ISB-Tree: A New Indexing Scheme with Efficient Expected Behaviour*

Alexis Kaporis[1], Christos Makris[1], George Mavritsakis[1], Spyros Sioutas[1], Athanasios Tsakalidis[1,2], Kostas Tsichlas[1], and Christos Zaroliagis[1,2]

[1] Dept of Computer Eng and Informatics, University of Patras, 26500 Patras, Greece
[2] Computer Technology Institute, N. Kazantzaki Str, Patras University Campus, 26500 Patras, Greece
{kaporis, makri, mayritsa, sioutas, tsak, tsihlas, zaro}@ceid.upatras.gr

Abstract. We present the interpolation search tree (ISB-tree), a new cache-aware indexing scheme that supports update operations (insertions and deletions) in $O(1)$ worst-case (w.c.) block transfers and search operations 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 a large class of (unknown) input distributions. The w.c. search bound of our indexing scheme is $O(\log_B n)$ block transfers. Our update and expected search bounds constitute a considerable improvement over the $O(\log_B n)$ w.c. block transfer bounds for search and update operations achieved by the B-tree and its numerous variants. This is also suggested by a set of preliminary experiments we have carried out. Our indexing scheme is based on an externalization of a main memory data structure based on interpolation search.

1 Introduction

More than three decades after its invention, B-tree [3, 4] 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 stable and guaranteed performance for search and update (insertion and deletion) operations, which both cost $O(\log_B n)$ block transfers in the worst-case, with B and n representing the number of elements in a disk block and the number of stored elements, respectively. 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. In such a query, the elements in a range $[z_1, z_2]$ can be found by first searching the B-tree for z_1 and then performing an in-order traversal

* This work was partially supported by the IST Programme (6th FP) of EC under contracts IST-2001-33058 (PANDA) and IST-2002-001907 (integrated project DELIS), and by the Action PYTHAGORAS of the Operational Programme for Educational & Vocational Training II, with matching funds from the European Social Fund and the Greek Ministry of Education.

X. Deng and D. Du (Eds.): ISAAC 2005, LNCS 3827, pp. 318–327, 2005.

in the tree from z_1 to z_2. These bounds hold for any cache-aware disk-access model, that is, a model that accounts memory transfers in disk blocks, as these transfers are the dominating operation w.r.t. time. In this paper, we consider one of the most known and widely used such models, namely the two-level memory hierarchy model introduced in [1]. 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 vast number of variants of the B-tree have been proposed since its appearance — B$^+$-trees [4] and B*-trees [4, 11] are some representative examples; see the excellent survey by Vitter [20] for an extended accounting of these and other variants and their applications — in order to improve its performance in practice for various applications, to make it parallel for use in multi-disk environments [18], to tune it for concurrency and recovery purposes [19], to extend it to cover other than the original field [6], etc. However, to the best of our knowledge, the aforementioned search and update bounds of B-tree and its variants remained untouched all these years. The same applies to the one-dimensional range search query bound, although some variants (with B$^+$-tree being the most popular) offer a slightly different procedure, since the leaves are linked together and hence allow for sequential access. Regarding the update operation, it should be noted that it consists of three consecutive phases: a search phase (to locate the element), a value-updating phase (to replace the element's key with its new value), 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 value-updating phase takes typically $O(1)$ block transfers (and/or time). In the case of B-tree and its variants, the rebalancing phase requires $O(\log_B n)$ block transfers in the worst-case. This implies that the update operation takes $O(\log_B n)$ block transfers, even in the case where the update position (block within which the update will take place) is given. Note that there are certain applications (see e.g., [13]) which justify the exclusion of the search phase in an update operation: once the requested element has been found, then the next element to be searched is located "near by" and hence a new search is redundant.

In this work, we present a new indexing scheme, called *ISB-tree* (Interpolation Search B-tree), that 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) that are explained 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 w.h.p.. The worst-case block transfers for the search operation in our indexing scheme are $O(\log_B n)$.

To achieve our expected search bound we consider a rather general scenario of μ-*random* insertions and *random* deletions, where μ is a so-called *smooth* probability density [2, 16]. 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 (see [12]). 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 [21]).

