B-trees and Cache-Oblivious B-trees with Different-Sized Atomic Keys

Michael A. Bender, Department of Computer Science, Stony Brook University
Roozbeh Ebrahimi, Google Inc.
Haodong Hu, School of Information Management and Engineering, Shanghai University of Finance and Economics
Bradley C. Kuszmaul, MIT CSAIL

Most B-tree papers assume that all N keys have the same size K, that $f = B/K$ keys fit in a disk block, and therefore that the search cost is $O(\log_{f+1} N)$ block transfers. When keys have variable size, B-tree operations have no nontrivial performance guarantees, however.

This paper provides B-tree-like performance guarantees on dictionaries that contain keys of different sizes in a model in which keys must be stored and compared as opaque objects. The resulting ***atomic-key dictionaries*** exhibit performance bounds in terms of the average key size and match the bounds when all keys are the same size. Atomic key dictionaries can be built with minimal modification to the B-tree structure, simply by choosing the pivot keys properly.

This paper describes both static and dynamic atomic-key dictionaries. In the static case, if there are N keys with average size K, the search cost is $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$ expected transfers. It is not possible to transform these expected bounds into worst-case bounds. The cost to build the tree is $O(NK)$ operations and $O(NK/B)$ transfers if all keys are presented in sorted order. If not, the cost is the sorting cost.

For the dynamic dictionaries, the amortized cost to insert a key κ of arbitrary length at an arbitrary rank is dominated by the cost to search for κ. Specifically the amortized cost to insert a key κ of arbitrary length and random rank is $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N + |\kappa|/B)$ transfers. A dynamic-programming algorithm is shown for constructing an search tree with minimal expected cost.

This paper also gives a cache-oblivious static atomic-key B-tree, which achieves the same asymptotic performance as the static B-tree dictionary, mentioned above. A cache-oblivious data structure or algorithm is not parameterized by the block size B or memory size M in the memory hierarchy; rather, it is universal, working simultaneously for all possible values of B or M. On a machine with block size B, if there are N keys with average size K, search operations costs $O(\lceil K/B \rceil \log_{1+\lceil B/K \rceil} N)$ block transfers in expectation. This cache-oblivious layout can be built in $O(N \log(NK))$ processor operations.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sorting and Searching*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Sorting and Searching*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: B-tree with different-sized keys, atomic keys, dynamic programming, cache-oblivious B-tree with different-sized keys

---

A preliminary version of this article appeared in [Bender et al. 2010].
Supported in part by NSF Grants ACI-0324974, CCF-0540897/05414009, CCF-0541209, CCF-0621439/0621425, CCF-0621511, CCF-0634793/0632838, CCF-0937822, CCF-0937860, CCF-1162148, CCF-1217708, CCF-1314547, CCF-1439084, CCF-1617618, CNS-0540248, CNS-0615215, CNS-0627645, CNS-1017058, CNS-1408695, CNS-1409238, IIS-1247726, IIS-1251137, DOE Grant DE-FG02-08ER25853, the Singapore-MIT Alliance, and Sandia National Laboratories.

DOI: http://dx.doi.org/10.1145/2907945

ACM Reference Format:
ACM Trans. Datab. Syst. 9, 4, Article 39 (March 2016), 33 pages.
DOI: http://dx.doi.org/10.1145/2907945

1. INTRODUCTION

Most published descriptions of B-trees (which for this discussion include common variants, such as B^+-trees) assume that all keys have the same size. If all N keys have size K, and $f = B/K$ keys fit in a disk block, then the search cost is $O(\log_{f+1} N)$ block transfers.

However, for over four decades most production-quality B-trees (e.g., those in databases and file systems) have supported variable-size keys. The basic search, insert, and delete operations all work correctly when keys have variable sizes, but the operations no longer have nontrivial performance guarantees. Roughly speaking, it is better to use small keys as pivots near the top of the tree because small keys means a larger branching factor and a more efficient search. However, one cannot simply choose to put the smallest keys in the root node because this choice may do a poor job of dividing the search space evenly.

Most B-tree rebalancing algorithms do not attempt to choose small pivot keys. As a result, they generally do not provide (nontrivial) performance guarantees.

For example, if a B-tree stores a mix of unit-sized keys and $\Theta(B)$-sized keys, a search may use $O(\log_{B+1} N)$ or $O(\log_2 N)$ memory transfers, depending on which keys are chosen for pivots. In particular, if most nodes in a root-to-leaf path employ small pivots with a branching factor of $\Theta(B)$ then the search cost is $O(\log_{B+1} N)$, but if most nodes employ large pivots with a branching factor of $O(1)$ then the search cost is $O(\log_2 N)$.

To demonstrate that real B-trees can suffer performance degradation by choosing pivot keys poorly, we created two Berkeley DB B-Trees, a "good" tree and a "bad" tree, each containing one million key-value pairs. We disabled the prefix compression since we wanted the keys to be atomic. The keys were mostly 4 bytes long, but $1/128$ of the keys were 999 bytes long. The keys were inserted in increasing order (from left to right). In the "good" tree, the long keys were not used as pivots and so the tree had fanout of 225 and a depth of 3, whereas in the "bad" tree, the long keys were used as pivots, so the tree had fanout 5 and a depth of 8. It was a simple matter to force the system to choose bad pivots, since the splitting algorithm is predictable. The keys with rank equal to $63 \pmod{128}$ get used as a pivot.

Empirical experience and folk wisdom suggest that, although there may not be formal guarantees, the data structure works pretty well most of the time. Nonetheless, in the worst-case B-tree performance can suffer because of a small number of large records.

Atomic-Key Dictionaries

This paper explores B-tree variations that have performance guarantees even when keys have variable sizes. Our objective is to modify the traditional B-tree as little as possible. We want the same basic search algorithm based on a k-ary tree and a simple rebalancing scheme.

In this paper we study ***atomic-key dictionaries***. The essential feature of an atomic-key dictionary is that a key is stored and manipulated as an atomic or opaque object. Keys cannot be split up, and so whenever a key is stored in the data structure, the entire key must be stored. Whenever the system compares keys are ordering, two entire keys must be brought into memory and compared. The data structure does not know anything about the internals of the keys (except for their size) or about the comparison function.

We analyze the performance using the Disk Access Machine (DAM) model [Aggarwal and Vitter 1988], which provides an internal memory of size M and an arbitrarily large external memory. A DAM transfers blocks of size B, each transfer incurring unit cost.

We are interested in performance guarantees for search, insert, and delete operations, and these guarantees should be parameterized by the average key size. The natural bound to strive for is $O(\log_f N)$ memory transfers, where f is the average number of elements that fit in a disk block. (Thus, f is the block size B divided by the average key length.) This is a natural bound because it matches the B-tree performance when all keys have the same size of B/f. Unfortunately, as we show in this paper, the bound is provably unattainable in general for atomic-key B-trees.[1]

Instead, this paper shows how to achieve these asymptotic bounds in expectation for both static and dynamic dictionaries. Thus, for a given number of elements and a total data size, the structure performs at least as well in expectation as if all the elements had the same size.

Alternatives for Supporting Variable-Size Keys

When the keys are strings the problem can be solved faster. For example, a ***string B-tree*** [Ferragina and Grossi 1999] can insert, delete or search for a key κ using $O(|\kappa|/B + \log_{B+1} N)$ transfers. This performance is optimal in the sense that the first term represents the cost to read a key and the second term represents the cost to search when all keys have unit size. It is also superior to what one can achieve with an atomic-key dictionary.

The string B-tree is not a solution to the problem posed in this paper, however. It has a different type of search algorithm from that used in a B-tree, and it is not an atomic-key dictionary. String keys can be chopped up and stored in different places and different parts of the key can be compared at different times. String dictionaries can achieve better asymptotic performance than atomic-key dictionaries.

Most dynamic dictionaries employ industrial-strength B-trees, not string B-trees. The ubiquity of B-trees helps justify why we strive to give provable bounds for variable-size keys.

Some production-quality B-trees are not atomic-key dictionaries because they also employ some variant of ***front compression*** [Clark et al. 1969; Wagner 1973; Bayer and Unterauer 1977], commonly called ***prefix compression***. In front compression if κ and κ' are two contiguous keys in a node and they share a common prefix of length ℓ, then κ' is encoded by the length ℓ of the shared prefix along with the unshared suffix. A set of front-compressed keys consumes the same space as the uncompacted trie of those keys [Knuth 1973].

Our work is incomparable to front compression or other compression techniques. Even in a tree with front compression, one can sometimes encounter long keys that propagate up the tree but do not share much prefix with their up-propagated cousins. On the other hand, there are situations where prefix compression saves more space and therefore gives more fanout than the techniques introduced in this paper. So both techniques are likely to be needed.

Results

Our results can be summarized as follows.

[1]Interestingly, the reference manual to Oracle Berkeley DB [Oracle 2009] claims that Berkeley DB achieves these target bounds, an indication that users may want them. Unfortunately, these bounds cannot be achieved in general when Berkeley DB is used as an atomic-key dictionary.

Static atomic-key B-tree. We show how to build a *static* atomic-key B-tree. With this construction, for a dictionary storing N keys of average size $\bar{K}$, the expected cost to search for a random key in the tree is $O(\lceil \bar{K}/B \rceil \log_{1+\lceil B/\bar{K} \rceil} N)$ memory transfers when keys are searched with uniform probability. Building the tree uses $O(N\bar{K})$ operations and $O(N\bar{K}/B)$ memory transfers.

