NoTree

Iliya Tronkov[1]

Abstract – **The present article introduces an external merge based index data structure called NoTree. What is distinctive for it is that it does not work with individual keys, but with intervals of keys. NoTree inherits the message model from buffer based trees, but goes one step forward as it allows selecting and specific multiway merging of buffers from different levels. This gives a possibility for choosing an optimal path of buffers which can be merged with minimum number of I/O operations. For comparison of the different families of indexing technologies, a unified approach is applied (an interval model) which highlights the relation between them. The studies include B+-tree, Buffer Trees, LSM-tree, Fractal Index Tree, WaterfallTree, NoTree and the accent is not on the number of comparisons that are made between the keys, but on the number of I/O operations. The NoTree index is suitable for applications with intensive flow of input data which is constantly been queried with range queries that include a significant subset of the available data in the index.**

Keywords – **Database, Benchmark, Index Data Structure, Indexing, Comparative Analysis.**

I. INTRODUCTION

The keys from each leaf of the B-tree [1-3] based trees define an *interval* which is [*minimal key from leaf, maximal key from leaf*]. The so defined intervals do not have common points, i.e. they do not intersect two by two. (Without losing generality we consider that all keys are unique). We will call such set of intervals a *level of intervals*. Therefore B-tree based trees have only one level of intervals formed from their leaves (Fig. 1). The keys that are located in the internal nodes of the tree are not part of the user data and are excluded from our considerations.

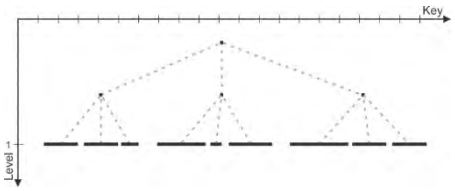

Fig. 1. Interval model of B-tree based data structure - has one level of intervals.

When random keys are entering the B-tree they end up in different intervals, i.e. different blocks will have to be accessed from the storage device. But if not all records fit into the memory, we will have to do an I/O operation to the external storage for only one record. This drastically decreases the speed of indexing. In order to reduce the negative effect from the random keys, buffers with messages are added in the internal nodes of the tree. The aim is to move messages not one by one but in groups with every I/O operation.

[1]Iliya Tronkov is with the Faculty of Computer Systems and Technologies at Technical University of Gabrovo, Bulgaria, E-mail: tronkov@gmail.com.

In buffered based trees (Buffer Tree [4-5], FractalIndex Tree [6], WaterfallTree [7-9]) we have several levels of intervals. In that case the leaves once again form one level of intervals. But the internal nodes also contain user data in the form of messages. Following the logic as in the leaves, the keys (of the messages) from every internal node define an interval. Therefore the buffer based trees have as many levels of intervals as their height, with the additional requirement for *subordination of the intervals* - every interval can intersect only with the intervals of its subtree and every level can have at most $(number\ of\ branches)^{level\ number\ -\ 1}$ intervals (Fig. 2):

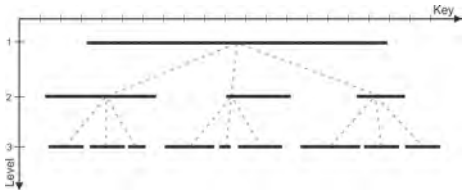

Fig. 2. Interval model of buffer tree-based data structure - has the same levels of intervals as the height of the tree with the additional subordination of the intervals.

Thanks to the buffers in buffer based trees the number of I/O operations is reduced, but in the meantime, if the speed with which the records fill a given buffer is faster than the speed in which the buffer is emptied, it becomes a tight spot for its whole subtree. This is mostly valid for the buffer in the root of the tree, because all data goes through it.

If we release ourselves from the restricting requirement of subordination of the intervals, we will derive to the idea of LSM-tree [10] (which is implemented in several systems BigTable [11], LevelDB, Apache HBase, RocksDB, etc.).

The structure of the LSM-tree is a combination of an index tree with messages into the memory (usually an AVL-tree) and several (usually two) B-tree indices on the storage device. (Most of the LSM-tree implementations have more than two levels on the disk device.) Thus, from the interval model point of view, in LSM-tree we have one level of intervals, which is defined by the AVL-tree and by a single level of intervals for every B-tree index. Every next level contains more or equal number of intervals from the previous level (Fig. 3). The logical partitioning of the AVL-tree in intervals is done in a way that every interval in it has approximately as many messages as a physical block on the storage device can store. (It is a matter of choice whether to include the intervals which are in memory into the model - we will include them for better illustration).

