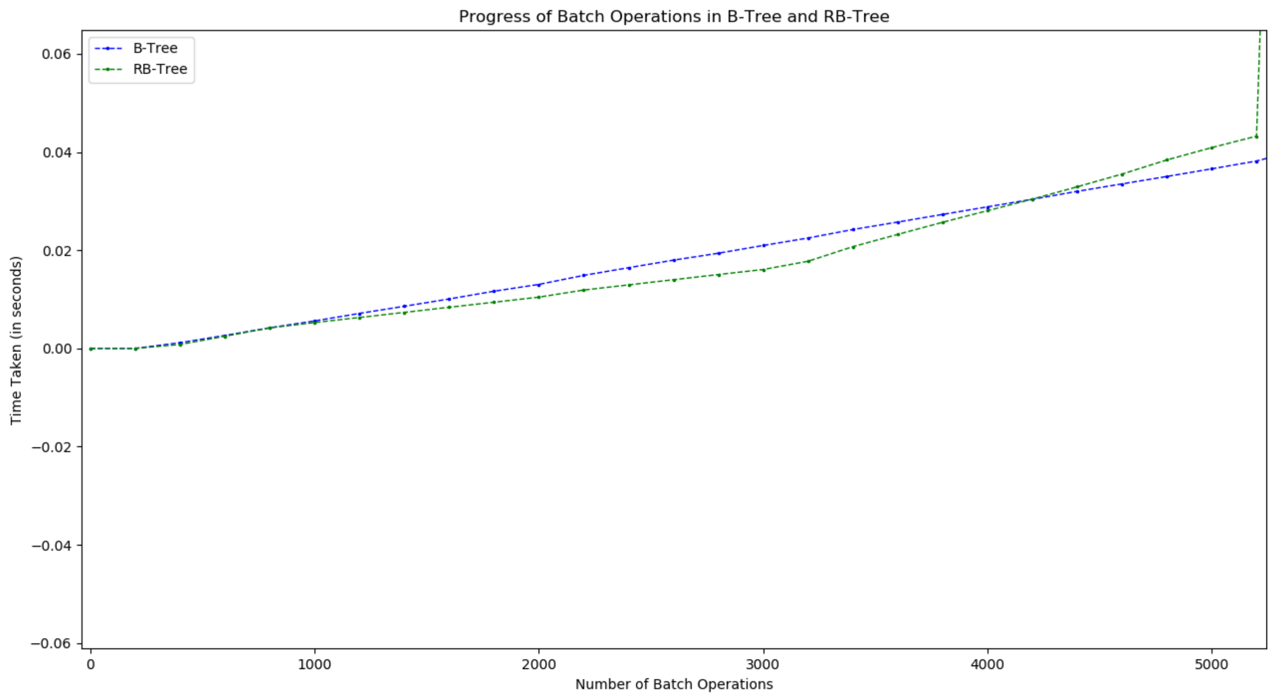


Fig. 4. Progress of 5000 insertions (initialization) in B-Tree and red-black tree, recorded at every 200 operations.

From Fig. 3 and Fig. 4, it is observed that the time taken for batch insertion operations is approximately the same for both B-Tree and red-black tree. Moreover, for B-Tree, insertion and deletion operations take same amounts of time, because of which the graph for B-Tree is almost a straight line.