To understand why this bound achieves our objective of searching for different-size keys with the same expected cost as same-size keys, plug in several values for the average key size $\bar{K}$. If $\bar{K} = O(1)$, then the expected search cost is $O(\log_{B+1} N)$, the performance for a B-tree storing unit-size keys. On the other hand, if $\bar{K} = O(B)$, then the expected search cost is $O(\log_2 N)$, which is what we expect if all keys have size $O(B)$ and the branching factor is constant. If the average key size is $\bar{K} = \Omega(B)$, then the branching factor is constant, but nodes can span many blocks, leading to an expected search cost of $O((\bar{K}/B) \log_2 N)$.

We prove that it is not possible to guarantee these bounds for arbitrary searches. This impossibility result helps explain one deviation from the structure of traditional B-trees. In traditional B-trees, all leaves reside at the same depth, whereas in atomic-key B-trees, we allow leaves to reside at different depths.

Dynamic atomic-key B-tree. We show how to build a *dynamic* atomic-key B-tree in which the expected cost to search for random keys matches that for static trees, the amortized cost to insert or delete a random key κ is $O(\lceil \bar{K}/B \rceil \log_{1+\lceil B/\bar{K} \rceil} N + |\kappa|/B)$, and the amortized cost to insert or delete an arbitrary key κ is dominated by the search cost.

Optimal static atomic-key B-tree. We present an $O(BN^3)$-operation dynamic program for constructing a static atomic-key dictionary with minimal expected search cost. For simplicity we present a version for which each key fits in a block. The algorithm takes as input the keys $\kappa_1, \ldots, \kappa_N$, their sizes, and their search probabilities $p_1, \ldots, p_N$. Unlike in the earlier results, here the search probabilities need not be equal.

Cache-oblivious static atomic-key B-tree. We present a cache-oblivious static atomic-key B-tree that achieves the same asymptotic performance as the static B-tree above. Specifically, on a machine with block size B, if there are N keys with average size $\bar{K}$, the expected search cost in our cache-oblivious static atomic-key B-tree is $O(\lceil \bar{K}/B \rceil \log_{1+\lceil B/\bar{K} \rceil} N)$ when all keys are searched with equal probability.

By ***cache-oblivious*** [Frigo et al. 1999; 2012; Prokop 1999], we mean that the algorithm or data structure has no memory-hierarchy-specific parameterization. Thus, an optimal-cache-oblivious algorithm or data structure is not parameterized by any block size, cache or RAM size, or memory- or disk-access times. Remarkably, many problems have optimal (and practical) cache-oblivious solutions. In particular, there is a series of papers on cache-oblivious static and dynamic B-tree [Prokop 1999; Bender et al. 2000; Brodal et al. 2002; Bender et al. 2005; Bender et al. 2002; 2004] and string B-trees [Brodal and Fagerberg 2006; Bender et al. 2006].

To build this data structure, we rely upon a family of B-trees for several carefully chosen values of block-size B, which all have the same parent-child topology but a different "layout" in memory. We show how to construct this family of trees. Then we prove that the trees in this family have the same performance as the static B-tree structures described above. Finally, we call upon the "split-and-refine" technique from [Alstrup et al. 2002] to collapse this family into a single cache-oblivious structure. This cache-oblivious layout can be built in $O(N \log(N\bar{K}))$ processor operations.

Outline

The rest of this paper is organized as follows. Section 2 describes the greedy structure of a static tree by giving a greedy strategy for selecting the root of the tree. Section 3 analyzes the search cost for static trees. Section 4 shows how to build a static tree efficiently, by proving that the greedy strategy from Section 2 can be realized with a small number of operations and I/Os. Section 5 shows how to build dynamic trees. Section 6 presents a dynamic program for building an optimal static search tree. Section 7 exhibits how to build a fixed-topology atomic-key B-tree, analyzes its performance, and then uses a family of fixed-topology atomic-key B-trees to achieve a cache-oblivious static B-tree layout. Section 8 discusses work related to atomic B-trees. and Section 9 concludes with a brief discussion of open problems.

2. STATIC ATOMIC-KEY B-TREE

In this section give the structure of a static atomic-key B-tree. We do so by giving a greedy algorithm for constructing a static atomic-key B-tree on N different-size keys. The idea is to store small keys near the top of the tree and big keys near the bottom while simultaneously choosing pivot keys that are spread out in the key-space. (In Section 4 we give an efficient realization of this algorithm that uses few I/O.)

We first give notation. Denote the average length of the N keys, $\{\kappa_i\}$, by $\bar{K}$. No key is larger than a constant fraction of memory. For block size B, define

$$f = \max\left\{3, \left\lfloor \frac{B}{\bar{K}} \right\rfloor\right\}. \tag{1}$$

Building the Root Node

Here we give a greedy algorithm for creating the root node, given a set of N keys in sorted order. Divide the keys into f sets $\{\mathcal{C}_i\}_{0\le i\le f-1}$ so that each set contains N/f keys. The first set $\mathcal{C}_0$ contains the smallest N/f keys, the second set $\mathcal{C}_1$ contains the next smallest N/f keys, and so on. For each set, except for the first and the last, we pick the ***representative key*** r_i to be the minimum-length key in each set; we do not need representatives from the first and the last sets.

We assign these $f-2$ representatives $\{r_i\}_{1\le i\le f-2}$ to be the pivot keys in the root node of the static atomic-key B-tree. The root node therefore has $f-1$ children, and we recursively use the greedy algorithm to assign pivot keys to the children nodes.

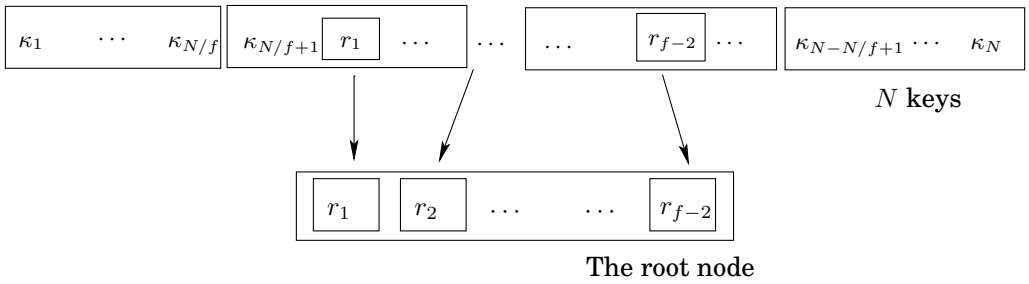

Fig. 1. The greedy algorithm for the root node of a static tree layout.

In the following two lemmas, we give upper bounds on the size of the root node in two cases, when $\bar{K} \le B/3$ and $\bar{K} > B/3$, respectively. Let $\hat{c}_i$ be the average length of the keys in the ith set $\mathcal{C}_i$ and let k_i' be the minimum-length key in $\mathcal{C}_i$.

LEMMA 2.1. *Suppose that $\bar{K} \leq B/3$. Then the root node has size less than B and thus fits within a single block.*

PROOF. By (1) and because $\bar{K} \leq B/3$, we have

$$f = \max\left\{3, \left\lfloor \frac{B}{\bar{K}} \right\rfloor\right\} = \left\lfloor \frac{B}{\bar{K}} \right\rfloor \leq \frac{B}{\bar{K}}. \tag{2}$$

Because the total length of all N keys is the sum of the lengths of the keys in each set $\mathcal{C}_i$ $(0 \leq i \leq f-1)$, we have

$$\sum_{i=0}^{f-1} \frac{N\hat{c}_i}{f} = N\bar{K}.$$

Replacing the average key length $\hat{c}_i$ by the smallest key length k_i' for each i, we obtain

$$\sum_{i=0}^{f-1} \frac{Nk_i'}{f} \leq N\bar{K}. \tag{3}$$

Multiplying by f/N and applying (2), (3) simplifies to

$$\sum_{i=0}^{f-1} k_i' \leq f\bar{K} \leq B. \tag{4}$$

Because the root node stores $f-2$ representatives, it has size

$$\sum_{i=1}^{f-2} k_i' < B,$$

as promised. □

LEMMA 2.2. *Suppose that $\bar{K} > B/3$. Then the root contains a single key whose length is less than $3\bar{K}$ and therefore fits in at most $\lceil 3\bar{K}/B \rceil$ blocks.*

PROOF. By (1) and because $\bar{K} > B/3$, we have

$$f = \max\left\{3, \left\lfloor \frac{B}{\bar{K}} \right\rfloor\right\} = 3.$$

Thus, the root node has fanout 2 and contains only a single representative key from the middle set $\mathcal{C}_1$. Since the total length of all keys in the set $\mathcal{C}_1$ is strictly less than the total length of the N keys in $\mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C}_2$, we have $N\bar{K} > \hat{c}_1 N/3$, which simplifies to

$$\hat{c}_1 < 3\bar{K}.$$

Thus, the root node fits in at most $\lceil 3\bar{K}/B \rceil$ blocks. □

These $f-2$ representatives separate the N keys into $f-1$ sets, which we denote $\{\mathcal{S}_i\}_{1 \leq i \leq f-1}$. Each set becomes a child of the root node. We denote the average length of each child set $\mathcal{S}_i$ to be $\bar{K}_i$.

