NEFOS: Rapid Cache-Aware Range Query Processing with Probabilistic Guarantees

Spyros Sioutas[1], Kostas Tsichlas[2], Ioannis Karydis[1], Yannis Manolopoulos[2], and Yannis Theodoridis[3]

[1] Department of Informatics, Ionian University, Greece
(sioutas,karydis)@ionio.gr
[2] Department of Informatics, Aristotle University of Thessaloniki, Greece
(tsichlas,manolopo)@csd.auth.gr
[3] Department of Informatics, University of Piraeus, Greece
ytheod@unipi.gr

Abstract. We present NEFOS (NEsted FOrest of balanced treeS), a new cache-aware indexing scheme that supports insertions and deletions in $O(1)$ worst-case block transfers for rebalancing operations (given and update position) and searching in $O(\log_B \log n)$ expected block transfers, ($B=$ disk block size and $n=$ number of stored elements). The expected search bound holds with high probability for any (unknown) *realistic* input distribution. Our expected search bound constitutes an improvement over the $O(\log_B \log n)$ expected bound for search achieved by the ISB-tree (Interpolation Search B-tree), since the latter holds with high probability for the class of *smooth* only input distributions. We define any unknown distribution as realistic if the smoothness doesn't appear in the whole data set, still it may appear locally in small spatial neighborhoods. This holds for a variety of real-life non-smooth distributions like skew, zipfian, powlaw, beta e.t.c.. The latter is also verified by an accompanying experimental study. Moreover, NEFOS is a B-parametrized concrete structure, which works for both I/O and RAM model, without any kind of transformation or adaptation. Also, it is the first time an expected sub-logarithmic bound for search operation was achieved for a broad family of non-smooth input distributions.

Keywords: Data Structures, Data Management Algorithms.

1 Introduction

More than three decades after its invention, B-tree [5] and its variants remain the ubiquitous external memory data structure for indexing and organizing large data sets with numerous applications, especially in database systems. Its popularity is mainly due to the $O(\log_B n)$ worst-case complexity (block transfers) for search and update operations. The most heavily used application is the efficient answering of one-dimensional range search queries using $O(\log_B n + r)$ block transfers, where $R = rB$ is the number of elements reported, B is the block-size and n the number of element. In this paper, we consider one of the most known

A. Hameurlain et al. (Eds.): DEXA 2011, Part I, LNCS 6860, pp. 62–77, 2011.
© Springer-Verlag Berlin Heidelberg 2011

and widely used such models, namely the two-level memory hierarchy model introduced in [2,25]. In this model, the memory hierarchy consists of an internal (main) memory and an arbitrarily large external memory (disk) partitioned into blocks of size B. The data from the external to the main memory and vice versa are transferred in blocks (one block at a time).

A large number of variants of the B-tree have been proposed since its appearance in order to improve its performance in practice for various applications — see the excellent survey by Vitter [24] for an extended accounting of these and other variants and their applications — to make it parallel for use in multi-disk environments [21], to tune it for concurrency and recovery purposes [14,22], to extend it to cover other than the original field [9], etc. Regarding the update operation, it should be noted that an update operation consists of three consecutive phases: a search phase (to locate the place of the update), an element-updating phase (to insert the new element, or delete the located element), and a rebalancing phase (to restore the B-tree structure). Excluding the first phase (search operation), the dominating phase of an update operation is the rebalancing one, since the element-updating phase takes typically $O(1)$ block transfers (and/or time). In the case of B-tree and its variants, the rebalancing phase requires $\Theta(\log_B n)$ block transfers in the worst-case. This implies that the update operation takes $\Theta(\log_B n)$ block transfers, even in the case where the update position (block within which the update will take place) is given.

ISB-tree (Interpolation Search B-tree) presented in [12], supports search operations in $O(\log_B \log n)$ expected block transfers *with high probability* (w.h.p.) for a large class of input distributions (including both uniform and non-uniform classes) described below, and update operations in $O(1)$ block transfers, provided that the update position is given. The search bound implies that a one-dimensional range search query can be supported in $O(\log_B \log n + r)$ expected block transfers with high probability. The worst-case block transfers for the search operation are $O(\log_B n)$.

The expected search bound was achieved by considering a rather general scenario of *μ-random* insertions and *random* deletions, where μ is a so-called *smooth* probability density [3,19]. An insertion is μ-random if the key to be inserted is drawn randomly with density function μ; a deletion is random if every key present in the data structure is equally likely to be deleted [13]. Informally, a distribution defined over an interval I is *smooth* if the probability density over any subinterval of I does not exceed a specific bound, however small this subinterval is (i.e., the distribution does not contain sharp peaks). Smooth distributions are a superset of uniform, bounded, and several non-uniform distributions (e.g., the class of regular distributions introduced by Willard [26]).

In this paper, we present NEFOS (NEsted FOrest of balanced treeS), a new cache-aware indexing scheme that supports insertions and deletions in $O(1)$ worst-case block transfers for rebalancing operations (provided that the update position is given) and searching in $O(\log_B \log n)$ expected block transfers, where B represents the disk block size and n denotes the number of stored elements. The expected search bound holds with high probability for any (unknown)

realistic input distribution. Our expected search bound constitutes an improvement over the $O(\log_B \log n)$ expected bound for search achieved by the ISB-tree (Interpolation Search B-tree), since the latter holds with high probability for the class of smooth only input distributions. Note here that *realistic* distributions are a superset of smooth distributions. Generally speaking, in $(f1, f2)$-smooth distributions, $f1$ measures how fine is the partitioning of an arbitrary subinterval as well as $f2$ measures the sparseness of this subinterval. In this context, any probability distribution is $(f1, \Theta(n))$-smooth. For smooth class of densities, $f2 = \Theta(n^{\delta})$, where $0 < \delta < 1$. We define any unknown distribution as realistic if there is at least one subinterval of $\Theta(n)$ sparseness and there are at least $\Theta(n^{\delta})$ consecutive subintervals of $\Theta(n^{1-\delta})$ sparseness, where $0 < \delta < 1$. This holds for a variety of real-life non-smooth distributions like skew, zipfian, powlaw, beta e.t.c., where the *smoothness* property appears locally in small spatial neighborhoods. The latter is also verified by an accompanying experimental study. Moreover, NEFOS is a B-parametrized concrete structure, which works for both I/O (arbitrary B) and RAM (B=2) model, without any transformation or adaptation. Also, it is the first time an expected sub-logarithmic bound for search operation was achieved for a broad family of non-smooth input distributions.