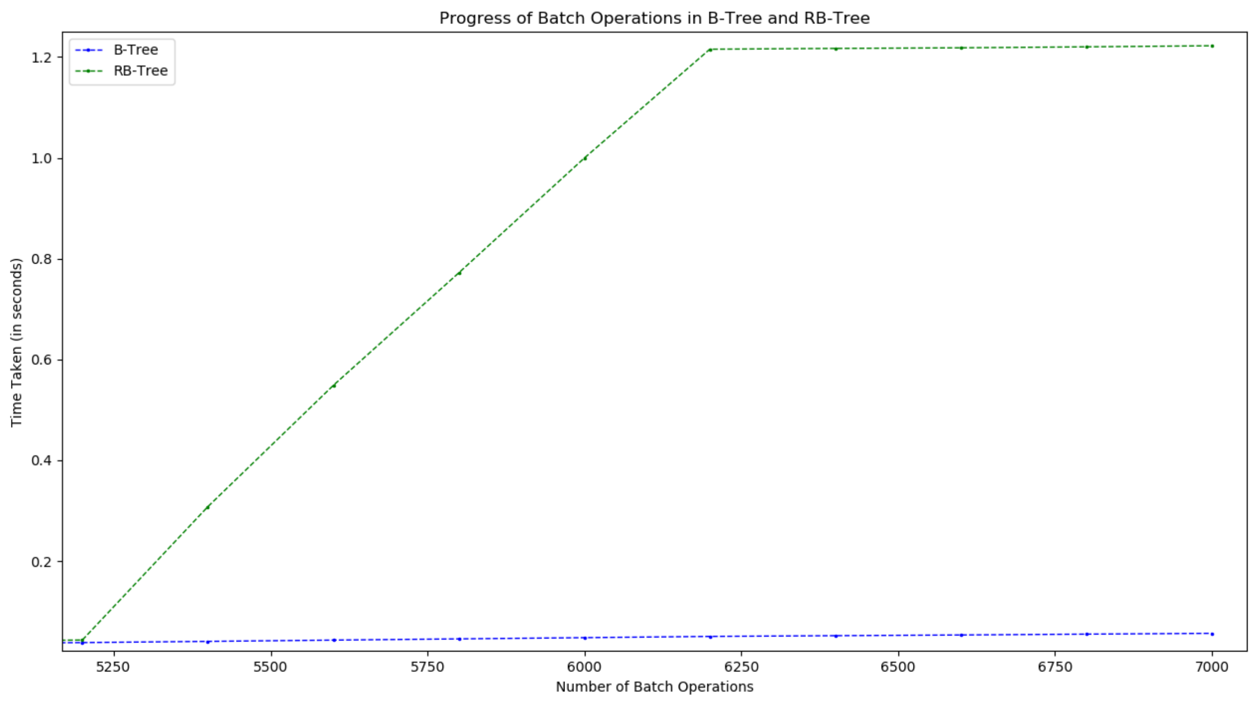


Fig. 5. Progress of 1000 deletions and 1000 insertions in B-Tree and red-black tree, recorded at every 200 operations.