Our indexing scheme is a two-level data structure. The upper level is an externalization of the static interpolation search tree presented in [9]. The lower level is a forest of buckets, each of which is implemented by a new variant of the classical B-tree, the *Lazy B-tree*, which is introduced in [10]. The lazy B-tree supports a search operation in $O(\log_B n)$ block transfers and an update operation in $O(1)$ block transfers, provided that the update position is given. However, a straightforward combination of the above structures does not necessarily lead to better bounds, since: (i) the number of elements within a bucket may grow arbitrarily large, as insertions are performed; and (ii) we strive for creating a *robust indexing scheme*, that is, a data structure that works correctly *without* apriori knowledge of the particular smooth distribution μ. To overcome these problems, we employ the combinatorial game of bins and balls introduced in [9] that allows to upper bound the number of elements in a bucket, and to approximate an unknown distribution by an almost uniform one.

To the best of our knowledge, this is the first work that uses the dynamic interpolation search paradigm in the framework of indexing data in external memory. External data structures related to our approach are those based on hashing [11, 15, 20]. The main representatives of external memory hashing methods are: extendible hashing [5], linear hashing [14], and external perfect hashing [7]. 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. The main result of this paper, the *ISB-tree*, with the complexity analysis of its operations is discussed in Section 3. Section 4 provides an experimental evaluation of our theoretical findings. We conclude in Section 5. Due to space limitations the full details of the paper can be found in [10].

2 Preliminaries

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. If there is room for the new element in l, then we simply store it there. Otherwise, we split l into two leaves l' and l'' of approximately the same size and insert the new element in the relevant leaf. The split of l results in the insertion of a new routing element in the parent of l, and thus the split may propagate up the tree. Propagation of splits can often be avoided by sharing some of the (routing) elements of the full node with a non-full sibling. A new (degree 2) root is produced when the root splits and the height of the tree grows by one. 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. If this results in l containing too few elements, then we either fuse it with one of its siblings (corresponding to deleting l and inserting its elements in a sibling), or we perform a share operation by moving elements from a sibling to l. Fuse operations may also propagate up the tree and eventually result in the height of the tree decreasing by one.

One of the first works, in the context of internal memory data structures, that investigated non-uniform distributions regarding insertions in an update sequence was that of Willard [21], who introduced the so-called *regular* distributions. A probability density μ is regular if there are constants b_1, b_2, b_3, b_4 such that $\mu(x) = 0$ for $x < b_1$ or $x > b_2$, and $\mu(x) \geq b_3 > 0$ and $|\mu'(x)| \leq b_4$ for $b_1 \leq x \leq b_2$. This has been further pursued by Mehlhorn and Tsakalidis [16], who introduced the *smooth* input distributions, a notion that was further generalized and refined in [2]. Given two functions f_1 and f_2, a density function $\mu = \mu[a, b](x)$ is (f_1, f_2)-*smooth* [2] if there exists a constant β, such that for all c_1, c_2, c_3, $a \leq c_1 < c_2 < c_3 \leq b$, and all integers n, it holds that

$$\int_{c_2 - \frac{c_3 - c_1}{f_1(n)}}^{c_2} \mu[c_1, c_3](x)dx \leq \frac{\beta \cdot f_2(n)}{n}$$

where $\mu[c_1, c_3](x) = 0$ for $x < c_1$ or $x > c_3$, and $\mu[c_1, c_3](x) = \mu(x)/p$ for $c_1 \leq x \leq c_3$ where $p = \int_{c_1}^{c_3} \mu(x)dx$. 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 [2, 9]. Actually, *any* probability distribution is $(f_1, \Theta(n))$-smooth, for a suitable choice of β.

The *static interpolation search tree* [9] is a static, explicit, and refined version of the search trees used in [2, 16]. A static interpolation search tree containing n elements can be fully characterized by three nondecreasing functions $H(n)$, $R(n)$ and $I(n)$, where $H(n)$ denotes the height of the tree, $R(n)$ denotes the out-degree of the root, and $I(n)$ denotes how fine is the partition of the set of elements and is defined by $I(n) = n \cdot g(H(n))$, where $g(n)$ should satisfy $\sum_{i=1}^{\infty} g(i) = \Theta(1)$.