Recursive Step, Base Case, and Leaf Structure

We recursively apply the greedy algorithm to each child set S_i and thus generate the static atomic-key B-tree. Observe that different children may have different fanouts, depending on the values of $\bar{K}_i$. The base case is when a child set $\mathcal{S}_i$ fits entirely within

a B-sized chunk of memory or there is a single element remaining. Then, we assign all of $\mathcal{S}_i$ to a single leaf node.

We next coalesce the leaf nodes into a single contiguous chunk to ensure fast range queries and linear space. A traditional B-tree with unit-size keys is built from the leaves up, and by construction, all leaves have size $\Theta(B)$ (unless there is only one leaf). Here the construction is top-down, which can result in leaves of size $o(B)$. Thus, we store all leaves in order in a single chunk of space of size $O(N\bar{K})$. The space consumption is linear and range queries are fast because the keys are stored contiguously and in order.

Performance Results

The greedy layout achieves the following performance, as we prove in Section 3.

THEOREM 2.3. *A static atomic-key B-tree with N elements and average key size $\bar{K}$ has an expected search cost of $O(\lceil \bar{K}/B \rceil \log_{1+\lceil B/\bar{K} \rceil} N)$ block transfers when all leaves are searched with equal probability. A scan of L contiguous elements of average size $\bar{K}_L$ takes $O(1 + L\bar{K}_L/B)$ additional memory transfers after the first element in the range has been examined. The data structure consumes $O(N\bar{K})$ space, that is, linear space.*

A static atomic-key B-tree can be built with the following cost, as we prove in Section 4.

THEOREM 2.4. *A static atomic-key B-tree can be built with $O(1 + N\bar{K}/B)$ memory transfers—the scan bound—for presorted data.*

We conclude this section by giving limitations on the worst-case search cost in atomic-key B-trees.

THEOREM 2.5. *The expected search cost can not be achieved for worst-case searches. Specifically, there exists a set of N keys with average length $O(1)$ and maximum length ℓ, where $B \leq \ell \leq M$, for which the following holds. For every tree layout having an asymptotically optimal expected search cost, the worst-case search cost (e.g., tree height) is a factor of $\Omega(\frac{\ell}{B} \log(B+1))$ worse than the expected search cost.*

PROOF. Consider a key set in which the first $\sqrt{N}$ elements have length ℓ, where $B \leq \ell \leq M$, and the remaining $N - \sqrt{N}$ elements have length 1. An information-theoretic lower bound for searching in a binary tree shows that in any tree there is some leaf of length ℓ that requires $\Omega(\frac{\ell}{B} \log_2 N)$ memory transfers to reach. In contrast, if all pivots of length ℓ are deeper than any pivot of length 1, then the expected search cost is $O(\log_{1+B} N)$ memory transfers. □

3. SEARCH ANALYSIS

In this section we prove Theorem 2.3, thus establishing the expected search cost in an atomic-key B-tree.

We prove by induction on N that a static atomic-key B-tree with N elements and average key size $\bar{K}$ has an expected search cost of $O((1 + \bar{K}/B) \log_{2+B/\bar{K}} N)$ block transfers when all keys are searched with equal probability. Then Theorem 2.3 will follow directly. Because

$$\left\lceil \frac{\bar{K}}{B} \right\rceil \leq 1 + \frac{\bar{K}}{B} \leq 2 \left\lceil \frac{\bar{K}}{B} \right\rceil \tag{5}$$

(6)

and

$$1 + \left\lceil \frac{B}{\bar{K}} \right\rceil \leq 2 + \frac{B}{\bar{K}} \leq 2 \left(1 + \left\lceil \frac{B}{\bar{K}} \right\rceil \right), \tag{7}$$

the search cost is equivalent to $O(\lceil \bar{K}/B \rceil \log_{1+\lceil B/\bar{K} \rceil} N)$.

In an atomic-key B-tree, the root node R comprises $f-2$ keys (which is always at least one). Thus, the root node has $f-1$ children $\{T_i\}_{1 \leq i \leq f-1}$, each of which is a subtree on the set $\mathcal{S}_i$.

Assume for the sake of induction that for the subtrees T_i of size $|\mathcal{S}_i|$ (less than N), the search cost is

$$c \left(1 + \bar{K}_i/B\right) \log_{2+B/\bar{K}_i} |\mathcal{S}_i|,$$

for some constant $c > 0$. We show that the search cost also applies to the tree T of size N.

The expected search cost in the tree $T = (R, T_1, \ldots, T_{f-1})$ is the number of block transfers to fetch the root node R plus the expected search cost in the appropriate subtree T_i. We first calculate the number of block transfers to fetch the root node R. By Lemmas 2.1 and 2.2, when $\bar{K} \leq B/3$, the root node R has size less than B; when $\bar{K} > B/3$, R has size less than $3\bar{K}$. In summary, the number of block transfers to fetch the root node R is at most $1 + 3\bar{K}/B$.

To prove Theorem 2.3, we show that for the same constant c,

$$1 + \frac{3\bar{K}}{B} + \frac{c}{N} \sum_{i=1}^{f-1} |\mathcal{S}_i| \left(1 + \frac{\bar{K}_i}{B} \right) \log_{2+B/\bar{K}_i} |\mathcal{S}_i| \leq c \left(1 + \frac{\bar{K}}{B} \right) \log_{2+B/\bar{K}} N, \tag{8}$$

subject to the following constraints:

— The tree contains N keys, i.e.,

$$\sum_{i=1}^{f-1} |\mathcal{S}_i| = N, \tag{9}$$

— the total length of all keys is

$$\sum_{i=1}^{f-1} |\mathcal{S}_i| \bar{K}_i = N\bar{K}, \tag{10}$$

— and by construction, for all i,

$$0 < |\mathcal{S}_i| < 2N/f. \tag{11}$$

To simplify our calculations, we introduce some notation. Let $x_i = \bar{K}_i/B$, $x = \bar{K}/B$, and $t_i = |\mathcal{S}_i|/N$. Thus, (8)-(11) become respectively (12)-(15).

We need to show that for some constant $c > 0$,

$$1 + 3x + c \sum_{i=1}^{f-1} t_i (1 + x_i) \log_{2+1/x_i} (t_i N) \leq c(1+x) \log_{2+1/x} N\,, \tag{12}$$

subject to the constraints:

$$\sum_{i=1}^{f-1} t_i = 1\,, \tag{13}$$

$$\sum_{i=1}^{f-1} t_i x_i = x\,, \tag{14}$$

$$\forall i, 0 < t_i < 2/f\,. \tag{15}$$

Before proceeding further, we prove the following claim.

CLAIM 3.1. *Let $x_i > 0$ and $x > 0$, and let t_i be constrained as follows:*

$$\sum_{i=1}^{f-1} t_i = 1 \quad \text{and} \quad \sum_{i=1}^{f-1} t_i x_i \le x\,.$$

Then the following inequality holds:

$$\sum_{i=1}^{f-1} \frac{t_i(1+x_i)}{\ln(2+1/x_i)} \le \frac{1+x}{\ln(2+1/x)}\,. \tag{16}$$

PROOF. Let function $h(x)$ be defined as follows

$$h(x) = \frac{1+x}{\ln(2+1/x)} \quad (x > 0)\,.$$

Then (16) can be rewritten as

$$\sum_{i=1}^{f-1} t_i h(x_i) \le h(x).$$

The above inequality holds as long as the function h is concave and increasing on the positive line, because

$$\begin{aligned}\sum_{i=1}^{f-1} t_i h(x_i) &\le h\left(\sum_{i=1}^{f-1} t_i x_i\right) && \text{concavity of } h\\ &\le h(x) && h \text{ is increasing in } (0,\infty).\end{aligned}$$

We show that these two properties for h. The first derivative is

$$h'(x) = \frac{\ln(2+1/x) + (1+x)/(2x^2+x)}{\ln^2(2+1/x)}\,,$$

which is positive for $x > 0$. The second derivative is

$$h''(x) = \frac{2 + 2x - (3x+1)\ln(2+1/x)}{(2x^2+x)^2 \ln^3(2+1/x)}\,. \tag{17}$$

Because $x > 0$, $\ln(2+1/x) > 0$. Thus, the numerator of (17) is negative, i.e.,

$$\ln(2+1/x) > \frac{2+2x}{1+3x}\,.$$

Let $y = 1/x$. Because x is positive, the range of y is also $(0,\infty)$. Thus, by replacing $1/x$ by y in the above inequality, we obtain

$$\ln(2+y) > \frac{2y+2}{y+3} = 2 - \frac{4}{y+3} \quad (y > 0). \tag{18}$$

The derivative of the left side of (18) is

$$(\ln(2+y))' = \frac{1}{2+y} > 0\,,$$

and the derivative of the right side of (18) is

$$\left(2 - \frac{4}{y+3}\right)' = \frac{4}{y^2+6y+9} > 0.$$

Thus, both $\ln(2+y)$ and $2-4/(y+3)$ are monotonically increasing.

We also need to show that $\ln(2+y)$ increases faster than $2-4/(y+3)$ in $(0,\infty)$, that is,

$$(\ln(2+y))' \geq \left(2 - \frac{4}{y+3}\right)'.$$

This inequality holds because

$$\frac{1}{2+y} \geq \frac{4}{y^2+6y+9}\,,$$

which is equivalent to

$$y^2 + 2y + 1 \geq 0\,.$$

Furthermore, at the initial point, when $y = 0$,

$$\ln(2+y) = \ln 2 \approx 0.69\,,$$

and

$$2 - 4/(y+3) = 2 - 4/3 \approx 0.67\,.$$

Thus, the left side of (18) has an initial value greater than the right side of (18), mean that (18) holds.