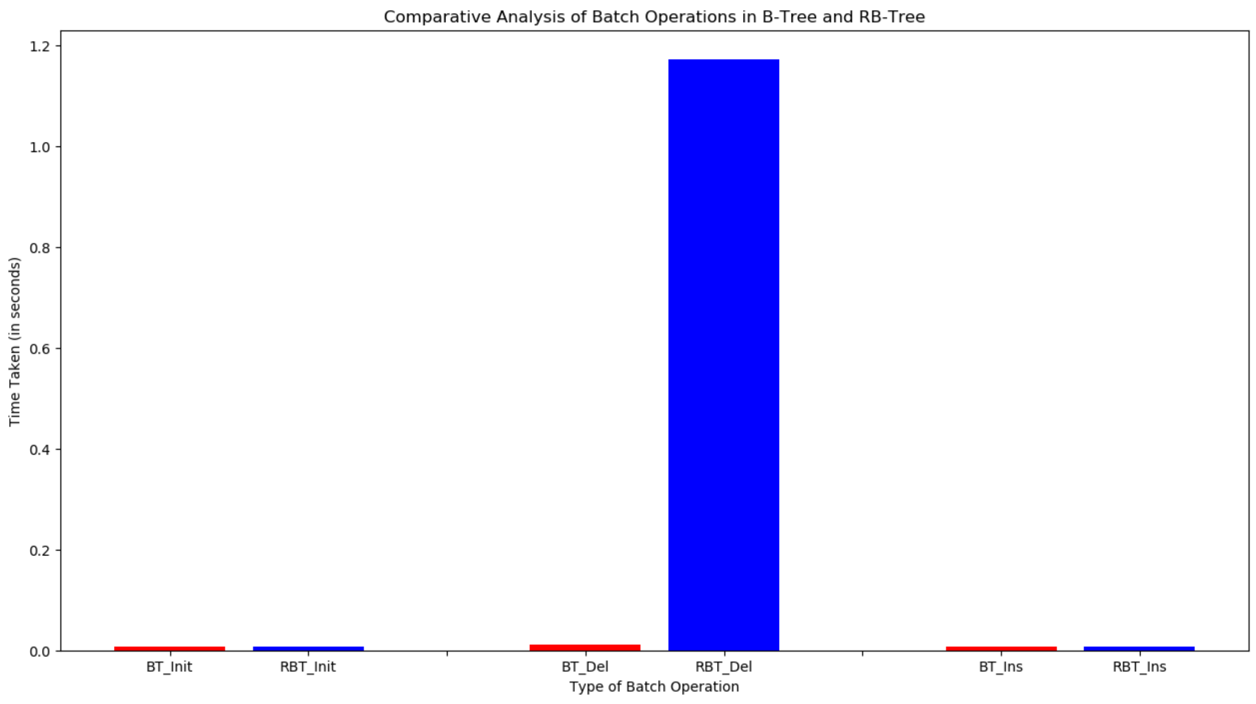


Fig. 6. Comparison of time taken for 1000 operations each of insertion (initialization), deletion and again insertion.

From Fig. 5, it is observed that while batch deletion in B-Tree takes approximately the same time as batch insertion, it is 98 times slower in red-black trees. Fig. 6 shows compares the total time taken for batch insertion and deletion in B-Tree and red-black tree structures.

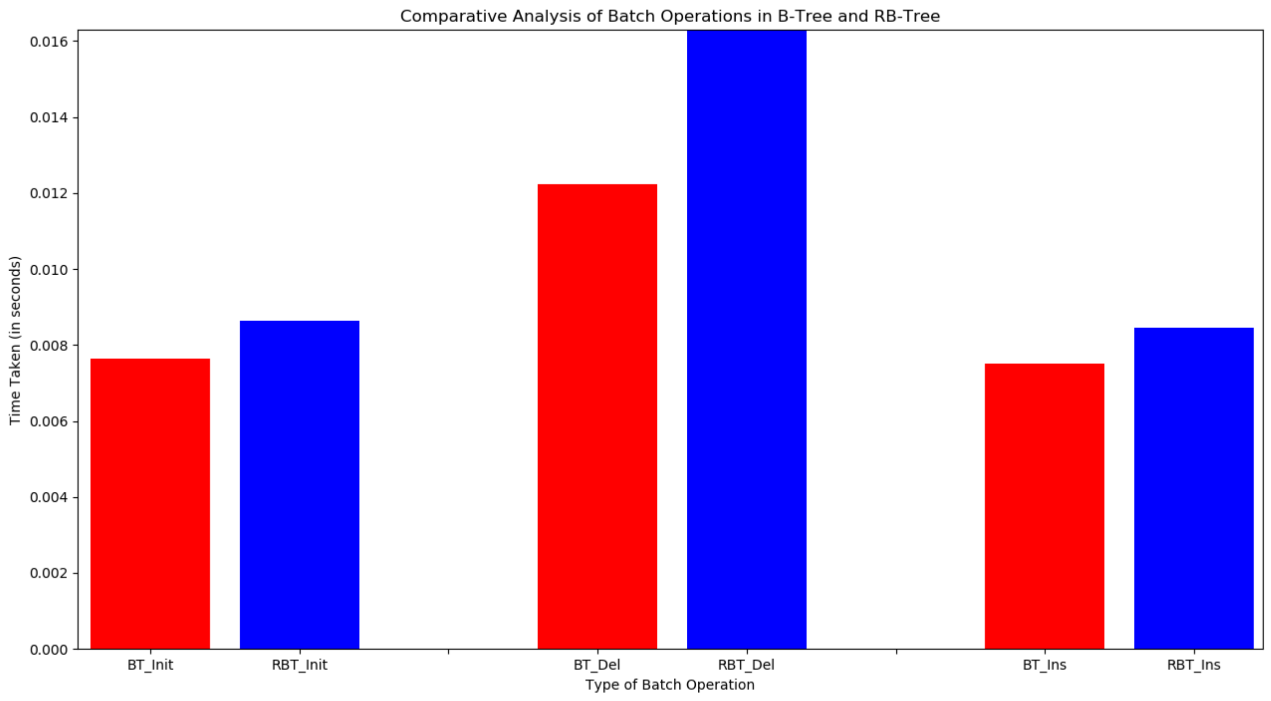


Fig. 7. Detailed comparison of time taken for 1000 operations each of insertion (initialization), deletion and again insertion.