External data structures related to our approach are those based on hashing [18,24]. The main representatives of external memory hashing methods include: extendible hashing [8], linear hashing [16], and external perfect hashing [10]. These hashing schemes and their variants need $O(1)$ expected block transfers for answering search queries, but they share various disadvantages when compared to our structure: (i) they do not support range queries; (ii) their expected case analysis usually assumes *uniform* input distributions (or input distributions that produce uniform hash key values); and (iii) they exhibit poor worst case performance.

The remainder of the paper is organized as follows. In Section 2, we discuss preliminary notions and results that are used throughout the paper, define formally the class of smooth probability distributions as well as discuss the *ISB-Tree*. The main result of this paper, the *NEFOS-tree*, with the complexity analysis of its operations is discussed in Section 3. Section 4 provides an experimental evaluation with synthetic and real data of our theoretical findings. We conclude in Section 5.

2 Preliminaries

This Section briefly describes B-trees, input distributions in the context of internal memory data structures as well as the static interpolation search tree.

The B-tree. The *B-tree* is a $\Theta(B)$-ary tree (with the root possibly having smaller degree) built on top of $\Theta(n/B)$ leaves. The degree of internal nodes, as well as the number of elements in a leaf, is typically kept in the range $[B/2, B]$ such that a node or leaf can be stored in one disk block. All leaves are on the same level and the tree has height $O(\log_B n)$. This guarantees that a search operation can be accomplished within $O(\log_B n)$ block transfers. An insertion is performed

in $O(\log_B n)$ block transfers by first searching down the tree for the relevant leaf l. The insertion there may cause a split and the latter may propagate up the tree. Similarly, a deletion can be performed in $O(\log_B n)$ block transfers by first searching down the tree for the relevant leaf l and then removing the deleted element. The deletion there may cause a fusion and the latter may propagate up the tree.

The Lazy B-tree. The Lazy B-tree of [12] is a simple but non-trivial externalization of the techniques introduced in [20]. The first level consists of an ordinary B-tree, whereas the second one consists of buckets of size $O(\log^2 n)$, where n is approximately equal to the number of elements stored in the access method. The following theorem provides the complexities of the Lazy B-tree:

Theorem 1. The Lazy B-Tree supports the search operation in $O(\log_B n)$ worst-case block transfers and update operations in $O(1)$ worst-case block transfers, provided that the update position is given.

Proof. see [12].

The ISB-tree. The ISB-tree is a two-level data structure. The upper level is a non - straightforward externalization of the Static Interpolation Search Tree (SIST) presented in [11]. In the definition of the (f_1, f_2)-smooth densities [26,19], intuitively, function f_1 partitions an arbitrary subinterval $[c_1, c_3] \subseteq [a, b]$ into f_1 equal parts, each of length $\frac{c_3 - c_1}{f_1} = O(\frac{1}{f_1})$; that is, f_1 measures how fine is the partitioning of an arbitrary subinterval. Function f_2 guarantees that no part, of the f_1 possible, gets more probability mass than $\frac{\beta \cdot f_2}{n}$; that is, f_2 measures the sparseness of any subinterval $[c_2 - \frac{c_3 - c_1}{f_1}, c_2] \subseteq [c_1, c_3]$. The class of (f_1, f_2)-smooth distributions (for appropriate choices of f_1 and f_2) is a superset of both regular and uniform classes of distributions, as well as of several non-uniform classes [3,11]. Actually, *any* probability distribution is $(f_1, \Theta(n))$-smooth, for a suitable choice of β. The following theorem presented in [12] follows and holds for the very broad class of $(n/(\log\log n)^{1+\epsilon}, n^{1-\delta})$-smooth densities, where $\delta = 1 - \frac{1}{B}$ and includes the uniform, regular, bounded as well as several non-uniform distributions.

Theorem 2. Suppose that the upper level of the ISB-tree is an external static interpolation search tree with parameters $R(s_0) = s_0^{\delta}$, $I(s_0) = s_0/(\log\log s_0)^{1+\epsilon}$, where $\epsilon > 0$, $\delta = 1 - \frac{1}{B}$, $s_0 = n_0$, n_0 is the number of elements in the latest reconstruction, and that the lower level is implemented as a forest of Lazy B-trees. Then, the ISB-tree supports search operations in $O(\log_B \log n)$ expected block transfers with high probability, where n denotes the current number of elements, and update operations in $O(1)$ worst-case block transfers, if the update position is given. The worst-case update bound is $O(\log_B n)$ block transfers, and the structure occupies $O(n/B)$ blocks.

Proof. see [12].

3 NEFOS

In the following we present the building phase of NEFOS as well as the complexity analysis of its basic operations.

3.1 Building NEFOS

Let $\mu(\cdot)$ random be the sequence of inserted keys and random be the sequence of deleted keys. Let n be the total number of w-bit keys, which are organized in block fashion. Let $n_1 = O(n/B)$ the number of these blocks. Let also $n_2 = O(n_1/log_B log_B n)$ super-blocks each of which contains $O(log_B log_B n)$ blocks. Let also $\mu_1(\cdot)$ random be the sequence of keys of super-block representatives. According to combinatorial game of bins and balls presented in [11], we store these keys in $N = n_2/lnn_2$ buckets, each of which contains $O(lnn_2)$ keys.

