International Journal of Computer Applications (0975 – 8887)
Volume 109 – No. 4, January 2015

Efficient Dynamic Index Structure for Natural Number Intensive Application

Mayank Patel
Department of Computer Science
Gujarat University, Ahmedabad, India

Bhavesh Parmar
Department of Computer Science
Gujarat University, Ahmedabad, India

Yatrik Patel
INFLIBNET Centre, Gandhinagar, India

Hiren Joshi
Dr. Babasaheb Ambedkar Open University, Ahmedabad, India

ABSTRACT

Wide range of indexing techniques exists in the world of relational database. Speed of data insertion & retrieval depends on the type of query and available Indexing mechanism. Prevalent mechanisms lack in terms of space-time efficiency and simple structure, for real time applications where the database system needs to handle queries like equality search & range search. Even for simple tasks like getting data by ID, a system imposes heavy resource utilization. For example, Applications such as, telephone directory, transaction information details in banking, status about railway reservation etc., backed with relational database system that employs complex structure like B-Tree or B$^+$-Tree. Hence in such cases, instead of those complex structures, if some lighter technique can be used, which can greatly enhance the overall performance in terms of memory usage and simpler in terms of working & implementation. The paper presents how the Proposed Technique can significantly impact the overall performance, if applied as Primary Indexing method for range search & equality search queries.

General Terms

Tree Structure, Indexing, Management in Data Structures, Trie Tree, B-Tree, B$^+$-Tree, Algorithms

Keywords

Natural numbers, Dynamic index structure, Indexing, Database management system.

1. INTRODUCTION

A Database is defined as organized collection of logically related data items [1]. A Relational Database is a database that represents data as a collection of tables, wherein all data relationships are represented by common values, in related tables [1]. An Index is a data structure used by DBMS that organizes data records on disk such that it optimizes certain kinds of data retrieval operations. With the help of an Index, a Database System can achieve efficient retrieval of those records, which satisfy search conditions on the search key fields of the Index [2]. An Index greatly reduces the searching overhead for DBMS where the query refers to only tiny portion of records in a file. DBMS uses diverse mechanisms, starting from sequential scanning to indexing; from hashing to combination of indexing; hashing to some other complex techniques to quickly fetch data [1, 3].

Indexes for Relational Database are broadly classified into Ordered Index & Hashed Index. Ordered Index is further classified into clustered index & non-clustered index. In clustered index the search key defines an order which is sequential whereas in non-clustered index the search key defines an order which is different than clustered order. Clustered index stores data physically in the dame order as index whereas in non-clustered a separate list is maintained that points to the actually stored data [2, 3].

In Hashed Index approach, the records in a file are grouped in buckets. Here this bucket consists of a primary page and possibly additional pages linked in a chain. The bucket, to which a record belongs, can be determined by applying a special function to the search key. This special function is called a hash function. Given a bucket number, a hash-based index structure allows us to retrieve the primary page for the bucket in one or two disk I/Os. On inserts, the record is inserted into the appropriate bucket & once the bucket is full, an 'overflow' pages are allocated as needed. To search for a record with a given search key value, the same hash function that was used for insertion, is applied to identify the bucket to which the required records reside. If the search key value for the record is not known then scanning of all pages in the file would be required [2].

Database system employs multiple engines and multiple Indexing techniques. One of many parts in database engine is query optimization; the task of it to take decision, which ultimately results in to minimum time of locating the requested data [4, 5]. For example, B-Tree and R-Tree, two different structures are employed by MyISAM engine for Indexing [6]. Now, Hashing i.e. Hashed based Index is used by DBMS for equality comparisons that use the '=' or '! =' operators only. This type of Index is not suitable for comparison operators like '>', '>=' or '<', '<=', which finds values in a range. MySQL documentation notes that it becomes difficult for MySQL to determine how many rows exists between two values, for range search [4].

Consideration of wider range of applications by the authors resulted into a deprivation of method for those applications that exhibit certain behaviour. It was observed that, a simple & efficient structure was missing for application, where the application most of the time need to deal with fetching of information by matching and comparing with natural numbers. The problem with traditional & prevalent methods was that they were quite complex in terms of their working and hence resulted in to big memory footprint & unnecessary usage of system resource. To name a few such applications, telephone directory, bank transactions, customer order tracking in e-commerce etc., were in this category.

Graefe, Goetz, and Harumi Kuno [4] notes that the basic design has not been changed much even after 40 year's effort in optimization. As a result their implementation continues to exhibit the same structure overhead when applied to the applications the authors just described. The hashing technique i.e. Hashed Index, imposes a constraint of load factor which limits the performance of hashing beyond certain pick value. In addition, enormous amount of hashing in case of collision, to find the right bucket to map the key to be retrieved or to be placed, is also the limiting factor in hashing technique [7-20].

A new structure has been introduced in this paper, which is a kind of tree structure but not the tree structure. In that there is header which maintains crucial information for structure. Out of 4 cells in the header, the 1st & 2nd cells include information about Start ID & Last ID respectively. The 3rd cell maintain information about Root Address of Indexing structure while the 4th cell about digit Length of Last ID number. Starting from, below the header to the last level, in the whole tree, there are arrays of length 10. Whilst the only last level contain arrays of length 11. This last level arrays points to actual data on the disk.

2. BACKGROUND

R. Bayer and E. McCreight, at Boeing Scientific Research Labs, proposed an external index mechanism with relatively low cost for most of the operations and called it a B-tree [1, 2]. According to the property of B-Tree, it contains variable number of children at each node. Important thing to note in this tree is that all of the child nodes have data contained within them.