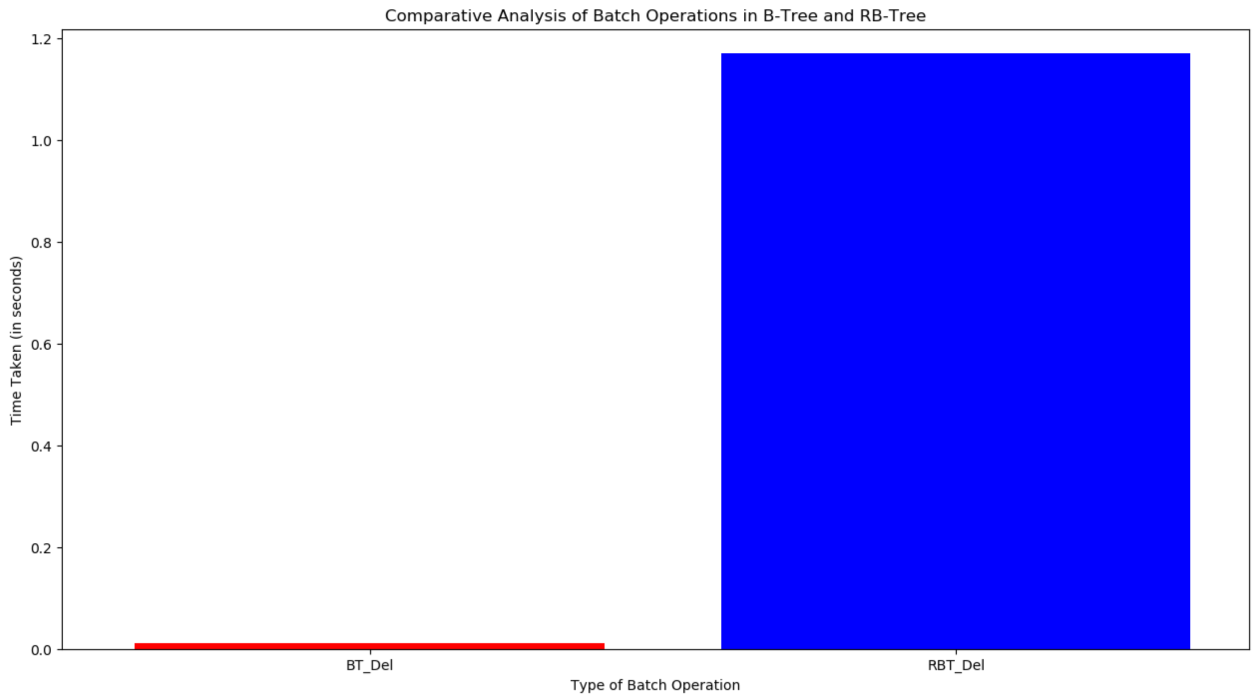


Fig. 8. Comparison of difference in time taken for 1000 deletion operations in B-Tree and red-black tree.

Fig. 9 shows a more detailed comparison of the time taken for batch insertion and deletion in B-Tree and red-black trees. While insertion takes slightly more time in red-black trees, the time for deletion extends far beyond the figure. Finally, Fig. 8 gives a true indication of the huge difference in batch deletion time between B-trees and red-black trees.

VI. Conclusion

This work has been motivated by a strong demand for main-memory database system design. B-Trees and red-black trees have been proposed to design the databases firstly, due to their inherently improved performance because of their nature as an index data structure, and secondly because of their self-balancing nature which provides additional optimization for data retrieval. After obtaining the results, it was observed that B-Trees performed slightly better at batch insertion and distinctly better at batch deletion than red-black trees. In fact, in the latter operation, they performed more than 98 times better than red-black trees. The results were highly conclusive in the direction that B-Trees are optimized for large blocks of data, while red-black trees are more suited for applications with sparse data manipulation, or at least where batch deletions are avoided wherever possible.