Lemma 1. If the sequence of inserted keys remains $\mu(\cdot)$ random then the sequence of super-block representative keys remains $\mu_1(\cdot)$ random.

Proof. See [11].

Lemma 2. Given a $\mu_1(\cdot)$ random sequence of inserted super-block representative keys and a random sequence of deleted super-block representative keys, the load of each bucket never becomes zero and never exceeds $\Theta(\text{polylog } N)$ keys in expected w.h.p. case.

Proof. See [11].
In $(f1, f2)$-smooth distributions, $f1$ measures how fine is the partitioning of an arbitrary subinterval and $f2$ measures the sparseness of this subinterval. In this context, any probability distribution is $(f1, \Theta(n))$-smooth. In any realistic distribution there are sparse subintervals of $\Theta(n)$ sparseness and dense subintervals of $\Theta(n^{1-\delta})$ sparseness, where $0 < \delta < 1$. For example in Figure 1, we depict with red color a sparse subinterval and with blue color a number of consecutive dense subintervals.

In the same context and according to Lemmas 1 and 2 we construct sparse and dense buckets of polylogaritmic load. For example see the red and blue buckets of Figure 2.

Let $bucket_{I_{max}}$ the bucket with the maximum range of keys (I_{max}) and $bucket_{I_{min}}$ the bucket with the minimum range of keys (I_{min}). For any realistic distributions $I_{max} = \Theta(n)$ and $I_{min} = \Theta(n^{1-\delta})$, where $0 < \delta < 1$.

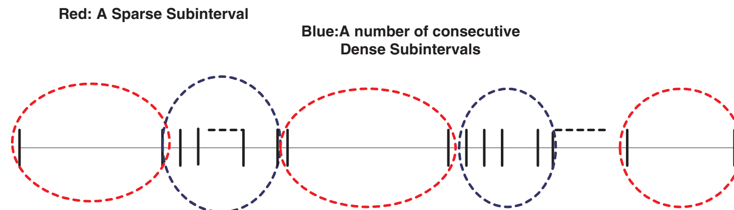

Fig. 1. Sparse and dense subintervals of any arbitrary distribution

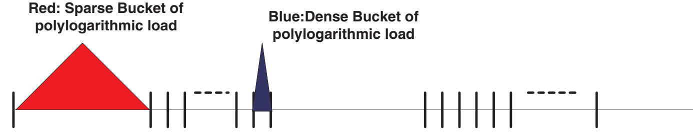

Fig. 2. Red and Blue Buckets

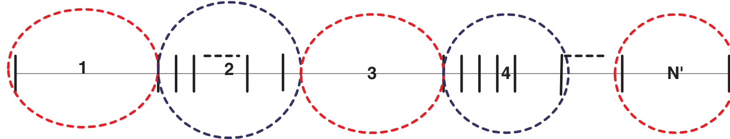

Fig. 3. Red and Blue Labeled Cluster_Nodes. Blue Cluster_node contains at least one dense subinterval.

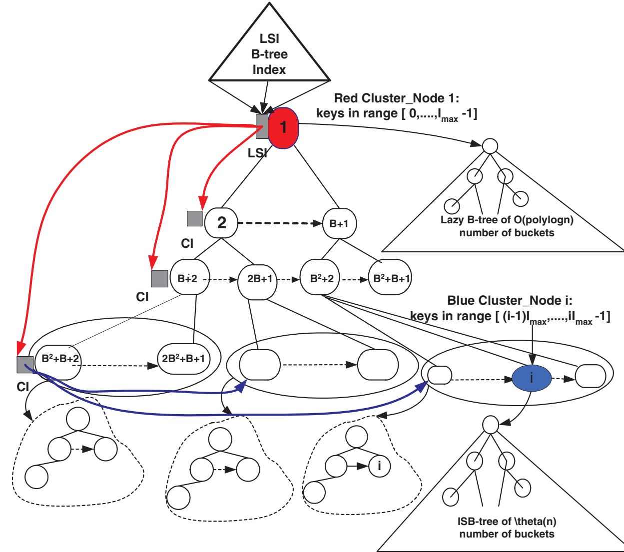

Fig. 4. NEsted FOrest of load-balancing treeS in I/O model

Now, we build labeled cluster_nodes. Each cluster_node with label i' (where $1 \leq i' \leq N'$) stores ordered buckets with keys belonging in the range $[(i'-1)I_{max}, \ldots, i'I_{max} - 1]$, where N' is the number of cluster_nodes (see Figure 3).

NEFOS stores cluster_nodes only, each of which is structured either as a Lazy B-tree if its color is red or as an ISB-tree if its color is blue. In particular, NEFOS is built by grouping cluster_nodes having the same ancestor and organizing them in a tree structure recursively. The innermost level of nesting (recursion) will be characterized by having a tree in which no more than B cluster_nodes share the same direct ancestor, where B is the disk block. Thus, multiple independent trees are imposed on the collection of nodes (see Figure 4).