Another variant of B-Tree is B^+-Tree. It stores data at leaf level only & the internal nodes contain only pointers to those data [1, 2, 5]. It is widely used in many relational database systems for metadata indexing. B*- tree is yet another variant of B-Tree. It reduces the space utilization by densely packing the internal nodes [5].

R. Bayer yet presented one more variant for multidimensional data indexing [19]. It is same as B^+- Tree except that records are stored according to Z-order or also called Morton order. However the algorithm exhibits exponential behaviour, for range search in multidimensional point data.

Bumbulis, Peter offered one more revised form of B-Tree called compact B-Tree [20]. It compact the tree by using the free space of siblings before overflow occurs in the node. This mechanism significantly reduces total amount of space utilized by reducing the no. of split, no. of nodes required.

Graefe, Goetz, and Harumi Kuno did survey on modern B-Tree technique & concluded that the core design of B-trees has remained unchanged in 40 years. This includes balanced trees, pages or other units of I/O as nodes, efficient root-to-leaf search, splitting and merging of nodes, etc. [4]. Many improvements have been done in every aspect like, multi-dimensional data, algorithms for accessing, for example, multi-dimensional queries. Improvement in data organization within each node is also another one, for example, compression & cache optimization [4].

Hashing is also used in indexing task by database system. In hashing, the data is mapped in to memory, called buckets, according to mathematical function defined. And to retrieve it, the same mathematical function is used which generates the same location of bucket where the data was previously mapped to [1, 2, 5]. Though hash table can offer rapid insertion, deletion, and search of both strings and integers, it requires a form of collision resolution to resolve cases where two or more keys are hashed to the same bucket. To resolve this, various mechanisms have been proposed like, linked lists [7] – used when number of keys is not known in advance, array hash [8] – a cache conscious scheme for previous method, open addressing – stores homogenous keys directly within bucket & gives better usage of CPU & cache [9, 10]. Open addressing schemes: Linear probing, where the interval between probes is fixed [18]; quadratic probing [12] where probe interval is increased by addition of successive outputs of a polynomial to the starting value; and double hashing [12] where probe interval is computed by second hash function. Previous three techniques still faces collisions.

Coming to relatively new concept regarding collision resolution, cuckoo hashing [13], is another open-addressing solution where it maintains two hash tables & two hash functions. When collision take place it moves the data around alternative bucket. Various schemes & mechanism have proposed to enhance the performance & reduce the space, like cuckoo hashing with pages – where each key has several possible locations, or cells, on a single page, and additional choices on a second backup page [14], with improved, insertion [15], look up [16] & setting upper bound for construction time in this technique [17].

Nikolas Askitis did comparison of bucketized cuckoo hashing (a new scheme to address collision, which employs several hash function & hash bucket that can store more than one key) and found out that despite a constant worst-case probe cost, it was consistently slower than the array hash to build, search, and delete keys with a skew distribution. The bucketized cuckoo hash table could only withstand with the performance of the array hash under heavy load and when there is no skew in the data distribution [18].

Important thing to note here was that it still has constraint of collision, load factor, distribution.

Edward Fredkin, gave a technique called Trie tree. Here the core part is that the alphabets or digits are stored by taking each digit at a time & checking where they differ. According to the position where they differ the alphabet or digit is stored at that place [22]. However, Trie tree [22] has space/time trade-off of managing child nodes i.e. either a potentially large and sparse array of indexes at each node would be needed or implementation of a secondary search algorithm to find the appropriate child would be needed.

Patricia tree [22], a space optimized Trie tree at small size; if it is well-designed it is comparable & sometimes faster than hash maps/balanced trees. However, due to the effect of pre-determining what the branching criterion is, it does lack performance.

Stefan Björnson proposed a new technique called "Management in Data Structure"; which was simplification to a tree called Trie. The main intention was to simplify the management of data storage for relatively simple task like storing digits & alphabets [23].

3. PROPOSED SYSTEM

This section addresses the logical structure of proposed technique, with in-depth discussion of how the technique works. The structure carries following property.

1. The Root node maintains information about first ID, last ID of the record inserted, root address of indexing structure plus length of last ID inserted.
2. Total no. of levels beneath the root node would be the maximum length of natural no.
3. Structure dynamically adds an array of length 10, as required in each level.
4. Intermediate level arrays contain addresses of child arrays & the last level arrays contain addresses of actual data.
5. Each last level array would contain an extra cell which is used to connect each of the last level arrays.

Following figures with example-description could give better idea of the property & their working.

Basic structure is shown in Figure 1. At the top there is a header which maintains information of Start ID, Last ID of the records to be indexed. The third information cell is the root address of the first array in the hierarchy. At the last, fourth cell contains information of maximum length of the last ID. Beneath the header structure there would be an array of fixed length 10; plus 1 extra cell for leaf level arrays. Leaf level arrays stores address of the actual data. The total number of levels in the structures would be equal to the length of the largest number in the ID column. So, for example, if ID=7685493 then total no. of levels except the header would be 7(the length of ID number).

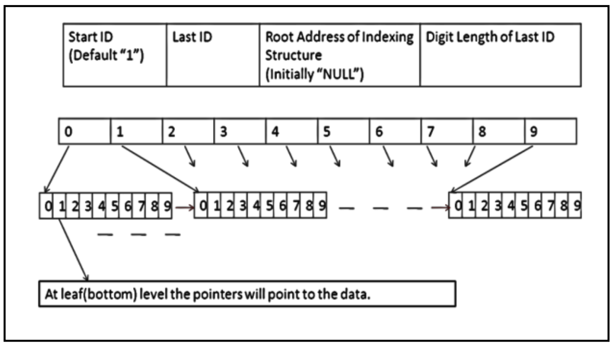
Start ID (Default "1")
Last ID
Root Address of Indexing Structure (Initially "NULL")
Digit Length of Last ID
At leaf(bottom) level the pointers will point to the data.