To guarantee the height of $H(n)$, it should hold that $n/R(n) = H^{-1}(H(n)-1)$. The children of the root have $n' = \Theta(n/R(n))$ leaves. Their height will be $H(n') = H(n)-1$, their out-degree is $R(n') = \Theta(H^{-1}(H(n)-1)/H^{-1}(H(n)-2))$, and $I(n') = n' \cdot g(H(n'))$. In general, consider an internal node v at depth i and assume that n_i leaves are stored in the subtree rooted at v. Then we have that $R(n_i) = \Theta(H^{-1}(H(n)-i+1)/H^{-1}(H(n)-i))$, and $I(n_i) = n_i \cdot g(H(n)-i)$. The node v is associated with an array REP$[1..R(n_i)]$ of sample elements, one sample element for each of its subtrees, and an ID$[1..I(n_i)]$ array that stores a set of sample elements approximating the inverse distribution function. By using the ID array, we can interpolate the REP array to determine the subtree in which the search procedure will continue. In particular, the ID array for node v is an array ID$[1..m]$, where $m = I(n_i)$, with ID$[i] = j$ iff REP$[j] < \ell + i(u-\ell)/m \leq$ REP$[j+1]$, where ℓ and u are the minimum and the maximum element, resp., stored in the subtree rooted at v. Let x be the element we seek. To interpolate REP, compute the index $j =$ ID$[\lfloor (m(x-\ell)/(u-\ell))\rfloor]$, and then search the REP array from REP$[j+1]$ until the appropriate subtree is located. For each node we explicitly maintain parent and child pointers. The required pointer information can be easily incorporated in the construction of the static interpolation search tree. Throughout the paper, we say that an event E occurs *with high probability* if $\Pr[E] = 1 - o(1)$.

3 The ISB-Tree

The ISB-tree is a two-level data structure. The lower level is a set of buckets each of which contains a subset of the stored elements and is represented by a unique representative. The representatives of the buckets are stored in the upper level structure.

The upper level data structure is an external version of the static interpolation search tree (SIST) described in [9] (see also Section 2), with parameters $R(s) = s^{\delta}$, $I(s) = s/(\log\log s)^{1+\epsilon}$, where $\epsilon > 0$, $\delta = 1 - \frac{1}{B}$, and s is the number of stored elements in the tree. The specific choice of δ guarantees the desirable $O(\log_B \log s)$ height of the upper level structure. For each node that stores more than $B^{1+\frac{1}{B-1}}$ elements in its subtree, we represent its REP and ID arrays as static external sorted arrays; otherwise, we store all the elements in a constant number of disk blocks. In particular, let v be a node and n_v be the number of stored elements in its subtree, with $n_v \geq B^{1+\frac{1}{B-1}}$. Node v is associated with two external arrays EREP$_v$ and EID$_v$ that represent the REP$_v$ and ID$_v$ arrays of the original SIST structure. The EID$_v$ array uses $O(\frac{I(n_v)}{B})$ contiguous blocks, the EREP$_v$ array uses $O(\frac{R(n_v)}{B})$ contiguous blocks, while an arbitrary element of the arrays can be accessed with $O(1)$ block transfers, given its index. Moreover, the choice of the parameter $B^{1+\frac{1}{B-1}}$ guarantees that each of the EREP$_v$ and EID$_v$ arrays contains at least B elements, and hence we do not waste space (in terms of underfull blocks) in the external memory representation.

On the other hand, the lower level is a set of ρ buckets. Let S_0 be the set of elements to be stored where the elements take values in $[a, b]$. Each bucket

$\mathcal{B}_i$, $1 \leq i \leq \rho$, stores a subset of elements and is represented by the element $rep(i) = \max\{x : x \in \mathcal{B}_i\}$. The set of elements stored in the buckets constitute an ordered collection $\mathcal{B}_1, \ldots, \mathcal{B}_\rho$ such that $\max\{x : x \in \mathcal{B}_i\} < \min\{y : y \in \mathcal{B}_{i+1}\}$ for all $1 \leq i < \rho - 1$. In other words, $\mathcal{B}_i = \{x : x \in (rep(i-1), rep(i)]\}$, for $2 \leq i \leq \rho$, and $\mathcal{B}_1 = \{x : x \in [rep(0), rep(1)]\}$, where $rep(0) = a$ and $rep(\rho) = b$.

The elements of each $\mathcal{B}_i$ are stored in a *Lazy B-tree*, which is a new variant of the classical B-tree. Due to space limitations, the details of the Lazy B-tree are discussed in the extended version of the paper [10]. The following theorem summarizes the properties of a Lazy B-tree.