VII. Future Scope

There is a growing need for efficient main-memory database design, in applications including but not limited to process line of products manufacturing management, logistics management goods retail management, inventory management etc. This work can be carried forward to incorporate more data structures, and finally come to a conclusion on the choice of data structure depending on the requirements and applications of the database management software. Moreover, cutting-edge algorithms can also be developed to couple these structures to fulfill high-performance requirements.

Acknowledgment

This work has been supported by Dr. Swathi J. N., Associate Professor (SCOPE), VIT University, Vellore, and the authors are thankful to her for her constant motivation, guidance and involvement in this work. The authors are also thankful to the developers of third party tools and software (Python, including the time, numpy and matplotlib packages etc.), which were extensively used in this work.

References

[1] B.N. Lee, Y.-W. Kim, H.J. Kim, Evolution of RFID applications and its implications-standardization perspective, in: Proceedings of Portland International Center for Management of Engineering and Technology, 2007, pp. 903–910.

[2] D.-L. Wu, W.W.Y. Ng, D.S. Yeung, H.-L. Ding, A brief survey on current RFID applications, in: Proceedings of the International Conference on Machine Learning and Cybernetics, Baoding, 2009, pp. 2330–2335.

[3] M.G.C.A. Cimino, F. Marcelloni, Autonomic tracing of production processes with mobile and agent-based computing, Information Sciences 181 (2011), pp. 935–953.

[4] J. Li, Q.S. Jia, X. Guan, X. Chen, Tracking a moving object via a sensor network with a partial information broadcasting scheme, Information Sciences 181 (2011) 4733–4753.

[5] D. Comer, The ubiquitous B-tree, ACM Computing Surveys 11 (1979) 121–137.

[6] T.J. Lehman, M.J. Carey, A study of index structures for main memory database management systems, in: Proceedings of the Twelfth International Conference on Very Large Data Bases, 1986, pp. 294–303.

[7] H. Lu, Y.Y. Ng, Z. Tian, T-tree or b-tree: main memory database index structure revisited, in: Proceedings of the Eleventh Australasian Database Conference, 2000, pp. 65–73.

[8] K.R. Choi, K.C. Kim, T*-tree: a main memory database index structure for real time applications, in: Proceeding of the Third International Workshop on Real-Time Computing Systems and Applications, 1996, pp. 81–88.

[9] J.V.D. Bercken, B. Seeger, An evaluation of generic bulk loading techniques, in: Proceedings of the 27th International Conference on Very Large Data Bases, 2001, pp. 461–470.

[10] G. Graefe, B-tree indexes for high update rates, ACM SIGMOD Record 35 (2006) 39–44.

[11] P.-M. Kerttu, S.-S. Eljas, Y. Tatu, Concurrency control in B-trees with batch updates, IEEE Transactions on Knowledge and Data Engineering 8 (1996) 975–984.

[12] T.W. Kuo, C.H. Wei, K.Y. Lam, Real-time access control and reservation on B-tree indexed data, Journal of Real-Time Systems 19 (2000) 245–281.

[13] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in: Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, 2002, pp. 1–16.

[14] R. Bayer, The universal B-tree for multi-dimensional indexing: general concepts, in: Proceedings of the International Conference on World-Wide Computing and Its Applications, Lecture Notes on Computer Science, Springer-Verlag, 1997, pp. 198–209.

[15] R. Bayer, C. McCreight, Organization and maintenance of large ordered indexes, Acta Informatica 1 (1972) 173–189.

[16] D.J. DeWitt, R.H. Katz, F. Olken, L.D. Shapiro, M.R. Stonebraker, D. Wood, Implementation techniques for main memory database systems, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 1984, pp. 1–8.

[17] Mario G. C. A. Cimino, Francesco Marcelloni, Autonomic tracing of production processes with mobile and agent-based computing, Information Sciences 181, 2011, pp. 935–953.

[18] Goetz Graefe, B-Tree indexes for high update rates, ACM SIGMOD Record 35, Issue 1, March 2006, pp. 39-44.

[19] Antonin Guttman, R-Trees: A dynamic index structure for spatial searching, in Proceeding, SIGMOD '84 Proceedings of the 1984 ACM SIGMOD international conference on Management of data, 1984, pp. 47-57.