Fig 1: Basic structure

Next, various operations are explained that needs to be performed on the structure.

3.1 Insertion

Initial values in header structure before insertion of first record would be as shown in Figure 2. Here Start ID & Last ID = 0 means no records. Root Address is NULL means no record & so there isn't any root.

Note: It should be noted that in all of the figures, all of the arrays beneath the header, having numbering given from 0 to 9, is for understanding purpose only. So it should not be confused with oblivion that there is array where the first row is numbered with 0 to 9 & in the second row it would have corresponding addresses. There would be only one array & it itself would contain address of the next array beneath it (if it is intermediate array) or pointer to the actual data (if it is the last level array in respective tracing of the ID).

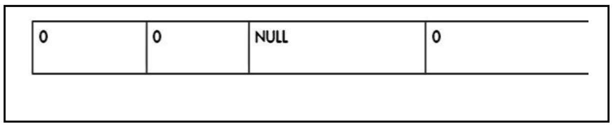

Fig 2: Empty structure (Initial values)

3.1.1 Inserting record with ID=1

As shown in Figure 3 when first record is inserted with ID=1, the values in header structure of Start ID & Last ID would be replaced with 1 & the third cell would contain the root address of the Indexing structure. Now, the first cell of newly added array would have null, as it doesn't point to any further Index structure beneath it. The second cell in the array would contain address of the data i.e. record associated with that ID. Subsequent records with ID=2 to ID=9 would be inserted accordingly in the same manner as described for 1.

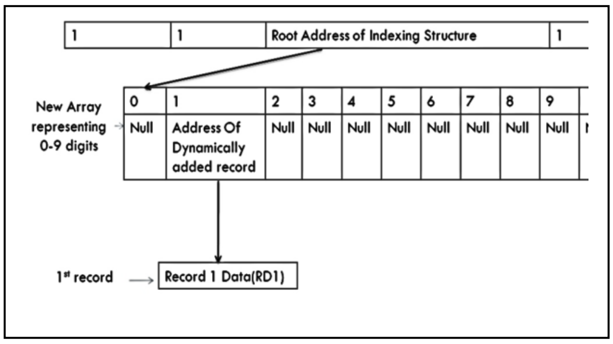
1
1
Root Address of Indexing Structure
1
New Array representing 0-9 digits
Null
Address Of Dynamically added record
1st record
Record 1 Data(RD1)

Fig 3: Insertion of 1st record with ID=1

3.1.2 Inserting record with ID=10

Since at the time of insertion of record with ID=10 the 1st array would be full. A new array of fixed length 10 would be inserted beneath the header structure & above the first array in the first level of the structure. Figure 4 illustrates the situation.

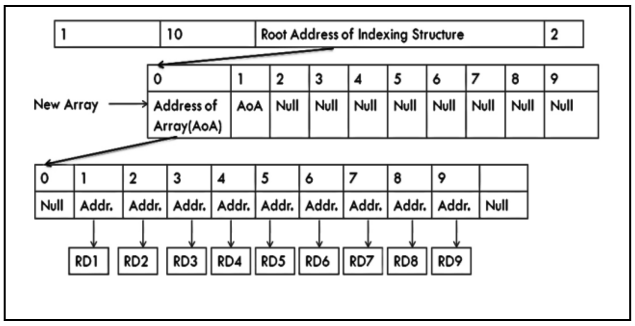

Fig 4: New array inserted beneath header structure & above first array of first level, in the structure

Now, as shown in Figure 4 the first cell of newly inserted array would contain address of the first cell of second level (which was first level before new array was inserted above it). This way splitting is avoided. Now a new array in 2nd level would point to the record data of ID 10 as shown in Figure 5.

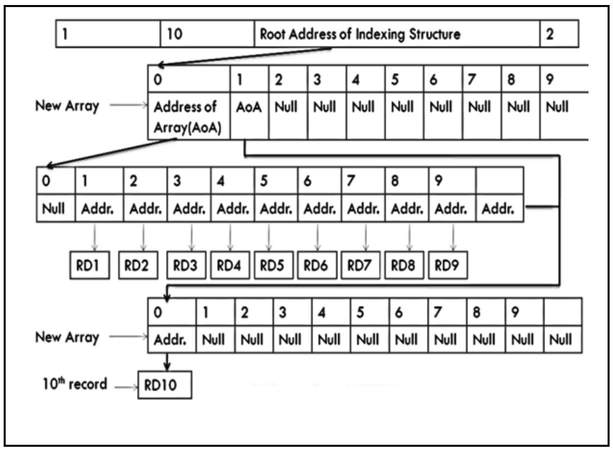

Fig 5: Record with ID=10 inserted

3.2 Lookup (Searching)

In lookup procedure first it would be checked whether there exists a record with that particular ID by comparing it with the first two values in header structure, Start ID & Last ID. If not then lookup procedure would be terminated from there itself otherwise it would be continued to next steps. In next step the digit starting from 1st to last position i.e. up to length of the ID would be traversed accordingly & finally the last position would end up further traversing to actual data.