The degree of the nodes at level $i > 0$ is $d(i) = t(i)$, where $t(i)$ indicates the number of nodes at level i. It is defined that $d(0)$=B and $t(0)$=1. It is apparent that $t(i) = t(i-1)d(i-1)$, and, thus, by putting together the various components, we can solve the recurrence and obtain $d(i) = t(i) = B^{2^{i-1}}$ for $i \geq 1$. We stope at the level where each collection contains $O(N'^{1/B})$ cluster_nodes. For example in figure 4, the root is located at level 0, thus the funout of root is exactly B. In particular the B cluster_nodes located at level 1 have labels 2, 3, $\ldots B + 1$ respectively. At level 1, the B labeled nodes $B + 2, \ldots, 2B + 1$ rooted at labeled node 2, the B labeled nodes $2B + 2, \ldots, 3B + 1$ rooted at labeled node 3, e.t.c. and the B labeled nodes $B^2 + 2, \ldots, B^2 + B + 1$ rooted at labeled node $B + 1$. At level 2, the funout is B^2, so the B^2 labeled nodes $B^2 + B + 2, \ldots, 2B^2 + B + 1$ rooted at labeled node $B + 2$ and so on.

We also equip the root cluster_node with a table named *Left Spine Index* (LSI), which stores pointers to the cluster_nodes of the left-most spine. We organize LSI table as a B-tree. For example in figure 4, see the red pointers beginning from cluster_node 1 towards cluster_nodes with labels 2, $B + 2$ and $B^2 + B + 2$ respectively.

Furthermore, each cluster_node of the left-most spine is equipped with a table named *Collection Index* (CI), which stores pointers to the collections of cluster_nodes presented at the same level. Cluster_nodes having the same father belong to the same collection. We also organize CI tables in block fashion. For example in figure 4, see the blue pointers beginning from the left-most collection towards the other collections located at the same level.

Finally, each cluster_node is organized in a Lazy B-tree manner and each collection is organized in a NEFOS manner at the next level of nesting (see the dash lines in figure 4).

Remark 1. If we parametrize B and choose such a small value (f.e. B=2) so as the whole structure can fit in main memory as well as replace each lazy B-tree and the B-tree of LSI structure with q*-heap machinery [27], then NEFOS becomes a data structure in RAM model with the same expected complexities w.h.p. for all operations, without any kind of transformation or adaptation.

3.2 Complexity Analysis

We will focus first on space and then on time complexity of NEFOS' basic operations.

Space Complexity Analysis. The double exponentially increasing fanout guarantees the following lemma:

Lemma 3. The height (or the number of levels) of NEFOS is $O(\log \log_B n)$ in the worst case.

Proof. It is obvious if we solve for i the equation $B^{2^{i-1}} = O(N'^{1/B})$.

Now, since the innermost level of nesting (recursion) is characterized by having a tree in which no more than B cluster_nodes share the same direct ancestor, the lemma 4 follows:

Lemma 4. The maximum number of possible nestings in NEFOS structure is $O(\log_B \log_B n)$ in the worst case.

Proof. It is obvious that in NEFOS structure of jth nested level, the last collections contain $O(N'^{1/B^j})$ cluster_nodes. Since the innermost level of nesting (recursion) is characterized by having a tree in which no more than B cluster_nodes share the same direct ancestor, it holds that $O(N'^{1/B^j}) = B$, meaning that $j = O(\log_B \log_B N')$ or $j = O(\log_B \log_B n)$.

Finally, the sizes of CI and LSI tables are described by the following lemma:

Lemma 5. The maximum size of the CI and LSI tables is $O(\frac{n^{1/B}}{B})$ and $O(\frac{\log \log_B n}{B})$ in worst-case respectively.

Proof. Since the maximum number $(O(n^{1/B}))$ of cluster_nodes appears at last level of the basic (non-nested) NEFOS structure, the length of CI is $O(n^{1/B})$. Since, CI has been organized in a block fashion,the $O(\frac{n^{1/B}}{B})$ space complexity follows. The length of LSI table depends on the height of NEFOS. Thus according to lemma3, this length becomes $O(\log \log_B n)$. Since, LSI has been organized in a block fashion (according to B-tree),the $O(\frac{\log \log_B n}{B})$ space complexity follows.

Each cluster_node appears in $O(\log_B \log_B n)$ nesting levels in the worst case. As a result each bucket appears in $O(\log_B \log_B n)$ nesting levels in the worst case and as a result each super-block appears in $O(\log_B \log_B n)$ nesting levels in the worst case. Since, each super-block contains $O(log_B log_B n)$ blocks, we have $O(n/B)$ blocks in total and the theorem foloows:

Theorem 3. The whole space of NEFOS remains linear.

Query Processing, Data Insertion, Data Deletion. Assume we are located at root cluster_node and seek a key k. First, we find the range where k belongs in. Let say $k \in [(j-1)\ I_{max}, jI_{max} - 1]$. The latter means that we have to search for cluster_node j. The first step of our algorithm is to find the level where the desired cluster_node j is located. For this purpose, we organize the cluster_node labels pointed by the LSI table in a B-tree manner (see Figure 4). Since, according to lemma 5, the maximum size of the LSI table is $O(\frac{\log \log_B n}{B})$ in worst-case, the theorem 4 follows.

Theorem 4. The level where the desired cluster_node j is located can be found out in $O(\log_B(\frac{\log \log_B n}{B}))$ I/Os.

Let say that cluster_node j is located at the i-th level. We follow the i-th pointer of the LSI table located at root cluster_node so as to reach the leftmost cluster_node x of level i. Then, we compute the collection in which the cluster_node j belongs. Since the number of collections at level i equals the number of cluster_nodes located at level $(i-1)$, we divide the distance between j and x by the factor $t(i-1)$. Let m (in particular $m = \left\lceil \frac{j-x+1}{t(i-1)} \right\rceil$) be the result of this division. The latter means that we additionally need $O(1)$ I/Os to follow the $(m+1)$-th pointer of the CI table so as to reach the desired collection. Since the

collection indicated by the CI[m+1] pointer is organized in the same way at the next nesting level, we continue this process recursively.