In summary, $h''(x) < 0$, and therefore $h(x)$ is concave. Thus, the claim follows. □

Our objective is to establish (12)-(15). We first simplify the lefthand side of (12). By (15), we obtain

$$1 + 3x + c\sum_{i=1}^{f-1} t_i(1+x_i)\log_{2+1/x_i}(t_iN) \leq 1 + 3x + c\sum_{i=1}^{f-1} t_i(1+x_i)\log_{2+1/x_i}(2N/f).$$

Moving $\ln(2N/f)$ out of the summation in the above inequality, we obtain

$$1 + 3x + c\sum_{i=1}^{f-1} t_i(1+x_i)\log_{2+1/x_i}(t_iN) \leq 1 + 3x + c\ln(2N/f)\sum_{i=1}^{f-1} \frac{t_i(1+x_i)}{\ln(2+1/x_i)}. \tag{19}$$

From Claim 3.1 and (19), we obtain

$$1 + 3x + c\sum_{i=1}^{f-1} t_i(1+x_i)\log_{2+1/x_i}(t_iN) \leq 1 + 3x + c\ln(2N/f)\frac{1+x}{\ln(2+1/x)}. \tag{20}$$

Reorganizing the above inequality, we obtain

$$1 + 3x + c\sum_{i=1}^{f-1} t_i(1+x_i)\log_{2+1/x_i}(t_iN) \leq c(1+x)\log_{2+1/x} N + 1 + 3x - c(1+x)\log_{2+1/x}(f/2). \tag{21}$$

To prove the theorem, we need to find the constant c such that the right part in (21) is less than $c(1+x)\log_{2+1/x} N$, that is

$$1 + 3x - c(1+x)\log_{2+1/x}(f/2) \le 0.$$

Therefore, we derive that

$$c \ge \frac{(1+3x)\ln(2+1/x)}{(1+x)\ln(f/2)}.$$

Because $(1+3x)/(1+x) = 3 - 2/(1+x) < 3$, it is sufficient to find the constant c such that

$$c \ge 3\frac{\ln(2+1/x)}{\ln(f/2)}.$$

To find such constant c, we use the following claim.

CLAIM 3.2. *For $x = \bar{K}/B$ and $f = \max\{3, \lfloor B/\bar{K} \rfloor\}$, there exists a constant c independent of x and f, such that*

$$c \ge 3\frac{\ln(2+1/x)}{\ln(f/2)}. \tag{22}$$

Specifically, let $c = 3\ln 6/\ln(3/2)$.

PROOF.

There are two cases.

The first case is when $B/\bar{K} < 3$, meaning that $f = 3$ and $1/x < 3$. Then we can choose $c \ge 3\ln 5/\ln(3/2)$.

The second case is when $B/\bar{K} \ge 3$. We have

$$f = \left\lfloor \frac{B}{\bar{K}} \right\rfloor = \left\lfloor \frac{1}{x} \right\rfloor$$

and $1/x \ge 3$. Therefore we have

$$\frac{\ln(2+1/x)}{\ln(f/2)} \le \frac{\ln(3+\lfloor 1/x \rfloor)}{\ln(f/2)} \le \frac{\ln(2\lfloor 1/x \rfloor)}{\ln(f/2)} = \frac{\ln(2f)}{\ln(f/2)} \le \frac{\ln 6}{\ln(3/2)}.$$

The first inequality is by $1/x \le 1 + \lfloor 1/x \rfloor$; The second inequality follows from $\lfloor 1/x \rfloor \ge 3$; the third equation is from $f = \lfloor 1/x \rfloor$ and the last inequality follows by the fact that $\ln(2f)/\ln(f/2)$ is monotonically decreasing and $f \ge 3$. Thus, we can choose $c = 3\ln 6/\ln(3/2)$ to satisfy (22) for both cases. □

In conclusion, by (21) and for the constant c from Claim 3.2, we establish

$$1 + 3x + c\sum_{i=1}^{f-1} t_i(1+x_i)\log_{2+1/x_i}(t_i N) \le c(1+x)\log_{2+1/x} N.$$

4. BUILDING AN ATOMIC KEY B-TREE

This section presents an efficient construction algorithm for atomic-key B-trees, proving Theorem 2.4 and establishing that the greedy algorithm from Section 2 can be realized with few I/Os.

The algorithm takes as input a set of N sorted keys of variable size and returns an atomic-key B-tree. The construction cost matches the scan bound of $O(1 + N\bar{K}/B)$ transfers, which is optimal. The idea for building the tree is to proceed level by level,

starting at the root and continuing down the tree. First select the pivot elements in the root, and then recursively build each subtree; see Section 2.

The algorithmic challenge addressed in this section is how to select the pivots quickly.

By way of comparison, a naïve solution scans through the elements to select the pivot keys. The running time is bounded by the height of the tree times the scan bound. Observe that the height of the tree could be much larger than the expected search cost, as Theorem 2.5 indicates.

This section gives a faster way to select pivot elements. The trick is to preprocess the keys to answer two different kinds of queries, "average-length queries" and "minimum-length queries," as defined below.

Average-Length Queries

The first preprocessing step is basic: preprocess the N keys to answer queries about the *average* length of the elements in a given range. Thus, a query is a pair of indexes (i, j), and the response is the average length of the elements i through j. Answering a query takes $O(1)$ memory transfers. Preprocessing takes $O(1+N\bar{K}/B)$ memory transfers and an additional $O(N)$ space. Preprocess by storing, for each key i, the total length of the first i keys. This step requires a simple linear scan of the data.

Minimum-Length Queries

The second preprocessing step is more involved: preprocess the N keys to answer queries about the *minimum* length of the elements in a range. A query is a pair of indexes (i, j), and the response is the index k of the shortest element ranked between i and j. As before, answering a query takes $O(1)$ memory transfers, and preprocessing takes $O(1 + N\bar{K}/B)$ memory transfers and an additional $O(N)$ space.

This algorithmic problem is well known as the ***Range Minimum Query (RMQ)*** problem and is closely related to the ***Least Common Ancestor (LCA)*** problem. In the RAM model there are linear time reductions between the two problems [Gabow et al. 1984], and the inputs can be preprocessed in linear time to answer constant-time queries [Harel and Tarjan 1984; Schieber and Vishkin 1988; Berkman and Vishkin 1993; Bender and Farach-Colton 2000].

In contrast, we are interested in external-memory data structures. In the DAM, there is no known linear-time (that is, matching the scan-bound) reduction between the RMQ and the LCA. There exist older LCA algorithms designed for external memory [Chiang et al. 1995]. However these algorithms have different tradeoffs between preprocessing and queries, and they are designed to answer multiple LCA queries in bulk.

It is relatively straightforward to adapt [Bender and Farach-Colton 2000] to solve RMQ queries for the case where the largest key has length $O(M/\log\log N)$ (or even $O(M/\log\log\log N)$). The idea is to build a data structure for RMQ when $j - i = \Omega(\log\log N)$ or even $j - i = \Omega(\log\log\log N)$. This restriction is good enough to solve all interesting cases for element sizes.

However, [Demaine et al. 2009] gives a cache-oblivious solution to RMQ problem with no restriction on element sizes. Their data structure answers queries in $O(1)$ transfers and preprocesses an array storing the lengths of the N elements in $O(1+N/B)$ memory transfers, which is the scan bound and therefore optimal. Building such array entails scanning all N elements, which takes $O(1 + N\bar{K}/B)$ memory transfers. (Cache-oblivious means that the solution is not parameterized by B or M. The algorithm is memory-hierarchy universal, working simultaneously for all values of B or M.) By using this data structure, we achieve the desired performance for minimum-length queries.

Tree Construction

We now show how to build an atomic-key B-tree. First perform the precomputation for average-length and minimum-length queries. Then build the atomic-key B-tree recursively, starting with the root.

The base case is when the atomic-key B-tree begins to fit in main memory (when its size becomes smaller than M) or contains a single element. For the base case, the cost to build the tree is one plus the size of the tree divided by B, that is, the cost of a linear scan.

Next we show how to build the root node when the atomic-key B-tree is larger than M and contains more than one element. First ask an average-length query to find the average key size $\bar{K}$ of all N keys; this uses $O(1)$ transfers. Then calculate $f = \max\{3, \lceil B/\bar{K} \rceil\}$.

Next pick the set of $f-2$ representative keys from the sets $\mathcal{C}_1 \ldots \mathcal{C}_{f-2}$. To do so, first calculate the boundaries between each set $\mathcal{C}_i$. The minimal-length element in each set, which is found by a minimum-length query, is the representative element.

Thus, the cost to construct the root node is as follows.

LEMMA 4.1. *The root node can be built using $O(f + \bar{K}/B)$ transfers.*

PROOF. There are two cases. If $\bar{K} \leq B/3$, then the cost to build the root is dominated by the cost to identify the representatives. We can identify each of the $f-2$ representative elements by a minimum-length query at a cost of $O(1)$ transfers per representative for a total of $O(f)$ memory transfers.