3.2.1 Equality search - Searching record with ID=16

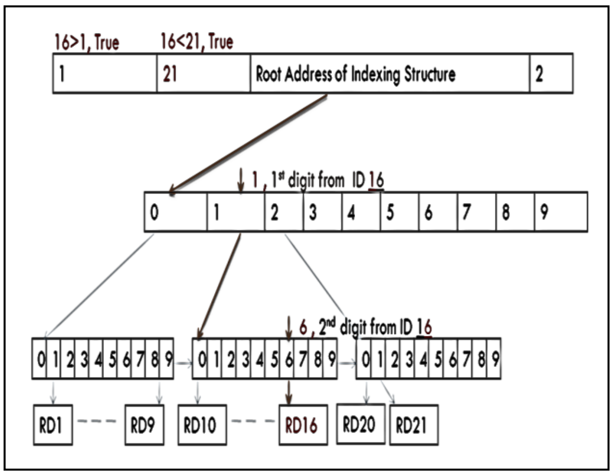

Fig 6: Searching record with ID=16

Refer Figure 6 that illustrates the searching procedure for ID-16. The searching flow would execute like this. First it would be checked whether ID=16 lies between the range of the total records inserted or not. It certainly does, so now digit at the first position is taken i.e. “1”. The 3rd cell in the header structure contain address of the 1st array at first level, so using that it reaches to the 1st location in that array & plus 1 the pointer (i.e., add one to the address) to go to the second cell of that array. After reaching 2nd cell in the first level array, address of the 2nd level array is known. Now 2nd digit of the ID is taken i.e. 6, so the position is added with 6 to reach at the 7th location in the second level array, which contain address of data having ID=16.

3.2.2 Range search – Searching all records i.e. from ID=1 to ID=21

In range search queries, all records between specified ID ranges would be retrieved. Here the 11th cell kept for connecting all last level arrays comes into picture. This 11th cell would be used to quickly jump to next record in the sequence of leaf level arrays.

Take for example, if all records between ID rang 1 to 21 needs to be fetched. This would be accomplished as follows.

After checking validity of the ID range, the record with first ID in the range is reached, by following the same procedure as discussed in “Search” part. Figure 7 depicts the situation. Once at leaf level, records would be fetched one by one. Once reached to the end of the concerned array i.e. the last cell, the 11th cell would be used to know the next array’s address & again all records in the ID range would be fetched. The procedure repeated until the last ID is reached in the range query.

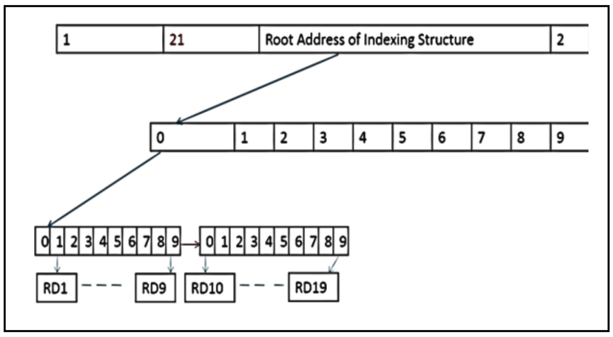

Fig 7: 11th cell in the leaf level arrays is used to get next address of array in the sequence

3.3 Update

The updating of record involves the same procedure of locating the record with the given ID as illustrated in above lookup procedure. After reaching at the specified ID the record is simply updated.

3.4 Deletion

In deletion there are two cases.

1. Deletion of single record.
2. Deletion of record in specified range

3.4.1 Deletion of single record

Here the same procedure is used described in ‘equality search’ to reach to the particular record to be deleted. After reaching to the record to be deleted; the record is deleted & NULL is set to indicate that it no longer points to any data.

3.4.2 Deletion of records in specified range

In this case, procedure described previously for searching records with specified ID range, is used. And then records within specified range are deleted.

In both cases, the spaces occupied by unused arrays can be freed when they no longer points to any data.

Figure 8 shows advantage of the proposed structure. Here the unused arrays are de-allocated & thus space is saved, as records between ID=10 to ID=19 were deleted.

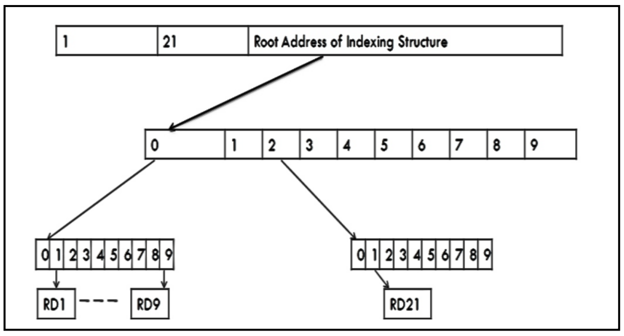

Fig 8: De-allocated unused array pointing to deleted records 10-19.

4. EXPERIMENT SETUP

In the experiment more emphasis has been given to the range search query performance evaluation. Testing & comparison is done of the proposed technique with B-tree and B^+-Tree.

Measurement of Insertion time and Searching time (in seconds) of all these method against Proposed Technique is done. Space requirement is also measured in KBs (Kilobytes).

In B^+-Tree, data inputs were set of natural numbers generated by computer program i.e., each index entry stored a corresponding number. For Proposed Technique authors provided natural numbers to be stored by the structure at the end-leaf level.

Calculation of Insertion time for each technique is done by summing up the time taken by insertion module. Starting from inserting first record to the last record with predefined benchmark; ranging from 1 Million to 15 Million. Similarly Search time is total of getting all the inserted records one by one from Index.

To measure Space, inbuilt system tool provided by the Windows and LINUX OS is used.

Microsoft Visual Studio 2010 IDE is used to run & test all three techniques. The machine was Intel Core 2 Quad Q6600 @ 2.40 GHz running 32-bit Windows 7 Ultimate and Fedora 20 LINUX having single user. The machine had 2GB of generic DDR2 666 RAM with 4MB of L3 cache and L1, L2 cache being 32 KB.

One limiting factor regarding testing was that as the technique has been proposed for database, it was required to integrate the proposed technique to at least one of available Relational Database System.