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

The ISB-tree is maintained by incrementally performing global reconstructions [17]. Let S_0 be the set of stored elements at the latest reconstruction, and assume that $S_0 = \{x_1, \ldots, x_{n_0}\}$ in sorted order. The reconstruction is performed as follows. We partition S_0 into two sets S_1 and S_2, where $S_1 = \{x_{i \cdot \ln n_0} : i = 1, \ldots, \frac{n_0}{\ln n_0} - 1\} \cup \{b\}$, and $S_2 = S_0 - S_1$. The i-th element of S_1 is the representative $rep(i)$ of the i-th bucket $\mathcal{B}_i$, where $1 \leq i \leq \rho$ and $\rho = |S_1| = \frac{n_0}{\ln n_0}$. The representatives $rep(i)$, $1 \leq i \leq \rho$, are stored in the external SIST (note that there is no need to store $rep(0)$). An element $x \in S_2$ is stored in $\mathcal{B}_i$, iff $rep(i-1) < x \leq rep(i)$, for $i \in \{2, \ldots, \frac{n_0}{\ln n_0}\}$; otherwise ($x \leq rep(1)$), x is stored in $\mathcal{B}_1$. The same condition holds for every new element inserted in the structure. In order to insert/delete an element, given the position (block) of the update, we simply have to insert/delete the element to/from the *Lazy B-tree* storing the elements of the corresponding bucket. Each time the number of updates exceeds cn_0, where c is a predefined constant, the whole data structure is reconstructed. Let n be the number of stored elements at this time. After the reconstruction, the number of buckets is equal to $\lceil \frac{n}{\ln n} \rceil$.

The search procedure for locating a query element x can be decomposed into two phases: (i) the traversal of internal nodes of the external SIST locating a bucket $\mathcal{B}_i$, and (ii) the search for x in the *Lazy B-tree* storing the elements of $\mathcal{B}_i$. Phase (i) starts from the root of the external SIST. It checks the external arrays on the root and by interpolating it decides into which child the search procedure will continue. More specifically, let v be a node in the search path for query element x, n_v be the number of leaves in its subtree, and let l_v and u_v be the minimum and the maximum element, resp., stored in the subtree rooted at v. As we have already mentioned, node v is associated with two external arrays EREP_v and EID_v that implement the REP_v and ID_v arrays of the SIST definition. To interpolate, we compute the value $i = \lfloor \frac{x - l_v}{u_v - l_v} R(n_v) \rfloor$ and find the index $j = \text{EID}_v[i]$, by retrieving the $\lceil \frac{i}{B} \rceil$-th block of the EID_v array. We then scan the blocks of the EREP_v array, starting from its $\lceil \frac{j}{B} \rceil$-th block, until locating an index l such that $\text{EREP}_v[l] \leq x < \text{EREP}_v[l+1]$. If the l-th son of v is not a bucket, then we continue recursively in the same manner in the l-th son of v, until we locate the representative of a bucket $\mathcal{B}_i$. In this case, the search

procedure is concluded by entering phase (ii) and by searching further in the *Lazy B-tree* of the bucket $\mathcal{B}_i$.

In the following, we will analyze the bounds of the search and update operations. Our result 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 [2, 9], and is stated by the following theorem.

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* $\epsilon > 0$, $\delta = 1 - \frac{1}{B}$, $s_0 = \frac{n_0}{\ln n_0}$ *and* 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. (sketch) As we have already mentioned, the search operation in the ISB-tree can be decomposed into two basic steps: (i) the traversal of internal nodes of the external SIST, and (ii) the search for x in the *Lazy B-tree* in the bucket that we located from step (i).

We can prove (in a way similar to that in the proof of [9–Theorem 1]) that the expected number of block transfers for step (i) is $O(h)$ w.h.p, where h is the height of the external SIST. The main point of the proof is that the expected number of blocks in the EREP_v array, which we need to linearly scan when interpolating at a node v, is $O(1)$ w.h.p. Since in our case, the height of the tree is $h = O(\log_B \log s_0)$, where s_0 is the number of stored elements at the latest reconstruction and $s_0 = O(n)$, we get that the expected number of block transfers for step (i) is $O(\log_B \log n)$ w.h.p.

Regarding the complexity of step (ii), we can use the same combinatorial game of balls and bins introduced in [9] and prove (similarly to [9–Theorem 6]) that w.h.p. the expected number of elements in each bucket is $O(\log n)$, when elements are μ-randomly inserted and randomly deleted, and μ is an unknown smooth probability density. Since we store the elements of each bucket in a lazy B-tree, we get from Theorem 1 that the block transfers of step (ii) are also $O(\log_B \log n)$. Consequently, the total expected complexity of the search procedure is bounded by $O(\log_B \log n)$ block transfers w.h.p.