If $\bar{K} > B/3$, then the root only has one element in it. The cost to build the root is dominated by the cost to write the element, which is $\lceil 3\bar{K}/B \rceil$. □

We now finish the proof of Theorem 2.4. In the base case of the construction algorithm, it is efficient to build an atomic-key B-tree, costing only a linear scan. In the recursive step, it may be more expensive to build (internal) nodes, as Lemma 4.1 indicates, since there is an additive cost of $O(f)$. However, we can charge the cost of building these internal node to the cost of touching each of its children. Thus, a static atomic-key B-tree can be built in the scan bound, which is $O(1 + N\bar{K}/B)$ memory transfers.

We conclude the section by giving construction bounds when the data is not sorted. The following (straightforward) bound applies to the case where no key is too large compared to main memory.

COROLLARY 4.2. *Suppose that the longest key has length $M^{1-\varepsilon}B^{\varepsilon}$, for constant ε ($0 < \varepsilon < 1$). Then an $(M/B)^{\varepsilon}$-way merge sort can presort the keys in*

$$O\left(\frac{N\bar{K}}{\varepsilon B}\log_{M/B}\frac{N}{B}\right)$$

transfers, which is asymptotically optimal, since ε is a constant.

5. DYNAMIC STRUCTURE

In this section we give a dynamic atomic-key B-tree on N elements.

Our results apply to two models of how insertions arrive, one strictly more general than the other. In the first model elements of arbitrary length are inserted at *random* locations in the atomic-key B-tree. In the second model elements of arbitrary length are inserted at *arbitrary* locations in the atomic-key B-tree. We give a recursive structure for the dynamic atomic-key B-tree, which is a modification of the structure presented in Sections 2-4.

We prove bounds showing that representative elements can retain their status despite a bounded number of subsequent inserts or deletes. We then give a dynamic atomic-key B-tree that supports efficient inserts and deletes when the average key size $\bar{K} = O(B)$. Finally, we explain how to use indirection for large keys to give a dynamic atomic-key B-tree that supports efficient inserts and deletes for both small and large keys.

We establish the following theorem.

THEOREM 5.1. *There exists a dynamic atomic-key B-tree with N elements and average key size $\bar{K}$ having the following performance bounds:*

(1) *Search cost. The expected search cost is $O(\lceil \bar{K}/B \rceil \log_{1+\lceil B/\bar{K} \rceil} N)$ transfers when all keys are searched with equal probability.*
(2) *Scan/range-query cost. Any subset of L contiguous elements with average size $\bar{K}_L$ can be scanned in $O(1 + L\bar{K}_L/B)$ transfers.*
(3) *Space consumption. The tree has linear size, i.e., size $O(N\bar{K})$.*
(4) *Construction cost. The tree can be built using a linear number of block transfers, i.e., $O(1 + N\bar{K}/B)$ transfers, if the keys are presorted. A subtree of L elements with average size $\bar{K}_L$ can be rebuilt in $O(1 + L\bar{K}_L/B)$ transfers.*
(5) *Random-insert cost. The expected amortized cost to insert or delete an element κ, of length at most a constant fraction of memory, at a random location in the atomic-key B-tree is*

$$O(\lceil \bar{K}/B \rceil \log_{1+\lceil B/\bar{K} \rceil} N + |\kappa|/B)$$

transfers, where $\bar{K}$ is the average key size after the modification.
(6) *Arbitrary-insert cost. The amortized cost to insert or delete an element κ, which fits in a constant fraction of memory, at an arbitrary location is asymptotically the same as the search cost, i.e., the amortized modification cost is dominated by the search cost.*

Recursive Structure

As with the static atomic-key B-tree, the algorithm builds the dynamic atomic-key B-tree recursively starting at the root. The algorithm operates as follows. Define a root parameter

$$f = \max\left\{2, \left\lfloor \frac{B}{\bar{K}} \right\rfloor\right\}, \tag{23}$$

and divide the N elements into $f + 1$ sets $\{\mathcal{C}_i\}_{i\in[0,f]}$. The first and last sets, $\mathcal{C}_0$ and $\mathcal{C}_f$, each contains $N/(2f)$ elements, and the remaining sets, $\mathcal{C}_1, \ldots, \mathcal{C}_{f-1}$, each contains N/f elements. Pick a representative key $\{r_i\}_{i\in[1,f-1]}$ from each of the middle sets $\mathcal{C}_1 \ldots \mathcal{C}_{f-1}$, and store the representatives in the root node. In Section 2 we selected the minimal-length key in each set as a representative. Here we relax this requirement. For each set $\mathcal{C}_i$, we have the freedom to choose any key whose length is at most a constant fraction larger than the average key length $\hat{c}_i$ of the elements in $\mathcal{C}_i$.

We first show that if the average key length $\bar{K}$ is at most $B/2$, then the root fits within a constant number of disk blocks.

LEMMA 5.2. *Suppose that $\bar{K} \le B/2$, and let $\beta > 0$ be a constant. If we choose a representative key r_i from $\mathcal{C}_i$ such that $r_i \le \beta\hat{c}_i$, then the root node has size at most βB and thus fits within $\lceil \beta \rceil$ blocks.*

PROOF. If $\bar{K} \leq B/2$, then by definition,

$$f = \max\left\{2, \left\lfloor \frac{B}{\bar{K}} \right\rfloor\right\} = \left\lfloor \frac{B}{\bar{K}} \right\rfloor. \tag{24}$$

Because the total length of all N keys is the sum of the lengths of the keys in each set $\mathcal{C}_i$, $0 \leq i \leq f$,

$$N\bar{K} = \sum_{i=1}^{f-1} \frac{N}{f}\hat{c}_i + \frac{N}{2f}\hat{c}_0 + \frac{N}{2f}\hat{c}_f \geq \sum_{i=1}^{f-1} \frac{N}{f}\hat{c}_i.$$

We choose a representative r_i from the set $\mathcal{C}_i$ such that $r_i \leq \beta\hat{c}_i$. Thus, the above inequality becomes

$$\sum_{i=1}^{f-1} r_i \leq \beta f \bar{K}.$$

Since $f = \lfloor B/\bar{K} \rfloor$, meaning that $f\bar{K} \leq B$, we obtain

$$\sum_{i=1}^{f-1} r_i \leq \beta B.$$

Since the root contains the representative keys, the lemma follows immediately. □

We next show that if the average element length $\bar{K} \geq B/2$, then the root has size $O(\bar{K})$.

LEMMA 5.3. *Suppose that $\bar{K} > B/2$, and let $\beta > 0$ be a constant. Then the root maintains a single element whose length is at most $2\beta\bar{K}$, and therefore fits in $\lceil 2\beta\bar{K}/B \rceil$ blocks.*

PROOF. If $\bar{K} > B/2$, then by definition

$$f = \max\left\{2, \left\lfloor \frac{B}{\bar{K}} \right\rfloor\right\} = 2. \tag{25}$$

Thus, the root node has fan-out 2 and contains only a single representative from the set $\mathcal{C}_1$. Since $|\mathcal{C}_1| = N/2$,

$$N\bar{K} > \hat{c}_1 N/2.$$

The representative r_1 satisfies the inequality that $r_1 \leq \beta\hat{c}_1$. Thus, $N\bar{K} \geq r_1 N/(2\beta)$, which simplifies to $r_1 \leq 2\beta\bar{K}$. Therefore the root node fits in $\lceil 2\beta\bar{K}/B \rceil$ blocks. □

Because the representative key need not be the minimal-length key in each set $\mathcal{C}_i$, we have the flexibility to choose r_i such that it is closer to the middle of $\mathcal{C}_i$. This requirement shows that we can have somewhat more balanced trees without hurting our search bounds. The following lemma quantifies how close the representative key r_i from $\mathcal{C}_i$ can be to the middle of that set.

LEMMA 5.4. *Let $\beta > 1$ be a constant. For set $\mathcal{C}_i$ $(1 \leq i \leq f-1)$, there are more than $(1 - 1/\beta)N/f$ keys that are shorter than $\beta\hat{c}_i$.*

PROOF. We divide the set $\mathcal{C}_i$ $(1 \leq i \leq f-1)$ into two disjoint subsets, $\mathcal{C}_i^s$ and $\mathcal{C}_i^\ell$, meaning that

$$|\mathcal{C}_i| = |\mathcal{C}_i^s| + |\mathcal{C}_i^\ell| = N/f\,. \tag{26}$$

The set $\mathcal{C}_i^s$ contains all elements shorter than $\beta\hat{c}_i$, and the set $\mathcal{C}_i^\ell$ contains all elements of length at least $\beta\hat{c}_i$. Since each element has length at least 1,

$$\hat{c}_i|\mathcal{C}_i| \geq \beta\hat{c}_i|\mathcal{C}_i^\ell| + |\mathcal{C}_i^s| > \beta\hat{c}_i|\mathcal{C}_i^\ell|. \tag{27}$$

By (26) and (27), we obtain

$$|\mathcal{C}_i^s| > \left(1 - \frac{1}{\beta}\right)\frac{N}{f}.$$

□

Now we explain how to choose the representative key. Our algorithm is parameterized by some constant $\beta > 1$. Unlike in Section 2, we do not choose the minimal-length key from the whole set $\mathcal{C}_i$. Instead, we choose a minimal-length key whose rank is sufficiently close to the middle of the set. Specifically, suppose that the set $\mathcal{C}_i$ $(1 \leq i \leq f-1)$ starts with the element of rank $\left(i - \frac{1}{2}\right)\frac{N}{f}$ and ends just before the element of rank $\left(i + \frac{1}{2}\right)\frac{N}{f}$. Then we choose the minimum length element whose rank lies between $\left(i - \frac{1}{2\beta}\right)\frac{N}{f}$ and $\left(i + \frac{1}{2\beta}\right)\frac{N}{f}$.