Unfortunately because of time limitation & very lengthy process of achieving that environment, the testing criterion has been bounded to comparing the all three methods at programming levels only. As a part of profiling, optimized codes of B-Tree & B^+-Tree method found on web are compiled to get the best results [26-28].

5. RESULT AND ANALYSYS

In this section various test result outputs with graphs & an analysis of the proposed technique with others has been given.

Note: Proposed Technique & B^+-Tree stores key-value (Actual Data), where B-Tree stored only key; & this exaggerated performance of B-Tree.

5.1 Windows Environment

5.1.1 *Time complexity analysis*

5.1.1.1 *Insertion time*

The graph in Figure 9 shows comparison of Insertion Time for Proposed Technique with other techniques i.e. B-Tree & B^+-Tree in Windows environment. It can be seen that Proposed Technique takes moderate time compared to other techniques.

5.1.1.2 *Search time*

Equality search

Test Results for SQL queries like “where ID=1” to “where ID=15M” (Million = 10^6).

Figure 10 shows a graph of Search Time comparison (in seconds) of records starting from 1M to 7M. Here Proposed Technique takes more time than other two but what happens when records increase beyond 7 million?

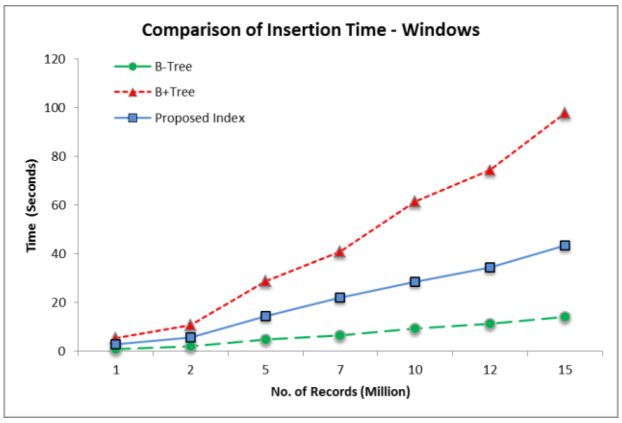

Fig 9: Insertion time comparison-Windows

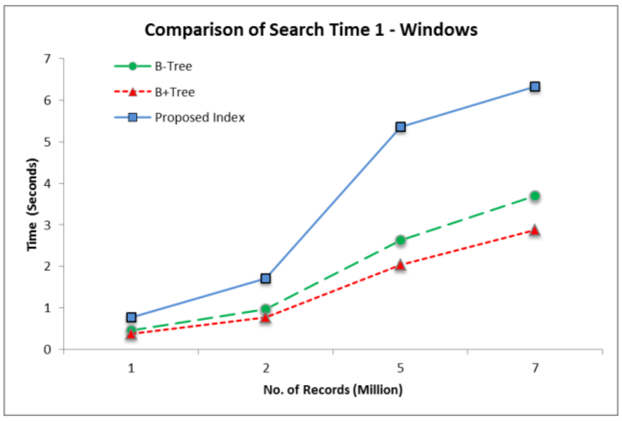

Fig 10: Search time comparison 1-Windows

In Figure 11, it can be seen that Proposed Technique takes almost same time as taken by B-Tree.

However the performance of B^+-Tree is degraded. Because of consumption of more space by B^+-Tree led to memory overflow and hence system experienced page faults.

Range search

Test Results for SQL Queries like “where ID>=1 and ID<=1M” to “where ID>=1 and ID<=15M”.

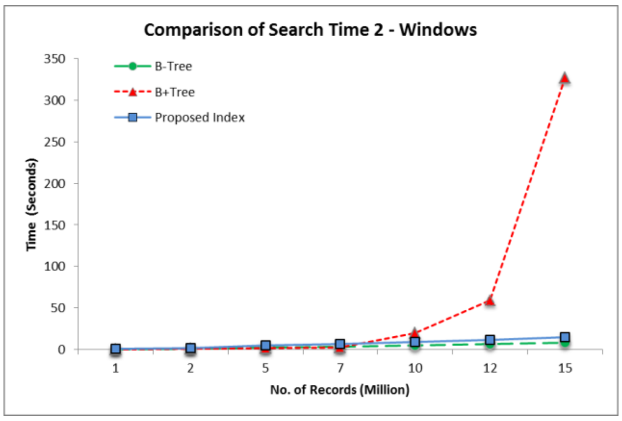

Fig 11: Search time comparison 2-Windows

In case of B-Tree, Range Search is not as simple as the other two.

For testing purpose ‘get all records’ (Traverse function) is used to obtain Range Search Time.

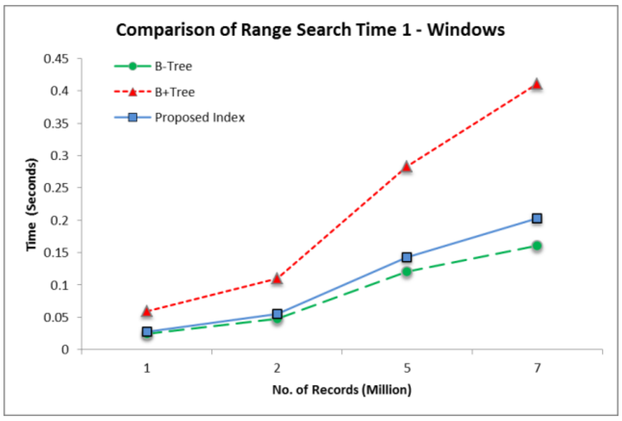

Fig 12: Comparison of Search Time for Range queries 1-Windows

Figure 12 shows a graph of Comparison of Range Search Time of records starting from 1M to 7M.