The elements of a given set of intervals can be permuted in a different way and in different number of levels. We will call such combination an *intervals configuration*. Thus, each of the different indexing paradigms follows a specific *interval model* – specific intervals configuration is maintained, which meets well defined restrictions (Fig. 4). From that point of view, the indexing algorithm transforms a configuration of intervals into one that has only one level of intervals.

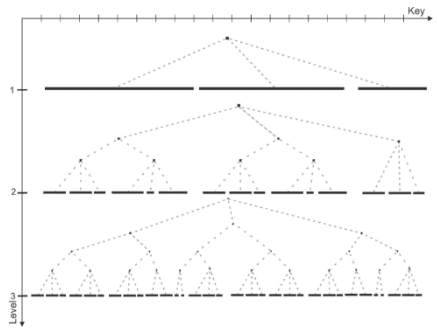

Fig. 3. Interval model of LSM-tree based data structure - has three levels of intervals without the requirement for subordination of intervals. Level 1 - AVL-tree (RAM), Level 2 - B+-tree (HDD), Level 3 - B+-tree (HDD).

As we can see, the intervals configuration of the different indexing technologies evolve towards increasing the number of levels and weakening the requirements for intersections among the intervals. But one thing remains unchanged in the current indexing algorithms – **the intervals do not change their levels and the merging of the data is always between two adjacent levels**. There are no limitations for us **to allow swapping of intervals between the levels and to merge multiple levels simultaneously** in a way that will reduce the overall number of I/O operations needed for the data indexing.

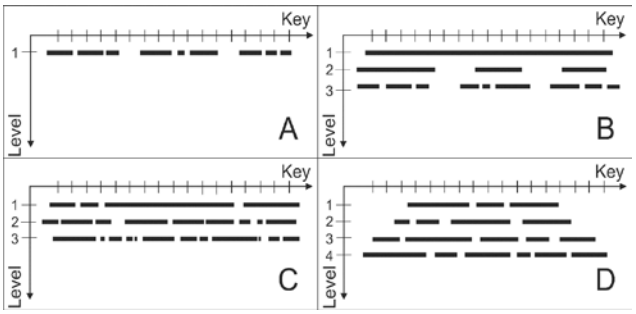

Fig. 4. Interval model of: A) B-tree – one level of intervals; B) Buffer Tree, FractalIndex Tree, WaterfallTree – subordination of the intervals; C) LSM-tree – increasing number of intervals in the levels; D) NoTree – no relations between the intervals.

II. NoTree

In NoTree, the input data is logged in files directly on the hard disk drive, each with approximately equal size B. The input data is logged in the form of messages. After a given file reaches its initially defined size B, the logging process continues in a new file. Parallel to the data logging, a statistic is maintained for the minimal and maximal key in every file. Thus, the data in each file defines an interval and in the process of logging the interval bounds are maintained. All intervals of the indexing structure are held into one set of intervals S. There are no connections, relations or subordination between the intervals (Fig. 4D). In that way the only limit for the incoming data is the write speed of the hard drive. The data indexing is done cyclically in two stages: *k-partitioning* and *multiway merging*:

1. **K-partitioning** – the intervals from S are distributed in sequence of groups which meet the following conditions:

a) For every group of intervals exists a configuration which has less than k levels. This condition guarantees that the intervals from a given group can be merged together with one pass when the available memory is limited to the size of k blocks ($k \times B$);

b) The k-partitioning should have as smaller *sum* as possible (see *K-partitioning of a set of intervals*). That sum (see Fig. 5) is the half of the number of I/O operations needed for transforming the index into one level of intervals. That follows from the definition of k-partitioning.

2. **Merging** – multiway merge of K levels of intervals in the last group of the k-partitioning.

The process continues until only one level of intervals in S remains. In the meantime, if read queries enter the NoTree, the indexing cycle is interrupted and is not executed for the entire set S, but for the subset of intervals in S defined by the queries. In that way the completed work until now is not lost and the merged intervals for the incoming queries contribute to the whole process of indexing. After execution of the queries, the indexing cycle of NoTree continues its work with the whole set of intervals S.

Essentially the indexing process is a modified external sort in two directions:

1. The k levels of intervals are chosen by the k-partitioning in such a way that the number of I/O operations when merging will be reduced. In case the index is static (no data is coming in for indexing) and the data has random keys, the order in which the intervals are chosen for merging would be irrelevant. But the structure itself is dynamic and while the existing intervals are merging, new data is coming in which forms new intervals. In that way two subsets of intervals are formed within the index – on one hand the already merged intervals which form levels of non-overlapping intervals and on the other hand newly formed intervals which form specific patterns depending on the incoming keys (in many applications the incoming data has a specific keys pattern). Thus, the k-partitioning is a significant step for determining the order for merging of intervals.