We give a brief definition. We say that a representative element r_i in $\mathcal{C}_i$ is $(\boldsymbol{\beta, \gamma})$-***good*** if r_i has length less than $\beta\hat{c}_i$, and there are at least a $(1/2-\gamma)$-fraction of elements in $\mathcal{C}_i$ both before and after r_i. Thus, we have the following corollary to Lemma 5.4.

COROLLARY 5.5. *Each representative element r_i $(i \leq 1 \leq f-1)$ is $(\beta, 1/(2\beta))$-good when it is chosen.*

These $f-1$ representatives separate the N elements into f sets $\{\mathcal{S}_i\}_{i\in[1,f]}$, each of which is stored in a different subtree of the root. We thus bound $|\mathcal{S}_i|$ as follows.

LEMMA 5.6. *If all representatives are $(\beta, 1/(2\beta))$-good, then for all child sets $\mathcal{S}_i$ $(1 \leq i \leq f)$ the greedy layout guarantees that $\left(1 - \frac{1}{\beta}\right)\frac{N}{f} \leq |\mathcal{S}_i| \leq \left(1 + \frac{1}{\beta}\right)\frac{N}{f}$.*

PROOF. Recall that the first set $\mathcal{S}_1$ and the last $\mathcal{S}_f$ are constructed differently from the remaining sets $\{\mathcal{S}_i\}_{2\leq i\leq f-1}$. We first bound the size of $\mathcal{S}_1$. (Bounding $\mathcal{S}_f$ has the same analysis.) Set $\mathcal{S}_1$ contains all elements before r_1 and is composed of two parts: (1) all the elements in $\mathcal{C}_0$ and (2) all the elements in $\mathcal{C}_1$ before r_1. By Corollary 5.5, which bounds the number of elements before r_1,

$$\left(1 - \frac{1}{2\beta}\right)\frac{N}{f} \leq |\mathcal{S}_1| \leq \left(1 + \frac{1}{2\beta}\right)\frac{N}{f}. \tag{28}$$

Now we bound the size of each set $\mathcal{S}_i$ $(2 \leq i \leq f-1)$. Recall that $\mathcal{S}_i$ comprises the set of keys between r_{i-1} and r_i, and contains two parts: (1) the keys in $\mathcal{C}_{i-1}$ after r_{i-1} and (2) the keys in $\mathcal{C}_i$ before r_i. By Corollary 5.5,

$$\left(1 - \frac{1}{\beta}\right)\frac{N}{f} \leq |\mathcal{S}_i| \leq \left(1 + \frac{1}{\beta}\right)\frac{N}{f}. \tag{29}$$

The lemma follows from (28) and (29). □

In summary, the atomic-key B-tree has the following structure:

LEMMA 5.7. *The dynamic atomic-key B-tree guarantees the following properties:*

(*1*) *The root node has size $\Theta(B + \bar{K})$.*
(*2*) *Each child set $\mathcal{S}_i$ of the root contains $\Theta(N/f)$ elements.*

As in Section 2, the base case is when the total length of all elements in a child set $\mathcal{S}_i$ is at most B or when there is a single element remaining. Using an almost identical analysis to that in Section 3, we can establish Property 1 of Theorem 5.1.

We now quantify the goodness of the representative elements after some elements have been inserted or deleted.

LEMMA 5.8. *Let $\beta > 5/3$. Suppose that the representative elements are $(\beta, 1/(2\beta))$-good when they are chosen. After $N/(3\beta f)$ elements have been inserted or deleted, these representatives are $((1/2 + 3\beta/2), 5/(6\beta))$-good.*

PROOF. If at most $N/(3\beta f)$ elements have been inserted or deleted, then the fraction of elements either before or after the representative r_i can increase by at most an additive $1/(3\beta)$ amount, that is, from $1/(2\beta)$ to $5/(6\beta)$.

When the representative was chosen, there were at most $(1 - 1/\beta)N/f$ elements shorter than r_i (those outside the range from which r_i was chosen) and at least $N/(\beta f)$ elements at least as long as r_i (those in the range from which r_i was chosen).

Since then, at most $N/(3\beta f)$ elements shorter than r_i were inserted and at most $N/(3\beta f)$ elements longer than r_i were deleted. Thus, at most $(1-2/(3\beta))N/f$ elements are shorter than r_i and at least $2N/(3\beta f)$ elements are at least as long.

The total length of all elements in $\mathcal{C}_i$ is greater than $2N/(3\beta f)r_i$. Let $\hat{c}_i$ now represent the average size of $\mathcal{C}_i$ after these inserts and deletes. The total length of all elements in $\mathcal{C}_i$ is less than $\hat{c}_i(N/f + N/(3\beta f))$. Thus, r_i has length at most $(1/2 + 3\beta/2)\hat{c}_i$, as promised. □

Leaf Structure and Construction

Because the tree is built recursively from top down, leaf nodes need not have size $\Omega(B)$. Consequently, we coalesce leaf nodes to guarantee optimal range query performance, linear space, and the ability to rebuild subtrees efficiently.

Leaf node sizes can vary enormously. If the leaf contains a (single) element whose size is greater than B, then the leaf is the size of the element. If the leaf only contains keys that are $O(B)$, then it has size $O(B)$ but can be as short as 1, depending on how the recursion bottoms out. Leaf nodes can shrink, grow, or split as the tree is updated.

We coalesce neighboring leaf nodes into chunks. We maintain the invariant that a chunk can have size $o(B)$ only if one of its neighboring chunks has size $\Omega(B)$. Thus, we divide chunks into two categories, large chunks and small chunks. A large chunk contains a single leaf of size greater than $B/2$. In a small chunk, all leaves have size at most $B/2$. As insertions occur, the size of the chunks can grow or shrink or some subset of contiguous chunks can be rebuilt from scratch. We can split and merge chunks similar to standard splitting and merging algorithms, but with a minor twist.

Only small chunks can merge and split with each other. Thus, if a small chunk gets too full (of size $3B/2$), then it splits into two small chunks, each of size at least $B/2$. If a small chunk gets too empty, then it merges with a neighboring small chunk, if one exists. This merge may subsequently induce a split. If it cannot merge, then all neighbors are large chunks.

We build the dynamic atomic-key B-tree efficiently by finding representative elements as described in Section 4.

The combination of chunks and splits ensures that Properties 2 and 3 of Theorem 5.1 are satisfied. Because we can perform efficient range queries, we can efficiently rebuild subtrees of the atomic-key B-tree. Thus, Property 4 of Theorem 5.1 is satisfied.

Insertions and Deletions (Special Case)

In this subsection we give insertion and deletion algorithms for the important special case when all keys have size $O(B)$.

In our algorithms, subtrees of the dynamic atomic-key B-tree are periodically rebuilt as elements are inserted or deleted. The atomic-key B-tree is parameterized by the constant β (Lemma 5.8), which determines when to rebuild. We say that a ***node is rebalanced*** if the entire subtree rooted at that node is rebuilt (but the parent is not). We say that a node is ***involved in a rebalance***, if it belongs to the subtree that is being rebuilt.

If a node v is the root of a subtree of size N', then it is rebalanced when there have been $N'/(3\beta f)$ inserts or deletes into the subtree rooted at v since v's previous rebalance. Each node in the atomic-key B-tree stores counters and other auxiliary information to determine when to rebalance.

To insert a key κ into the atomic-key B-tree rooted at a given node v, first examine v to decide whether the insert triggers a rebalance of v. If so, incorporate κ into the new subtree being rebuilt. Otherwise, search for the subtree of v (storing child set $\mathcal{S}_i$) where κ should reside, and recursively insert into it.

To delete a key κ from an atomic-key B-tree rooted at a node v, proceed similarly. First examine v to decide whether the delete triggers a rebalance of v. If so, rebuild the atomic-key B-tree, removing κ. Otherwise, check whether κ is stored in v as a representative element. If so, mark κ as deleted. Do not remove it because it can still be used for guiding searches. Then search for the subtree of v (storing a child set $\mathcal{S}_i$) where κ belongs and recursively delete κ from $\mathcal{S}_i$.

The base case is when the entire set $\mathcal{S}_i$ fits in a leaf, in which case, we rearrange the elements in the leaf inserting or deleting κ, as appropriate. As described in the previous subsection, rearranging elements in the leaf may cause the size of the leaf to change, triggering restructuring of one or several chunks. Similarly, rebuilding subtrees of the atomic-key B-tree requires restructuring of the chunks.

LEMMA 5.9. *Consider a dynamic atomic-key B-tree with N elements each of size $O(B)$ and average element size $\bar{K}$. The amortized cost to rebalance the root of the tree per insert/delete is at most the cost to read the root of the tree, which is $O(1)$.*

PROOF. Consider two consecutive rebalances of a node and the interval in between these rebalances. Let R_1 and R_2 denote the cost to perform the first and second rebalance, respectively. Let $\bar{K}_1$ and N_1 denote the average length of an element and number of elements in the node's subtree during the first rebalance. Let f_1 be defined according to (24) for the first rebalance.