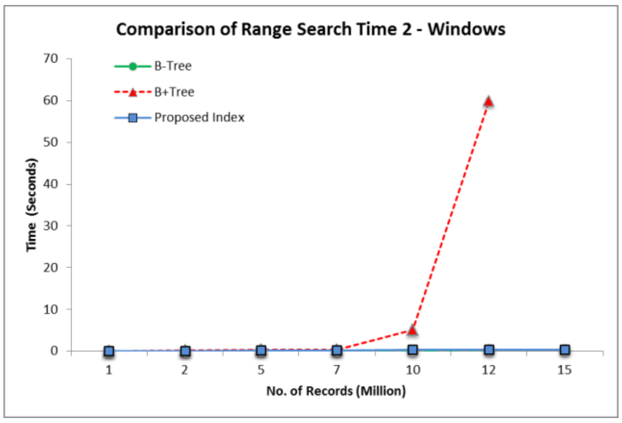

Fig 13: Comparison of Search Time for Range queries 2-Windows

Graph in Figure 13 shows that proposed technique operates near to B-Tree. Here, records up to 15 million have been considered for Range Search.

It can be observed that, in case of B^+-Tree performance is decreased just like it did in Equality Search Time scenario due to more memory requirement.

5.1.2 Space complexity analysis

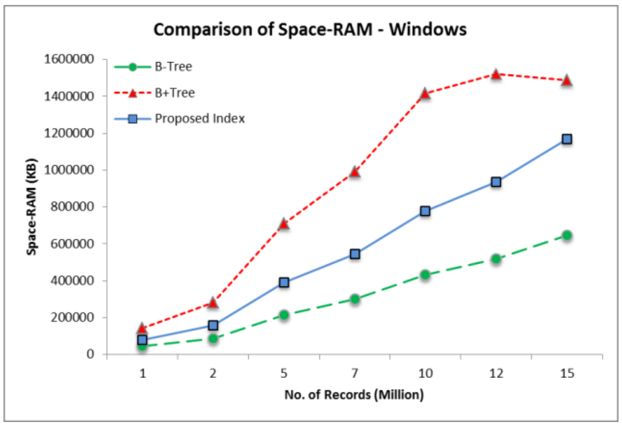

Fig 14: Comparison of space requirement - Windows

Figure 14 shows a graphical view of space utilization during Equality Search & Range Search by all the techniques. Proposed Technique takes moderate & nearly constant space then other two techniques.

5.2 Linux environment

5.2.1 Time complexity analysis

5.2.1.1 Insertion time

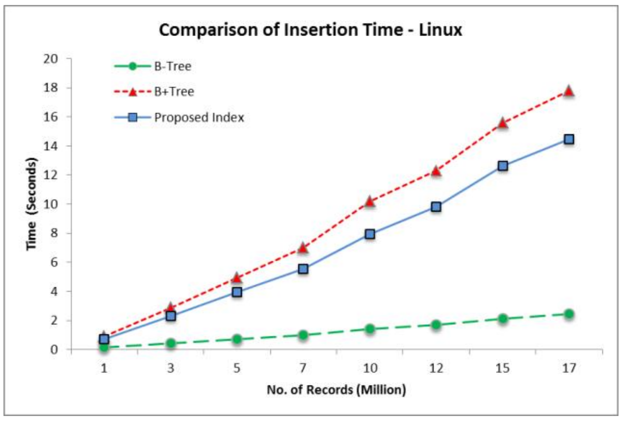

Fig 15: Insertion time comparison -Linux

Figure 15 shows graphical representation of test data collected for Insertion Time in Linux environment.

It can be observed that Proposed Technique operates near to B^+-Tree.

5.2.1.2 Search time

Equality search

Figure 16 shows a graphically view of test data collected for Search Time in Linux environment for Equality Search query.

Here Proposed Technique takes more time than others.

Note: Here the tool used was not able to compute the records more than the capacity of RAM, so as with Windows, it was not possible to evaluate performance of B^+-Tree for more than 20M records.

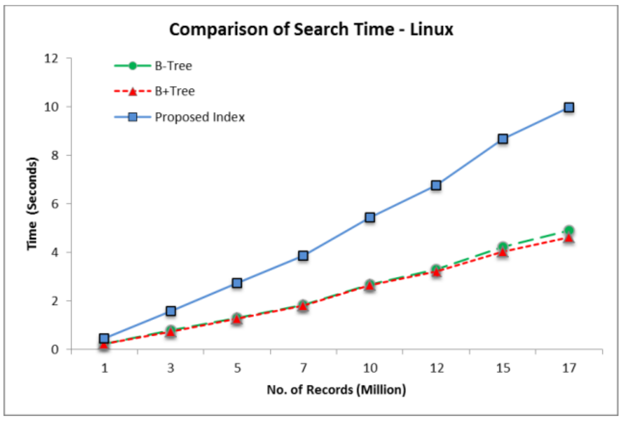

Fig 16: Search Time comparison for Equality Search query-Linux

Range search

Figure 17 shows graphical view of test results for Range Search query in Linux environment.

For Range Search, Proposed Technique performs better than B-Tree & B^+-Tree.