Generally speaking, we need $O(\log_B(\frac{\log\log_B n}{B})) + O(1)$ I/Os for locating the desired collection and we have to continue this process recursively for all nesting levels. Since the maximum number of nesting levels is $O(\log_B \log_B n)$ in the worst case (according to lemma 4), the whole searching process requires $T_1(n)$ I/Os to locate the target cluster_node, where:

$$T_1(n) = \sum_{i=1}^{\log_B \log_B n} \log_B(\frac{\log\log_B n^{\frac{1}{B^{i-1}}}}{B}) \tag{1}$$

from which we get:

$$T_1(n) < O(\log_B \log n)$$

Then, we have to locate the target bucket by searching the respective Lazy B-tree or ISB-tree, requiring $T_2(n)$ I/Os.

If the located cluster_node is red, then its load is polylogarithmic and the lazy B-tree index is sufficient to give us the desired complexity. If it's blue then its load may be $\Theta(n)$, and the question is how we can compute the new maximum range of keys there. Let say it $I_{max(1)}$ (see the Figure 5).

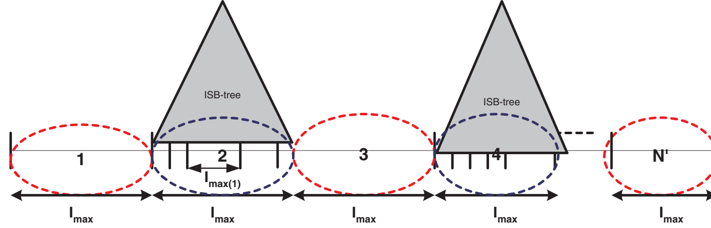

Fig. 5. Blue cluster_nodes are structured in NEFOS manner with new range Imax(1)

Since each blue cluster_node contains at least one dense subinterval of $\Theta(n^{1-\delta})$ sparseness, where $0 < \delta < 1$, for the new range $I_{max(1)}$ the following holds:

$I_{max(1)} = I_{max} - f(n) \cdot \Theta(n^{1-\delta})$, where the number of dense subintervals appearing inside a blue cluster_node is a function of n.

By setting $I_{max(1)} = \Theta(n^{1-\delta})$, we get the following:

$\Theta(n^{1-\delta}) = \Theta(n) - f(n) \cdot \Theta(n^{1-\delta})$. The latter means that: $f(n) = \frac{\Theta(n) - \Theta(n^{1-\delta})}{\Theta(n^{1-\delta})}$ or $f(n) = \Theta(n^{\delta})$. The property 1 follows:

Property 1. The number of consecutive dense subintervals is $f(n) = \Theta(n^{\delta})$.

So, now it's time to formally define what we mean with the term *any realistic distribution*.

Definition 1. Let $\mu(\cdot)$ be any random distribution in which there are sparse subintervals of $\Theta(n)$ sparseness and dense consecutive subintervals of $\Theta(n^{1-\delta})$

sparseness, where $0 < \delta < 1$. If the number of consecutive dense subintervals satisfies the property 1, then the $\mu(\cdot)$ random distribution is called *realistic distribution*.

We have to denote here that, the most known non-smooth (bad) distributions like skew, zipfian, powlaw, beta e.t.c. satisfy the property 1 for many real applications, thus would be called realistic for a huge variety of applications.

In other words, for any *realistic* distribution, each blue cluster_node in NEFOS structure satisfies the *smooth* property.

So, if the located cluster_node is red, then its load is polylogarithmic and the Lazy B-tree index requires a logarithmic number of I/Os:
$T_2(N) = O(\log_B(poly \log n))$ or $T_2(N) = O(\log_B \log n)$ I/Os or block-transfers.

If the located cluster_node is blue then its load may be $\Theta(n)$ in worst-case. Moreover, it's obvious that for any *realistic* distribution, each blue cluster_node satisfies the *smoothness* property. For this reason, we organize each blue cluster_node as an ISB-tree. In this case, $T_2(n)$ becomes as follows:

$$T_2(n) = O(\log_B \log \Theta(n)) \qquad (2)$$

As a result, the total processing time requires $T(n) = T_1(n) + T_2(n)$ I/Os and the theorem follows:

Theorem 5. Exact-match queries in the NEFOS structure require $O(\log_B \log n)$ I/Os for any *realistic* input distribution.

Having located the target cluster_node for key k_ℓ and exploiting the order of keys in each bucket, range queries of the form $[k_\ell, k_r]$ require an $O(\log_B \log n + |A| / B)$ I/Os, where $|A|$ is the number of cluster_nodes between the buckets responsible for k_ℓ, k_r respectively that are accessed in a block manner. The theorem follows.

Theorem 6. Range queries of the form $[k_\ell, k_r]$ in the NEFOS structure require $O(\log_B \log n + |A| / B)$ I/Os for any *realistic* input distribution, where $|A|$ is the answer size.

Finally, provided that the position of update is given, meaning that we have already located the target cluster_node, it remains to insert/delete the key inside the cluster_node. Since, the latter is structured either as a Lazy-B tree or as an ISB-tree, the theorem 7 follows:

Theorem 7. Update queries in NEFOS require $O(1)$ I/Os for rebalancing operations in worst-case, provided that the update position is given.

4 Experimental Evaluation

In this section, we investigate the practical merits of the NEFOS structure. Our prime concern is to (merely) investigate the practical difference of the *asymptotic complexities* (in block transfers) of search and rebalancing operations between the NEFOS structure, the ISB-tree and a cache-aware B-tree. Although there are several cache-aware B-tree variants, all of them exhibit the same asymptotic

complexities in block transfers except for lazy B-tree which guarantees constant number of block transfers for update operations. Since lazy B-tree has been incorporated into ISB-tree in order to speedup the update operations, we compare the performance of NEFOS with the ISB-tree and a simple variant of the cache-aware B-tree. Moreover, we do not compare the performance of our rebalancing operations (after an update) with hashing schemes and their variants, since the expected-case analysis of such schemes usually assumes *uniform* input distributions (or input distributions that produce uniform hash key values), and hence they exhibit poor worst-case performance for update operations. In our experimental study, we have considered both synthetic and real-world data.