2. A significant part of the merging process is the "skipping" of intervals. The intervals that are already merged are non-overlapping and short, i.e. they have a high "density". The newly entered intervals are long and overlap with many of the existing intervals but have a low density. That means that between two keys in the interval the probability for having intervals which can be skipped is increased with the increasing size of the index. Similar thing is happening in LSM-tree when two subsequent levels are merging – blocks for which no input keys are coming in are not loaded.

A. K-partitioning of a set of intervals

Let X be a set of objects for which a total (linear) order is defined [12]. Then the concept of closed interval over X is well defined. Let I be a set of intervals over X. We will mark with I^p the subset of I that contains all intervals in which the point $p \in X$ belongs, i.e. $I^p = \{i \in I | p \in i\}$. We will define *depth* of I as $\bar{I} = \max_{p \in X} |I^p|$, i.e. the maximal number of overlapping

intervals in a point. It is obvious that no intervals configuration of I exists, with less levels of intervals than the depth of I.

Let I be a set of intervals and every interval refers a block with size B. If the available operating memory is M and $k = M/B$, then in every given moment we can simultaneously keep k blocks in memory. Therefore we can merge simultaneously k levels of intervals. Our goal is to re-order and merge the intervals of I with the least amount of I/O operations, in a way that at the end I will hold only one level of intervals, i.e. $\bar{I} = 1$.

We will call *K-partitioning* of I the sequence $\Delta_k = G_1, G_2, \dots, G_n$, from groups of intervals $G_i \subseteq I$, for which the following conditions are met:

1. $G_i \neq \emptyset,\ i = 1, 2, \dots, n$, i.e. in every group there is at least one interval;
2. $G_i \cap G_j = \emptyset, 1 \leq i, j \leq n, i \neq j$, i.e. the groups do not contain common intervals;
3. $\bigcup_{i=1}^{n} G_i = I$, i.e. all intervals from I participate in Δ_k;
4. $\bar{G}_i < k,\ i = 1, 2, \dots, n$, i.e. every group G_i of Δ_k has a depth less than k, where $k \geq 2$ is constant.

We define a *sum* of k-partitioning $\Delta_k = G_1, G_2, \dots, G_n$ as $S(\Delta_k) = 1|G_1| + 2|G_2| + \dots + n|G_n| = \sum_{i=1}^{n} i|G_i|$, where $|G_i|$ is the number of intervals in the group $G_i \in \Delta_k$ (Fig. 5). Directly from the definition follows that if $\alpha_k = A_1, A_2, \dots, A_m$ and $\beta_k = B_1, B_2, \dots, B_n$ are two k-partitions of I, where $m \leq n$ and $|A_i| \leq |B_i|$ for $i = 1,2, \dots, m$, $S(\alpha_k) \leq S(\beta_k)$. We can see that the sum defines a partial order [32] between the k-partitions, i.e. we will say that for two k-partitions α_k and β_k, $\alpha_k \leq \beta_k$, exactly when $S(\alpha_k) \leq S(\beta_k)$. Therefore for every set of intervals I exist k-partitions with a minimal sum, which we will call *minimal* k-partitions.

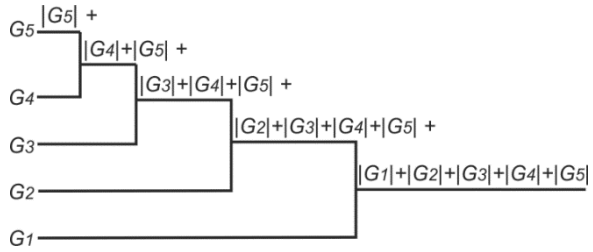

Fig. 5. Sum of k-partitioning that contains a total of 5 groups of intervals.

Exactly the minimal k-partitions are interesting to us, because the upper bound of I/O operations needed for converting a random set of intervals in a set that has one level of intervals is $2 \times S(\Delta_k)$ (once for writing and once for reading).

After finding a k-partition, begins a process of merging the intervals of the last group. Moreover, when the last group is merged, a new level of intervals is formed which can change the k-partition in such one that has a smaller sum.