Let us now consider the update bound. Between reconstructions the block transfers for an update are clearly $O(1)$, since we only have to update the appropriate Lazy B-tree which can be done in $O(1)$ block transfers (cf. Theorem 1). The reconstructions can be easily handled by using the technique of global rebuilding [13]. With this technique the linear work spent during a global reconstruction of the upper level structure may be spread out on the updates in such a way that a rebuilding cost of $O(1)$ block transfers is spent at each update.

Finally, the worst-case search complexity of $O(\log_B n)$ block transfers can be achieved by using two data structures, an ISB-tree and a Lazy B-Tree, and hence storing each element twice. A search for a query element is performed

by searching simultaneously both structures and terminating the search when locating for the first time the sought element. The worst-case update and space complexity remain asymptotically unaffected and so the theorem is proven. □

4 Experimental Evaluation

In this section, we investigate the practical merits of the ISB-tree. 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. We compared the implementation of the ISB-tree with that of a B-tree on the same data sets (implementations were carried out in `C++`). 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. 1. 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 experiments revealed that the expected number of block transfers in the ISB-tree remains constant even for gigantic data sets (Terabytes - TB). Moreover, for data sets larger than 100 GB, the expected number of block transfers is reduced by a factor ranging from 1/3 (for normal and Gaussian distributions) to 1/2 (for uniform and regular distributions) compared with the B-tree. This behaviour is justified by the time complexity of the search operation and by the fact that for data sets up to 1 TB and block size of 341 elements, the ISB-tree is a two level structure, where the first level (SIST structure) consists of only one node equipped with the appropriate EID and EREP arrays, while the second level (lazy B-tree) consists of only one block of elements. Thus, we need 2 block transfers in the first level (one for each array) and 1 block transfer in the second level. Our experiments also show that for uniform and regular distributions, the position of EREP array (which has been located by its corresponding entry in EID) points in almost all cases to the correct subset within which the search has to be continued in the second level. For the case of normal and Gaussian distributions, we often had to move to the immediately next block and this adds one additional block transfer to the search operation. Naturally, for small data sets (smaller than 10 MB), our data structure becomes less efficient than B-trees, due to the overhead of the two-level structure.

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, we expect that in such cases the ISB-tree will have a much better performance.

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

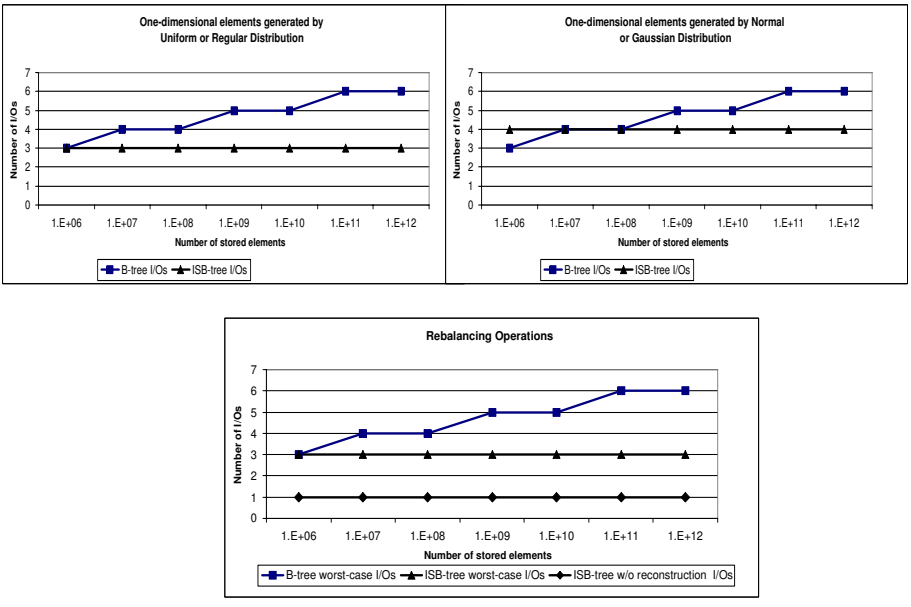

Fig. 1. Search performance for uniform and regular distributions (upper left) and normal and Gaussian distributions (upper right). Block transfers of rebalancing operations after an update (bottom).

sequences of length $n_0/2$ and $2n_0$. The data structure is reconstructed every n_0 operations (i.e., we chose $c = 1$). Our experimental results are reported in Fig. 1. The values represent averages over the smaller update sequence (where no reconstruction occurs) and the larger one (where a reconstruction indeed occurs). We have observed that both numbers of rebalancing operations are independent of the distribution.