4.1 Synthetic Data

For evaluation purposes we used the Java NEFOS-simulator (source code of NEFOS index is available at http://www.ionio.gr/∼sioutas/New-Software.htm). The NEFOS-simulator is extremely efficient delivering $> 100,000$ cluster nodes in a single computer system, using 32-bit JVM 1.6 and 1.5 GB RAM and full GUI support. When 64-bit JVM 1.6 and 5 RAM is utilized the NEFOS-simulator delivers $> 500,000$ cluster nodes and full GUI support in a single computer system. We have conducted an experimental study making the customary assumption that the page size is 4096 bytes, the length of each key is 8 bytes, and the length of each pointer is 4 bytes. Consequently, each block contains $B = 341$ elements. We considered data sets of size $n_0 \in [10^6, 10^{12}]$ elements generated by a variety of smooth distributions, namely uniform, regular, normal and Gaussian and non-smooth distributions, namely beta and pow-law. We compared the implementation of NEFOS, with the ISB-tree and that of a B-tree on the same data sets. Our main concern was to measure the performance, in simulated block transfers (I/Os), of the search and update operations.

The experimental results regarding the search operations are reported in Fig. 6 and 7. The sequence σ of search operations had length equal to its corresponding data set and the reported values are averages over the whole sequence. Our

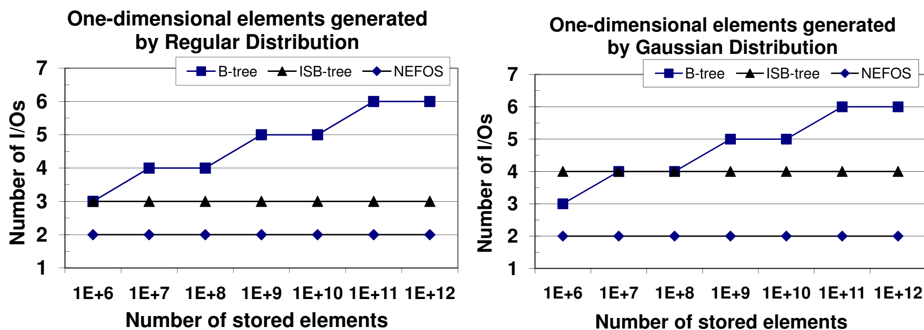

Fig. 6. Search performance for regular distributions (left) and Gaussian distributions (right)

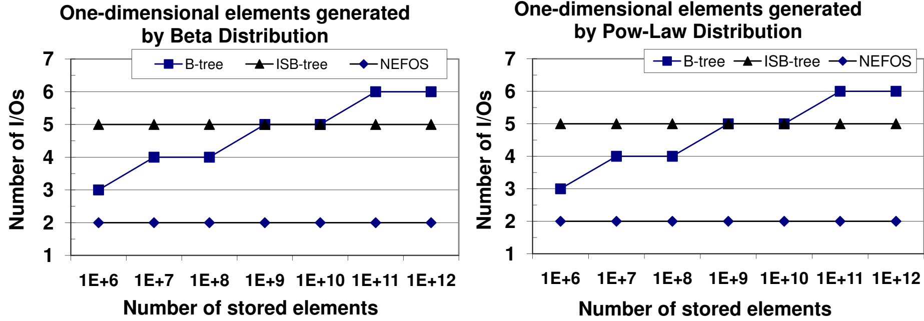

Fig. 7. Search performance for non-smooth beta (left) and powlaw distributions (right)

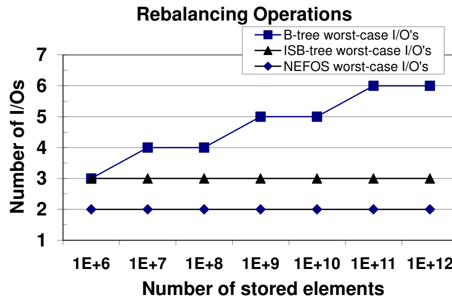

Fig. 8. Block transfers of rebalancing operations after an update

experiments revealed that the expected number of block transfers in NEFOS structure remains constant even for gigantic data sets (Terabytes - TB).

Regarding the number of block transfers required for rebalancing after an update operation to the data structure, we again considered the above values of $n_0 \in [10^6, 10^{12}]$ for our initial data sets upon which we performed update sequences of length $n_0/2$ and $2n_0$. The data structure is reconstructed every n_0 operations. Our experimental results are reported in Fig. 8. The values represent worst-case block transfers over the update sequence. We observe that the number of rebalancing operations in NEFOS structure is independent of the distribution.

4.2 Real-World Spatial Data

In this section, we deploy one-dimensional data taken from a real-world spatial dataset "LA rivers and railways" [Tiger1] and "LA streets" [Tiger2], containing 128971 and 131461 *M*inimum *B*ounded *R*ectangles (MBRs), respectively; see [23].

The one-dimensional data are taken by the x- and y-projections of MRBs and the values in each axis are normalized in [0,10000]. For all experiments, the disk page size is set to 512 bytes, the length of each key to 8 bytes, and the length of each pointer to 4 bytes. Consequently, each block contains $B = 42$ elements.