III. Test Results

All of the tests (Fig. 6, Fig. 7) are performed with the publicly available open source tool *Database Benchmark 3.0*. The tests with random keys are conducted on the following computer configuration: CPU Intel Core i7-3770 3.4 GHz with 8MB L3 cache, RAM 32GB DDR3-1600 MHz, HDD 3TB Western Digital Caviar Green, OS Windows 7 Professional Service pack 1 64 bit with NTFS / Ubuntu 14.04 64 bit with EXT4. The test data has the following structure: ***key*** – integer of type long (8 bytes); ***record*** – complex type (~52 bytes). The number of records for all tests is $N = 10^9$. A pseudo-random generator is used for generating the random keys. A random-walk algorithm is used for generating the records. The tests are conducted with the following databases: NoTree (STSdb 5 alpha), FractalIndex Tree (TokuDB 7.5.5 Community Edition for MySQL 5.5.41), LSM-tree (LevelDB 1.16).

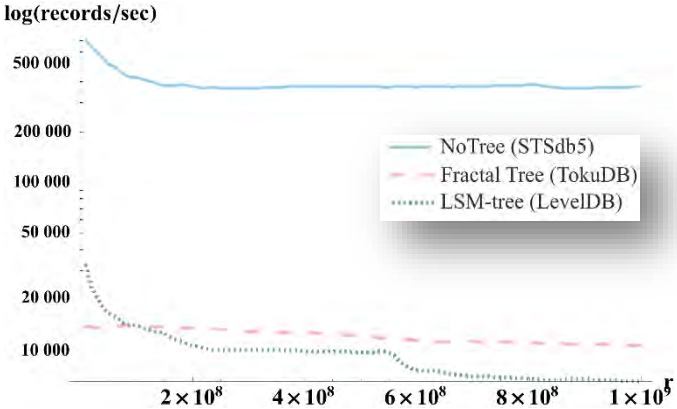

Fig. 6. Insert of 10^9 records with random keys. Average speeds: NoTree - 371 504 rec/s, FractalIndex Tree - 10 869 rec/s, LSM-tree - 6 592 rec/s.

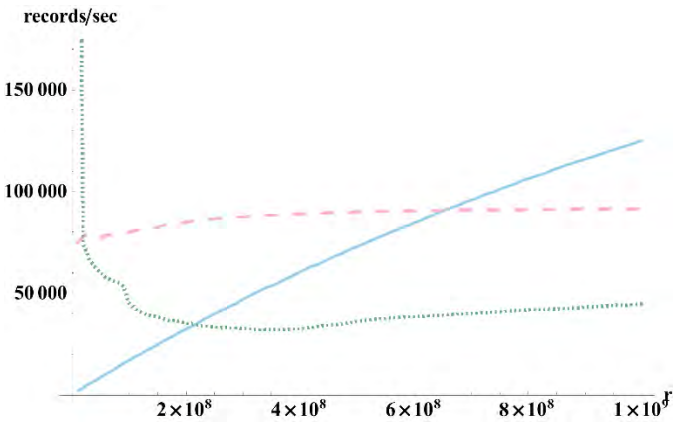

Fig. 7. Immediately full read. First sequential (in ascending order) read of all records immediately after they are inserted. Average speeds: NoTree – 125 149 rec/s, FractalIndex Tree – 91 485 rec/s, LSM-tree – 44 509 rec/s.

Total time for the whole test (Insert + Immediately full read) of 10^9 records: NoTree – 2h 58min=(45min + 2h 13min), FractalIndex Tree – 28h 35min=(25h 33min + 3h 2min), LSM-tree – 48h 22min=(42h 8min + 6h 14min).

IV. Results Analysis

The insert test (on Fig. 6) shows that when inserting a 10^9 records with random keys, NoTree is 34 times faster than FractalTree and 56 times faster than LSM-tree. (The results do not include a representative of a B-tree index because its speed drops to under 500 rec/sec during the tests. After 5 days it cannot insert even half of the records.)

The high speed which the NoTree index achieves is a direct consequence of its philosophy – there are no connections, relations or subordination between the data intervals (Fig. 4D). Thus, a main limitation for the incoming data into the index is the throughput of the external storage device. In other technologies the indexing structure dominates by enforcing constraining relations between the intervals. Those relations chiefly define a *local merging strategy* whereas in NoTree we have a *global merging strategy* for the intervals, i.e. the choice which buffers to be merged is done between all buffers that are available in the index.

When there is no previous information for the distribution of the incoming keys in the index and the data is coming at a faster rate that can be indexed, one of the things that can be done is to postpone the actual ordering process of the records. That by itself leads to increase in the *levels of intervals* in LSM-tree, FractalTree and NoTree. Thus, after the insert test (in general) there will be more than one level of intervals in the indexing structures. And so the indexing task is actually brought down to effectively reducing the number of levels.