We give an amortized analysis for paying for the cost R_2 of the second rebalance by charging to the inserts and deletes in between. Observe that $R_1 = O(1 + N_1\bar{K}_1/B)$. Let I denote the sum of the lengths of all the keys inserted between the two rebalances.

The cost of the second rebalance is at most the cost of the first rebalance plus the cost to scan all the keys inserted in the second rebalance. That is:

$$R_2 = O(R_1 + I/B). \tag{30}$$

Since there are $N_1/(3\beta f_1)$ inserts or deletes between rebalances, for constant $\beta > 5/3$, we have the following (loose) bounds on I:

$$\frac{N_1}{3\beta f_1} \leq I \leq \frac{N_1 B}{3\beta f_1}. \tag{31}$$

Therefore, by (30) and (31),

$$R_2 = R_1 + O(I/B) = O(1 + N_1\bar{K}_1/B + N_1/f_1). \tag{32}$$

We charge the rebalance cost R_2 to the $\Theta(N_1/f_1)$ inserts and deletes that take place between the rebalances. Thus, by (24) and (32) the amortized rebalance cost per inser-

tion is

$$O(R_2 f_1/N_1) = O(f\bar{K}_1/B + 1) = O(1), \tag{33}$$

establishing the claim. □

Observe that the proof of Lemma 5.9 needed only very loose bounds on I; see (31). Specifically, in the rebalance, each newly inserted/deleted element can contribute at most one I/O per element, which follows from our assumption that elements have length $O(B)$.

We now give the amortized cost to insert into the entire tree.

LEMMA 5.10. *Consider a dynamic atomic-key B-tree with N elements, each of size $O(B)$. The amortized cost to insert or delete an element κ into the tree is at most the cost to perform a search for κ.*

PROOF. In order to insert κ into the tree, it first needs to be read into memory, which costs $O(1 + |\kappa|/B) = O(1)$ memory transfers, since $\kappa = O(B)$.

Then κ is inserted into the atomic-key B-tree. By Lemma 5.8, the pivot keys are always good representatives. The insert makes its way along a prefix of a root-to-leaf search path until it triggers a rebalance. The insert/delete has effectively modified the balance of all of the nodes along this path. By Lemma 5.9, the amortized cost to rebalance these nodes is $O(1)$ per node. Thus, the amortized cost to perform this insert or delete is at most the cost to read all the nodes along the root-to-leaf path. □

Lemma 5.10 establishes Properties 5 and 6 of Theorem 5.1 for the case when all keys have length $O(B)$.

This atomic-key B-tree may not achieve good performance bounds when $\bar{K} = \Omega(B)$. The proof of Lemma 5.10 gives some indication why. One difficulty is that the average element size $\bar{K}_1$ during the first rebalance can be very different from the average element size $\bar{K}$ later when an insert/delete takes place. This difference affects the accounting argument and becomes important when the amortized rebalance cost (see (33)) is $\Omega(1)$.

Storing Large Elements Using Indirection

The solution is to use indirection for large keys. Small representative keys are stored in nodes as described earlier. A representative key that is sufficiently large (e.g., at least $4B$) is not stored directly in a node. Rather, the node maintains a pointer to the key along with that key's size, and the key is stored elsewhere. This strategy is reminiscent of how binary large objects (blobs) are kept in a B-tree.

The rest of the space in the tree node can be left empty. For practical reasons, one can coalesce these pointers and sizes into fewer blocks. However, this compaction is not necessary to achieve good asymptotic bounds.

The advantage of indirection is that it obviates the need to recopy large keys when rebalancing subtrees. A rebalance now requires manipulating pointers and examining sizes, not looking at the keys themselves. Without indirection, the cost to rebalance a tree with N nodes and average key size $\bar{K}$ is $O(1 + N\bar{K}/B)$, which can be large if $\bar{K}$ is large. With indirection, the rebalance cost may be smaller smaller, because even when keys are larger than B, we do not have to pay more than $O(1)$ I/Os for any key involved in a rebalance. This means that with indirection, the rebalance cost is $O(1 + N \min\{\bar{K}, B\}/B)$.

Insertions and Deletions

With indirection we obtain the following generalizations of Lemmas 5.9 and 5.10.

LEMMA 5.11. *Consider a dynamic atomic-key B-tree with N elements. The amortized cost per insertion or deletion to rebalance the root of the tree is at most $O(1)$.*

PROOF. Similar to the proof of Lemma 5.9, consider two consecutive rebalances of a node and the interval in between these rebalances. Let R_1 and R_2 denote the cost to perform the first and second rebalance, respectively. Let $\bar{K}_1$ and N_1 denote the average length of an element and number of elements in the node's subtree during the first rebalance. Let f_1 be defined according to (24) for the first rebalance.

We give an amortized analysis for paying for the cost R_2 of the second rebalance by charging to the inserts and deletes in between. Observe that now, since keys can have length that is $O(B)$ or $\Omega(B)$,

$$R_1 = O(1 + N_1 \min\{\bar{K}_1, B\}/B). \tag{34}$$

Define the ***truncated length of a key*** to be the minimum of the key length and B. Let $\hat{I}$ and I denote the sum of the truncated lengths and the sum of the full lengths of all the keys inserted between the two rebalances.

The cost of the second rebalance is at most the cost of the first rebalance plus the cost to scan all the truncated lengths in the second rebalance. Because of indirection, no key contributes more than one I/O when it participates in a rebalance, no matter how long it happens to be. Thus, we obtain:

$$R_2 = O(R_1 + \hat{I}/B). \tag{35}$$

Since there are $N_1/(3\beta f_1)$ inserts or deletes between rebalances, for constant $\beta > 5/3$, we have the following (loose) bounds on $\hat{I}$:

$$\frac{N_1}{3\beta f_1} \le \hat{I} \le \frac{N_1 B}{3\beta f_1}. \tag{36}$$

Therefore, by (34), (35), and (36),

$$R_2 = R_1 + O(\hat{I}/B) = O(1 + N_1 \min\{\bar{K}_1, B\}/B + N_1/f_1). \tag{37}$$

We charge the rebalance cost R_2 to the $\Theta(N_1/f_1)$ inserts and deletes that take place between the rebalances, meaning that

$$\text{the amortized cost per insertion} = f_1 R_2/N_1. \tag{38}$$

There are two cases for (38), depending on whether, $\bar{K}_1 \le B/2$, in which case (24) holds, or whether $\bar{K}_1 > B/2$, in which case (25) holds.

In the first case, where $f_1 = \lfloor B/\bar{K}_1 \rfloor$, Eq. (38) simplifies to

$$O(R_2 f_1/N_1) = O(f_1 \bar{K}_1/B + 1) = O(1). \tag{39}$$

In the second case, where $f_1 = 2$, Eq. (38) also simplifies to

$$O(R_2 f_1/N_1) = O(1), \tag{40}$$

establishing the lemma. □

LEMMA 5.12. *Consider a dynamic atomic-key B-tree with N elements. The amortized cost to insert or delete an element κ into the tree is $O(1 + |\kappa|/B)$, the cost to read the element, plus the cost to perform a search for κ in the tree.*

PROOF. In order to insert κ into the tree, it first needs to be read into memory, which costs $O(1 + |\kappa|/B)$ memory transfers. Key κ will reside in memory during the entire search procedure.

Then κ is inserted into the atomic-key B-tree as described in Lemma 5.10. By Lemma 5.12, the amortized cost to rebalance these nodes is $O(1)$ per node. Thus, the amortized cost to perform this insert or delete is at most the cost to read all the nodes along the root-to-leaf path. □

Lemma 5.12 establishes Properties 5 and 6 of Theorem 5.1.

6. AN OPTIMAL DYNAMIC PROGRAM

This section presents a dynamic program that produces an optimal static search tree for N atomic keys, each of which may have a different size, and each of which has a fixed probability of being queried.

The optimal structure minimizes the expected number of blocks transfer in a root-to-leaf path.

This dynamic program assumes that every key can fit into a block, leaving enough space if needed for pointers to other blocks. The program produces only those search trees in which each node comprises a single block.

The strategy used is that trees for larger sets of keys are constructed by joining trees for smaller sets. An optimal tree for the set $\kappa_i, \ldots, \kappa_j$ occupying space less than S in the root is constructed by joining an optimal tree for the set $\kappa_{r+1}, \ldots, \kappa_j$ with space less than $S - K_r$ in the root and an optimal tree for the set $\kappa_i, \ldots, \kappa_{r-1}$ in an optimal way.

Figure 2 illustrates the dynamic program.

Let

$$K_{i,j} = \sum_{i \le x \le j} K_x$$

and

$$P_{i,j} = \sum_{i \le x \le j} p_x.$$

Let $t(i, j, k)$ be the optimal tree and $c(i, j, k)$ be its cost for keys $\kappa_i, \ldots, \kappa_j$ subject to the constraint that the sum of the lengths of the keys in the root node is less than k.