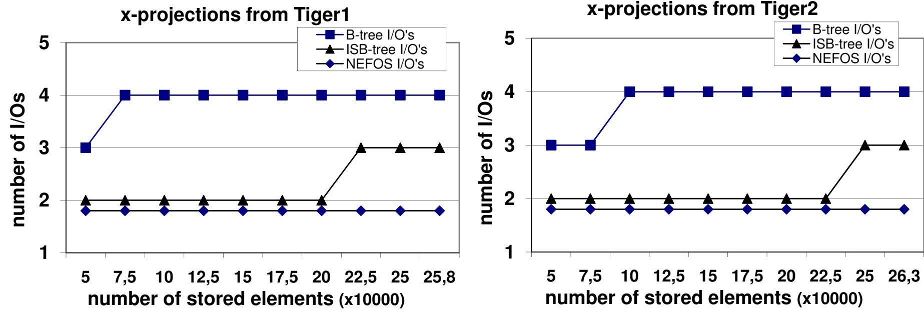

Fig. 9. Search performance for MBR's x-projections of [Tiger1] (left) and [Tiger2] (right)

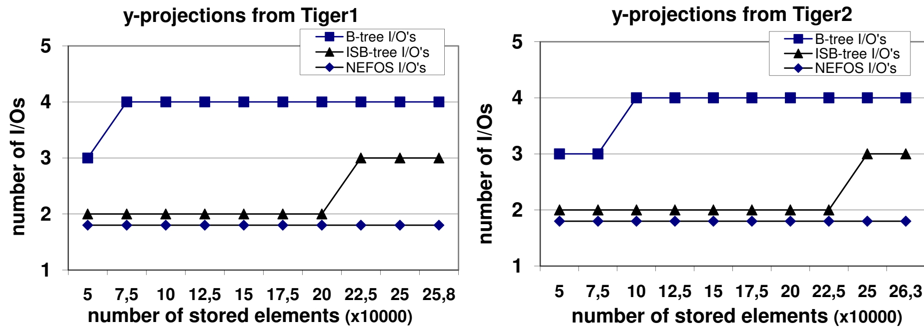

Fig. 10. Search performance for MBRs' y-projections of [Tiger1] (left) and [Tiger2] (right)

We use a relatively small page size so that the number of nodes in an index simulates realistic situations, where the data set cardinality is higher. A similar methodology was also used in [4].

Fig. 9 and Fig. 10 depict the efficiency of NEFOS structure on searching for real spatial one-dimensional data. In particular, in Fig. 9 we measured the number of I/Os required for search operations during the insertion of a total of $2 \times 128971 = 257942$ and of $2 \times 131461 = 262922$ x-projections from [Tiger1] and [Tiger2], respectively. Similarly, in Fig. 10 we measured the number of I/Os required for search operations during the insertion of a total of $2 \times 128971 = 257942$ and of $2 \times 131461 = 262922$ y-projections from [Tiger1] and [Tiger2], respectively.

Fig. 11 and Fig. 12 depict the efficiency of NEFOS structure on updating real spatial one-dimensional data. In Fig. 11 we measured the number of I/Os required for the rebalancing operations during insertions of a total of $2 \times 128971 = 257942$ x-projections and of $2 \times 131461 = 262922$ x-projections from [Tiger1] & [Tiger2], respectively. In the same way, in Fig. 12 we measured the number of I/Os required for rebalancing operations during insertions of $2 \times 128971 = 257942$

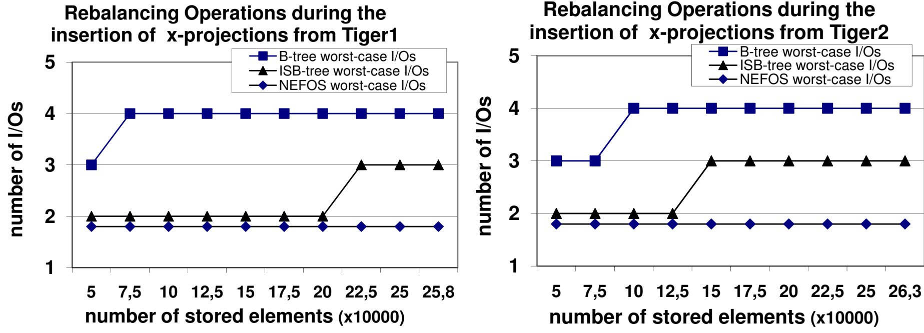

Fig. 11. Performance of rebalancing operations after an update for MBRs' x-projections of [Tiger1] (left) and [Tiger2] (right)

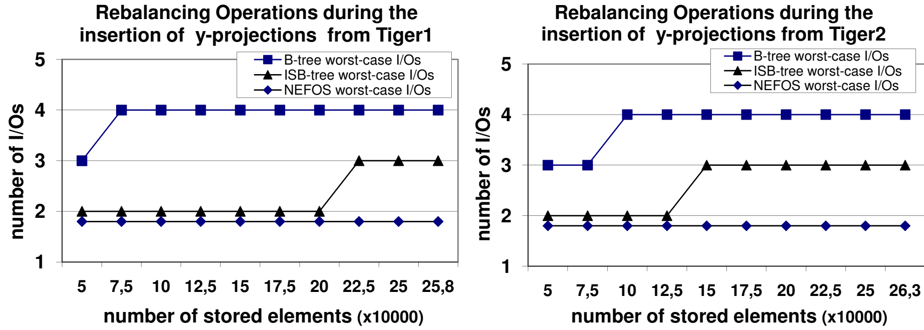

Fig. 12. Performance of rebalancing operations after an update for MBRs' y-projections of [Tiger1] (left) and [Tiger2] (right)

y-projections and of $2 \times 131461 = 262922$ y-projections from [Tiger1] & [Tiger2], respectively.

The above experiments show that NEFOS has approximately the same behaviour with ISB-tree requiring no more than 2 I/Os on average for both searching and rebalancing operations. This stems from the fact that the MBRs' projections from the data sets [Tiger1] & [Tiger2] follow an almost uniform distribution, due to the almost uniform decomposition of spatial maps. Better performance in [Tiger 2] is due to the fact that this is a dense spatial map and hence the derived one-dimensional data produce densely populated elements.