5 Conclusions

We presented a new indexing scheme, the ISB-tree, that supports update operations in $O(1)$ worst-case block transfers and search operations in $O(\log_B \log n)$ expected block transfers w.h.p. for a large class of input distributions. The ISB-tree is innovative in the sense that it shoots down for the first time the optimal $O(\log_B n)$ block transfer bound of B-tree and its variants when the updates are drawn from a large class of input distributions. Its analysis is based on the traditional I/O model of [1], but we feel that it can also be implemented in the cache-oblivious model [8] with the same complexities.

Acknowledgements. We thank Y. Theodoridis for several interesting discussions.

References

1. A. Aggarwal and J.S. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116-1127, 1988.
2. A. Andersson and C. Mattson. Dynamic Interpolation Search in $o(\log \log n)$ Time. *In Proc. 14th International Colloquium on Automata, Languages and Programming (ICALP)*. LNCS 700, pp. 15-27, 1993.
3. R. Bayer and E. McCreight. Organization of large ordered indexes. *Acta Informatica*, 1:173-189, 1972.
4. D. Comer. The Ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121-137, 1979.
5. R. Fagin, J. Nievergelt, N. Pippinger, H.R. Strong. Extendible Hashing-A fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315-344, 1979.
6. P. Ferragina and R. Grossi. 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.
7. E. Fox, Q. Chen, A. Daoud. Practical Minimal Perfect Hash Functions for Large Databases. *Communications of the ACM*, 35(5):105-121, 1992.
8. M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *In Proc. 40th IEEE Symp. on Foundations of Computer Science (FOCS)*, pp. 285-297, 1999.
9. A. Kaporis, C. Makris, S. Sioutas, A. Tsakalidis, K. Tsichlas, and C. Zaroliagis: Improved Bounds for Finger Search on a RAM. *In 11th Ann. European Symp. on Algorithms (ESA)*, *LNCS* Vol. 2832, pp. 325-336. Full version as Tech. Rep. TR-2003/07/01, Computer Technology Institute, Patras, July 2003.
10. A. Kaporis, C. Makris, G. Mavritsakis, S. Sioutas, A. Tsakalidis, K. Tsichlas and C. Zaroliagis. ISB-Tree: A New Indexing Scheme with Efficient Expected Behaviour. Computer Technology Institute Tech. Report TR 2005/09/02, September 2005.
11. D.E. Knuth. *Sorting and Searching*, Vol. 3 of *The Art of Computer Programming*, Addison-Wesley, 1973.
12. D.E. Knuth. Deletions that preserve randomness. *IEEE Trans. Softw. Eng.*, 3:351-359, 1977.
13. C. Levcopoulos and M.H. Overmars: Balanced Search Tree with $O(1)$ Worst-case Update Time. *Acta Informatica*, 26:269-277, 1988.
14. W. Litwin. Linear Hashing: A new tool for files and tables addressing. In *In Proc. Int. Conf. on Very Large Databases (VLDB)*, 6:212-223, 1980.
15. Y. Manolopoulos, Y. Theodoridis, V. Tsotras. *Advanced Database Indexing*. Kluwer Academic Publishers, 2000.
16. K. Mehlhorn and A. Tsakalidis. Dynamic Interpolation Search. *Journal of the ACM*, 40(3):621-634, 1993.
17. M. Overmars, J. van Leeuwen. Worst Case Optimal Insertion and Deletion Methods for Decomposable Searching Problems. *Information Processing Letters*, 12(4):168-173.
18. B. Seeger and P.A. Larson. Multi-Disk B-trees. *In Proc. SIGMOD Conference*, pp. 436-445, 1991.
19. V. Srinivasan and M.J. Carey. Performance of B+ Tree Concurrency Algorithms. *VLDB Journal*, 2(4):361-406, 1993.
20. J.S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209-271, 2001.
21. D.E. Willard. Searching Unindexed and Nonuniformly Generated Files in $\log \log N$ Time. *SIAM Journal of Computing*, 14(4):1013-1029, 1985.