In order to make an estimate of how much time is needed for each one of the indices to convert its set of intervals into one level of intervals, it is needed all of the records to be read (in ascending order by their keys) right after the insert test completes. That forces the index – either tree based or non–tree based, to bring the structure into one that has only one level of intervals. The result from the immediate full read is shown on Fig. *7*. The chart shows that the read speed of NoTree increases because as time progresses fewer levels of intervals remain.

It is clear from the total time for the complete test (*insert + immediate full read*) that the needed time for a complete execution of the whole test by NoTree is 10 times less than that of FractalTree and 16 times less than the time of LSM-tree.

A rough estimate can also be made about the time needed for the full ordering of the data. Because the indexing process in the different indexing technologies is distributed between the *Insert* and *Immediate Full Read* processes, the estimate can be made by subtracting the times *for generating, sequential write in a file* and *sequential read from the file* of all records from the total time. In our case, for 10^9 records and specified hardware this time (measured with the same application) is around 43 minutes. Thus, the actual time needed by NoTree for the complete ordering of 10^9 records is 12 times less than the time of FractalTree and 21 times less than LSM-tree.

It can also be noted that for every following data read – regardless if it is a point or range query, NoTree will increase its speed additionally compared to the other indexing technologies. NoTree will be naturally transformed into a linear structure from non-overlapping intervals after the full ordering of the data after the first read, whereas all other tree based indexing structures will have an internal height (because of their tree-like nature), regardless of the fact that they also have one level of intervals.

I. Conclusion

The present article introduces a new external index data structure called NoTree, which can be defined as a next step in the development of index data structures. NoTree introduces a new approach in terms of building the indices. Instead of building indices based on keys, NoTree builds the indices based on intervals from keys.

NoTree works by periodically performing a process of *selecting* and *merging* of data intervals, until non-overlapping intervals are left. The selecting and merging are done with the described *k-partitioning* of intervals and the subsequent *merging* of some of them. Thus, with the so described *k-partitioning* and *merging*, NoTree achieves a significant reduce of the number of I/O operations needed in the indexing process.

Apart from the more efficient indexing, NoTree also removes the need of subordination between the stored buffers. NoTree lacks a root and an idea for subordination that are common for the tree-based data structures, which limit the write and read speed. This gives possibilities for different parallelizations. In the process of working, NoTree can transform itself into an entirely linear structure of non-overlapping intervals, which increases the random and sequential read of records.

The article also proposes an *interval model* (view), which shows the conceptual differences between NoTree and the currently widespread tree-based indexing structures. The article also includes comparative tests and results.

The proposed cycle for *k-partitioning* and *merging* can also be used for effective external sort. The question for finding the minimal k-partition of a set of intervals remains open.

References

[1] R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indexes", *Acta Informatica*, vol. 1, p. 173–189, 1972.

[2] D. Comer, "The Ubiquitous B-Tree", *Computing Surveys*, vol. 11 No 2, June 1979.

[3] D. E. Knuth., Sorting and Searching, volume 3 of The Art of Computer Programming, vol. 3, Massachusetts: Addison-Wesley, 1973.

[4] L. Arge, "The Buffer Tree: A New Technique for Optimal", *Algorithms and Data Structures*, vol. 955, pp. 334-345, 1995.

[5] L. Arge, "The buffer tree: A technique for designing", *Algorithmica*, vol. 37, pp. 1-24, September 2003.

[6] M. A. Bender, M. Farach-Colton, J. T. Fineman, B. C. K. Y. Fogel and J. Nelson, "Cache-oblivious streaming B-trees.", *In Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 81-92, June 9–11 2007.

[7] I. Tronkov, "WaterfallTree — External indexing data structure", in *Automation, Quality and Testing, Robotics, 2014 IEEE International Conference*, Cluj-Napoca, 2014.

[8] I. Tronkov, "Mathematics behind WaterfallTree", in *International scientific conference UNITECH*, Gabrovo, 2014.

[9] I. Tronkov, "WaterfallTree implementation details", in *International scientific conference UNITECH*, Gabrovo, 2014.

[10] P. O'Neil, E. Cheng, D. Gawlick and E. O'Neil, "The log-structured merge-tree (LSM-tree)", *Acta Informatica*, vol. 33, pp. 351-385, June 1996.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chadra, A. Fikes and R. E. Gruber, "Bigtable: A distributed storage system for structured data.", *ACM Transactions on Computer Systems*, p. 26(2), June 200.

[12] К. Куратовски, Увод в теорията на множествата и топологията, София: Наука и изкуство, 1979.