As a final remark, we note that there are applications with uniform key sizes larger than 8 bytes, resulting in a smaller value of B. The main example of such applications involve manipulation of strings. In this case, the size of the block may be as small as 2. Consequently, in such cases the NEFOS structure exhibits a much better performance.

5 Conclusions

We presented NEFOS (NEsted FOrest of balanced treeS), the first cache-aware indexing scheme, which supports expected w.h.p. sub-logarithmic range query processing for any (unknown) realistic input distribution. Moreover, NEFOS is the first concrete access method, which works for both I/O and RAM model, avoiding any kind of transformation or adaptation. The innovation of our solution was also verified by an accompanying experimental study. We leave for journal version a more detailed theoretical analysis as well as an exhaustive experimental evaluation of multi-dimensional exact-match and range queries.

References

1. Arge, L., de Berg, M., Haverkort, H.J., Yi, K.: The Priority R-Tree: A Practically Efficient and Worst-Case Optimal R-Tree. In: SIGMOD Conf., pp. 347–358 (2004)
2. Aggarwal, A., Vitter, J.S.: The Input/Output Complexity of Sorting and Related Problems. C. ACM 31(9), 1116–1127 (1988)
3. Andersson, A., Mattson, C.: Dynamic Interpolation Search in $o(\log \log n)$ Time. In: Lingas, A., Carlsson, S., Karlsson, R. (eds.) ICALP 1993. LNCS, vol. 700, pp. 15–27. Springer, Heidelberg (1993)
4. Beckmann, N., Krigel, H., Schneider, R., Seeger, B.: The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In: SIGMOD (1990)
5. Bayer, R., McCreight, E.: Organization of large ordered indexes. Acta Informatica 1, 173–189 (1972)
6. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. C. ACM 51, 107–113 (2008)
7. Dietz, P., Raman, R.: A constant update time finger search tree. Information Processing Letters 52, 147–154 (1994)
8. Fagin, R., Nievergelt, J., Pippinger, N., Strong, H.R.: Extendible Hashing-A fast access method for dynamic files. ACM Trans. Database Systems 4(3), 315–344 (1979)
9. Ferragina, P., Grossi, R.: The String B-tree: A New Data Structure for String Search in External Memory and Its Applications. Journal of the ACM 46(2), 236–280 (1999)
10. Fox, E., Chen, Q., Daoud, A.: Practical Minimal Perfect Hash Functions for Large Databases. C. ACM 35(5), 105–121 (1992)
11. Kaporis, A., Makris, C., Sioutas, S., Tsakalidis, A., Tsichlas, K., Zaroliagis, C.: Improved Bounds for Finger Search on a RAM. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 325–336. Springer, Heidelberg (2003)
12. Kaporis, A., Makris, C., Mavritsakis, G., Sioutas, S., Tsakalidis, A., Tsichlas, K., Zaroliagis, C.: ISB-Tree: A New Indexing Scheme with Efficient Expected Behaviour. In: Deng, X., Du, D.-Z. (eds.) ISAAC 2005. LNCS, vol. 3827, pp. 318–327. Springer, Heidelberg (2005)
13. Knuth, D.E.: Deletions that preserve randomness. IEEE Trans. Softw. Eng. 3, 351–359 (1977)
14. Lehman, P., Bing Yao, S.: Efficient Locking for Concurrent Operations on B-Trees. ACM Trans. Database Systems 6(4), 650–670 (1981)
15. Levcopoulos, C., Overmars, M.H.: Balanced Search Tree with $O(1)$ Worst-case Update Time. Acta Informatica 26, 269–277 (1988)

16. Litwin, W.: Linear Hashing: A new tool for files and tables addressing. In: International Conference on Very Large Databases, vol. 6, pp. 212–223 (1980)
17. Litwin, W., Lomet, D.: A New Method for Fast Data Searches with Keys. IEEE Software 4(2), 16–24 (1987)
18. Manolopoulos, Y., Theodoridis, Y., Tsotras, V.: Advanced Database Indexing. Kluwer Academic Publishers, Dordrecht (2000)
19. Mehlhorn, K., Tsakalidis, A.: Dynamic Interpolation Search. Journal of the ACM 40(3), 621–634 (1993)
20. Raman, R.: Eliminating Amortization: On Data Structures with Guaranteed Response Time. PhD Thesis, Dept. of Computer Science, University of Rochester, New York; Technical Report TR-439 (1992)
21. Seeger, B., Larson, P.A.: Multi-Disk B-trees. In: Proc. SIGMOD Conference, pp. 436–445 (1991)
22. Srinivasan, V., Carey, M.J.: Performance of B+ Tree Concurrency Algorithms. VLDB Journal 2(4), 361–406 (1993)
23. Theodoridis, Y.: The R-tree Portal (2003), `http://www.rtreeportal.org`, [Tiger1] and [Tiger2] data sets in `http://www.rtreeportal.org`
24. Vitter, J.S.: External memory algorithms and data structures: dealing with massive data. ACM Computing Surveys 33(2), 209–271 (2001)
25. Vitter, J.S., Shriver, E.A.M.: Optimal Algorithms for Parallel Memory I: Two-Level Memories. Algorithmica 12(2-3), 110–147 (1994)
26. Willard, D.E.: Searching Unindexed and Nonuniformly Generated Files in $\log \log N$ Time. SIAM Journal of Computing 14(4), 1013–1029 (1985)
27. Willard, D.E.: Examining Computational Geometry, van Emde Boas Trees, and Hashing from the Perspective of the Fusion Tree. SIAM Journal of Computing 29(3), 1030–1049 (2000)