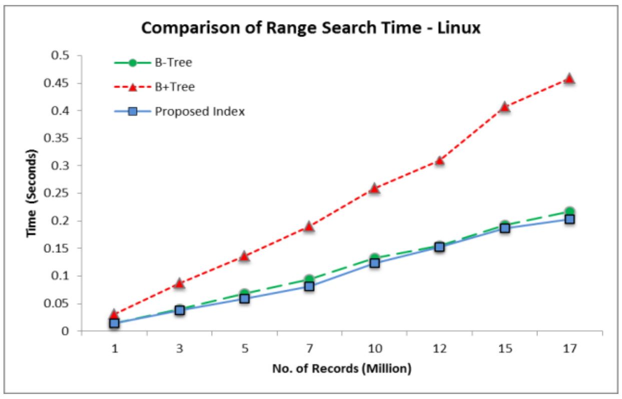

Fig 17: Search Time comparison for Range Search query-Linux

5.2.2 *Space complexity analysis*

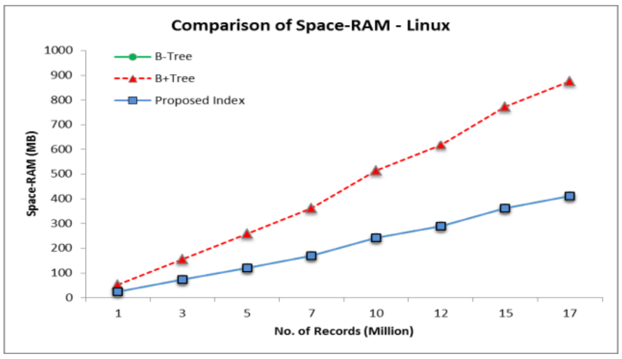

Fig 18: Space utilization comparison-Linux

Figure 18 shows graphical view of space utilization. It can be seen that Proposed Technique and B-Tree lines overlap each other. Thus it proves that authors were able to achieve the "All data at leaf level" by using same amount of space as B-Tree & saved ~50% space as compared to B$^+$-Tree.

5.2.3 *Cache miss analysis*

Cache miss is a failed attempt to read a piece of data in the cache.

There are three kinds of cache misses: Instruction Read miss, Data Read miss and Data Write miss.

5.2.3.1 *Insertion, equality & range search cache miss*

Figure 19 shows column view of the statistics collected for cache miss during all 3 operations being performed on the Proposed Technique.

With Cachegrind tools in Linux; cache misses during Insertion, Equality Search and Range Search queries by all the three candidate techniques was measured for this experiment.

In 1st graph, it can be seen that B$^+$-Tree experience the highest no. of cache misses i.e. Data write miss in First Level & Last Level of cache. On the other hand B-Tree & Proposed Technique operates moderately compared to B$^+$-Tree.

In 2nd graph, it can be seen that Proposed Technique outperform the other two technique – works best for Equality Search query, by experiencing lowest no. of cache misses in reading data.

In 3rd graph, it can be observed that B-Tree & Proposed Technique experience almost same no. of cache miss in data writes at First & Last level. On the other hand data read miss at both the level is moderate.

Note: There were two reasons why B-Tree performed better. First reason was that, the code used for B-Tree [27, 28] while testing was just implementation of its algorithm. It was not pointing to any record data like B$^+$-Tree [26] and Proposed Index techniques did. This led to performance improvement of B-Tree as there was no overhead of managing pointers for actual data in every operation.
Secondly, one study of worst case and average case behaviors of B-trees concludes that "adding periodic rebuilding of the tree, . . . the data structure . . . is theoretically superior to standard B$^+$-trees in many ways [and]. . . rebalancing on deletion can be considered harmful"[29].

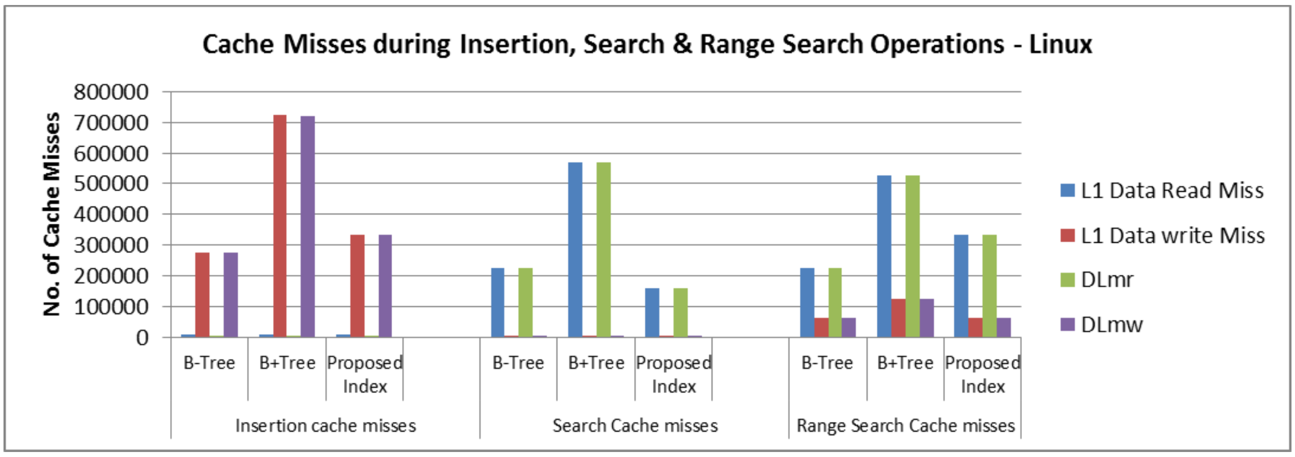

Fig 19: Cache misses during Insertion, Search & Range Search Operations-Linux

6. CONCLUSION

The most used database indexing methods in prevalent database systems have been discussed. Considering the applications which involve immense processing of data, based on natural numbers; ranging from bank transaction to railway ticket reservation, telephone number directory to order tracking in e-commerce, authors found that due to inherent complexity of structure & management in the existing methods the database system experiences performance degradation. The authors were able to achieve better performance in new technique because the total numbers of levels in the index structure are equal to the length of the natural number i.e. ID digits. Further the proposed technique structure does not involve any splitting, so it results in very less overhead compared to other two methods B-Tree & B$^+$-Tree. The results of the experiment carried out clearly states that proposed technique gives best performance in Linux environment for Range Search. Beside this, it also offers less memory footprint compared to other candidates of experiment. Even further it outperformed others in case of number of cache misses, during the most vital operation of any Database System 'Search'. In all, the Proposed Technique gives advantages in terms of speed, space & management point of view compared to the most used database indexing methods like B-Tree & B$^+$-Tree.

With minor changes Proposed Technique has potential to replace B$^+$-Tree & B-Tree for most of their applications. I.e. for string data, date time data, random numbers etc.

7. ACKNOWLEDGMENTS

We are grateful to our Guide Dr. Hiren Joshi for giving us the opportunity of working under his guidance for research in the area of Database Indexing. He has shared his valuable knowledge in each and every step for this ambitious work throughout. His time-to-time suggestion and knowledge imparted in the domain of Analytics was the back bone for the work presented. Our deepest thank to our Guide Mr. Yatrik Patel for the knowledge he delivered in the field of database and also for the valuable time which he spent for guiding in further enhancement in the research work. His guidance has always inculcated confidence in us.

8. REFERENCES

[1] Ramakrishnan, R., and J. Gehrke. "Database Management Systems." (2003)..

[2] Silberschatz, Abraham, H. Korth, and S. Sudarshan. "Database System Concepts", 2010.

[3] Pachev, Alexander, and Sasha Pachev. *Understanding MySQL Internals.* " O'Reilly Media, Inc.", 2007.

[4] Graefe, Goetz, and Harumi Kuno. "Modern B-tree techniques." In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pp. 1370-1373. IEEE, 2011.

[5] Knuth, D. E. "The art of computer programming: Sorting and Searching, 2nd edn., vol. 3." (1998).

[6] Ashok Rathi, Huizhu Lu, G.E. Hedrick, "Performance Comparison Of Extendible Hashing And Linear Hashing Techniques", 1990

[7] Zobel, Justin, Steffen Heinz, and Hugh E. Williams. "In-memory hash tables for accumulating text vocabularies." *Information Processing Letters* 80, no. 6 (2001): 271-277.

[8] Askitis, Nikolas, and Justin Zobel. "Cache-conscious collision resolution in string hash tables." In *String Processing and Information Retrieval*, pp. 91-102. Springer Berlin Heidelberg, 2005.

[9] Heileman, Gregory L., and Wenbin Luo. "How Caching Affects Hashing." In *ALENEX/ANALCO*, pp. 141-154. 2005.

[10] Peterson, W. W. (1957), 'Open addressing', *IBM Journalof Research and Development* **1**(2), 130–146.

[11] Askitis, Nikolas. "Efficient data structures for cache architectures." PhD diss., PhD thesis, RMIT University. RMIT Technical Report TR-08-5. http://www. cs. rmit. edu. au/naskitis, 2007.

[12] Stein, Clifford, T. Cormen, R. Rivest, and C. Leiserson. "Introduction to algorithms." *The MIT Press* 31 (2001): 77.

[13] Pagh, Rasmus, and Flemming Friche Rodler. "Cuckoo hashing." *Journal of Algorithms* 51, no. 2 (2004): 122-144.

[14] Dietzfelbinger, Martin, Michael Mitzenmacher, and Michael Rink. "Cuckoo hashing with pages." In *Algorithms–ESA 2011*, pp. 615-627. Springer Berlin Heidelberg, 2011.

[15] Fountoulakis, Nikolaos, Konstantinos Panagiotou, and Angelika Steger. "On the insertion time of cuckoo hashing." *SIAM Journal on Computing* 42, no. 6 (2013): 2156-2181.

[16] Jang, Joonhyouk, Yookun Cho, Jinman Jung, and Gwangil Jeon. "Enhancing lookup performance of key-value stores using cuckoo hashing." In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, pp. 487-489. ACM, 2013.

[17] Drmota, Michael, and Reinhard Kutzelnigg. "A precise analysis of cuckoo hashing." *ACM Transactions on Algorithms (TALG)* 8, no. 2 (2012): 11.

[18] Askitis, Nikolas. "Fast and compact hash tables for integer keys." In *Proceedings of the Thirty-Second Australasian Conference on Computer Science-Volume 91*, pp. 113-122. Australian Computer Society, Inc., 2009.

[19] Bayer, Rudolf. "The universal B-tree for multidimensional indexing: General concepts." In *Worldwide Computing and Its Applications*, pp. 198-209. Springer Berlin Heidelberg, 1997.

[20] Bumbulis, Peter. "System and methodology for providing compact B-Tree." U.S. Patent 6,694,323, issued February 17, 2004.

[21] Grant Fritchey and Sajal Dam, "SQL Server 2008 Query Performance Tuning Distilled", Apress 2009.

[22] Sartaj Sahani,Dinesh P Mehta,"Tries" in Handbook of datastructures & Applications, Chapman & Hall/CRC, US 2005.

[23] Stefan Björnson, "Management in data structures", EP1040430 B1, July 3 2009.

[24] "Oracle® Database, Performance Tuning Guide," 12c Release 1 (12.1), accessed February 20, 2014. http://docs.oracle.com/cd/E16655_01/server.121/e15857/title.htm

[25] "MySQL 5.6 Reference Manual", Accessed February 22, 2014. https://dev.mysql.com/doc/refman/5.5/en/optimization-indexes.html

[26] B$^+$-Tree code, version 1.12, http://www.amittai.com/

[27] B-Tree code, http://www.geeksforgeeks.org/b-tree-set-1-insert-2/

[28] Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest.

[29] S. Sen and R. E. Tarjan, "Deletion without rebalancing in multiway search trees," ISAAC, pp. 832–841, 2009