We further assume that p is the size of a pointer from one block to another. In the rest of the paper we did not need to worry about the storage for the pointer p because such considerations do not change the asymptotics.

$$c(i, j, k) = \begin{cases} P_{i,j} & \text{if } K_{i,j} \le k \\ \min_{i \le r \le j} F(i, r, j, k) & \text{otherwise.} \end{cases}$$

where $K_r + p \le k$, then

$$\begin{aligned} F(i, r, j, k) = \quad & c(i, r-1, B) + P_{i,r-1} && \text{(a subtree)} \\ & +p_r && (\kappa_r \text{ here}) \\ & +c(r+1, j, k - K_r - p). && \text{(rest of block)} \end{aligned}$$

The condition that $K_r + p \le k$ corresponds to being able to fit a pointer to the subtree, plus the key κ_r into the block of size k. If $K_r + p > k$ then

$$F(i, r, j, k) = \infty.$$

The cost of this dynamic program is $O(Bn^3)$ operations.

To handle keys that are larger than a single block is in principle straightforward, but the dynamic program seems substantially more complex without yielding any real insight to the problem.

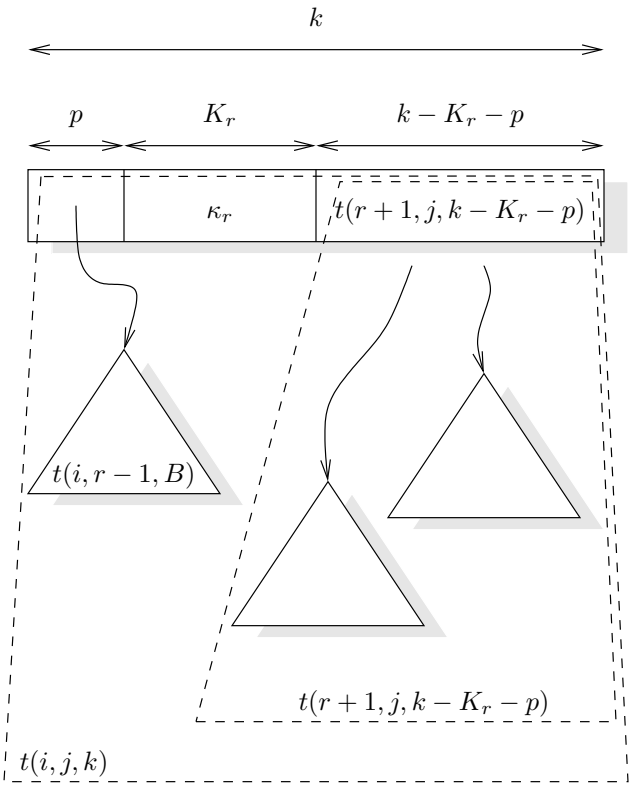

Fig. 2. The dynamic program produces a tree $t(i, j, k)$ by finding an optimal way of allocating the root block of size k among a pointer of size p to a left subtree, a key κ_r, and a right subtree. The left subtree has root block of size B and holds keys $\kappa_i \ldots \kappa_{r-1}$, and incurs the cost of a following a pointer. The right subtree holds keys $\kappa_{r+1} \ldots \kappa_j$. The right subtree's root block is "inline" with the block and uses space $k - K_r - p$ to hold its root.

7. CACHE-OBLIVIOUS STATIC ATOMIC-KEY B-TREE

In this section we give a method for building a static cache-oblivious atomic-key B-tree. We prove that for any given fixed block size B, the performance of the cache-oblivious dictionary is asymptotically the same as the performance of an atomic-key B-tree.

In order to build our cache oblivious data structure, we use the "split-and-refine" technique [Alstrup et al. 2002], which converts a family of layouts for known block sizes into a universal layout that works for all block sizes. This conversion increases the expected block cost by at most a constant factor, as we explain below.

Overview of Cache-Oblivious Approach

Here we give an overview of our approach for building a cache-oblivious static atomic-key B-tree by using "split-and-refine."

Split-and-refine was originally proposed as a technique for solving the following tree-layout problem [Alstrup et al. 2002]. We are given a (not necessarily balanced) fixed-topology tree; how can we lay out the tree in external memory to minimize the expected block cost for searches?

Alstrup et al [Alstrup et al. 2002] first show how to solve this problem for a given block size B; in this case, laying out the tree means optimizing how to pack nodes into blocks of size B. Then the paper shows a universal (and hence cache-oblivious)

solution to the problem, which is simultaneously optimal for all possible values of B. This layout is based upon selecting several values of B, each leading to a different layout, and then combining the layouts appropriately. (Alstrup et al. also assume a uniform distribution on the keys for searches.)

This careful combination is what we call split-and-refine. What makes this technique particularly interesting is that the packing of tree nodes into blocks for one specific value of B may have little to do with the packing for other values of B. Nonetheless, this approach still works.

We want to apply split-and-refine here for building cache-oblivious atomic-key B-trees, but there are several serious obstacles.

The most daunting one is that when we have an atomic-key B-tree, we do not actually have a fixed-topology tree—in fact, quite the opposite. That is, in the tree-layout problem of Alstrup et al. [Alstrup et al. 2002], the tree topology is explicitly fixed. In our present paper, we have the freedom to choose whichever tree topology we want to minimize the block cost. And each value of B leads to a different tree topology (see Sections 2, 5, and 6).

However, the following subsection shows that, remarkably, there exists a single fixed-topology binary tree that works for all values of B. That is, we can force this particular pointer structure asymptotically for free and then "pack" those nodes into blocks. No other tree topology can give asymptotically better bounds than our fixed topology-plus-packing solution. The math is this paper is in the same spirit as what appears earlier in the paper, but once we require a universal binary tree structure, the math becomes somewhat more involved.

Now we have a fixed-topology atomic-key B-tree plus block-packing strategy, which, for all values of B, leads to asymptotically optimal searches. Next we insure ourselves that split-and-refine still works even when keys have variable sizes, but this is relatively straightforward. Finally, we invoke split-and-refine, obtaining a universal layout (linearization of the tree nodes), which is asymptotically optimal for all values of B.

A Fixed-Topology Atomic-Key Binary Tree

We define the ***min-third-pivot binary tree*** $\boldsymbol{T_{\mathrm{mtp}}(A)}$ on a sorted array A recursively. First, we divide the array into three sections so that each of them contains $N/3$ keys. We pick the smallest-length element in the middle section as the root r of the tree. All the elements to the left of r go into the left subtree, and all the elements to the right of r go into the right subtree. The left and right children of r are chosen recursively.

This binary tree has height at most $\log_{3/2} N$. Although all leaves do not have the same depth, the height of its shortest branch, which is at least $\log_3 N$, is at most $\frac{\log_3 N}{\log_{3/2} N} = \log_3 3/2 = 1 - \log_3 2$ times shorter than the height of the longest branch.

A Fixed-Topology Atomic-Key B-Tree

We now explain how to build a B-tree by "blocking" appropriate nodes of T_{mtp}. Denote the average length of the N keys, $\{\kappa_i\}$, by $\bar{K}$. Let

$$h = \max\left\{1, \left(\frac{B}{\bar{K}}\right)^{\log_3 2}\right\} \tag{41}$$

$$d = \max\left\{j \mid 2^j \le h\right\} \tag{42}$$

$$g = 2^{d+1} - 1. \tag{43}$$

We collect the root of T_{mtp} together with all its descendants up to depth d as the pivot keys in the root of the fixed-topology atomic-key B-tree. These g pivots divide the entire

array into $g+1$ sets that form the children of the root of the B-tree. We denote each of these sets by $\{\mathcal{S}_i\}_{1\le i\le g+1}$ and the average length of keys in them by $\bar{K}_i$. The children nodes of B-tree root are built on $\{\mathcal{S}_i\}_{1\le i\le g+1}$ recursively using the same procedure as above.

Base case. The base case of the recursive algorithm occurs when a child set $\mathcal{S}_i$ has total length less than B or when there is a single element remaining. Here, we assign all of $\mathcal{S}_i$ to a single leaf node.

Coalescing the leaf nodes. To support fast range queries and save space we coalesce all the leaves in order in a single chunk of space of size $O(N\bar{K})$. Note that for a range query, not all the keys in a range are stored in leaves. However, the two search queries for the boundaries of the range down the fixed-topology atomic-key B-tree contain all the elements in a range query. Therefore, there is no additional cost of fetching them beyond the boundary search costs.

Size of the Root Node in a Fixed-Topology Atomic-Key B-Tree

In the next two lemmas, we bound the size of the root in a fixed-topology atomic-key B-tree.

LEMMA 7.1. *Suppose that $\bar{K} \le B/3$. Then the root node of the fixed-topology atomic-key B-tree has size less than $5B$ and thus fits within 5 blocks.*

PROOF. Since $\bar{K} \le B/3$, we have that $\left(B/\bar{K}\right)^{\log_3 2} \ge 2$, $h \ge 2$, $d \ge 1$, and $g \ge 3$.

The description of the min-third-pivot binary tree T_{mtp} determines a set of keys from which each pivot was chosen. For $0 \le j \le d$, let p_i^j be the ith pivot in the level (depth) j of T_{mtp} and let $\mathcal{P}_i^j$ be the set from which p_i^j was chosen. Note that at depth j, 2^j pivots are chosen to be in the tree and thus i ranges in $\{0, 2, \cdots, 2^j - 1\}$.

Let $\hat{c}_i^j$, n_i^j, and $k_i'^j$ be the average length of keys, number of keys, and the length of the pivot in $\mathcal{P}_i^j$, respectively.

By construction of T_{mtp} we have

$$N\left(\frac{1}{3}\right)^{j+1} \le n_i^j \le \frac{N}{3}\left(\frac{2}{3}\right)^j. \tag{44}$$

The quantity we want to bound is the size of the root, $\mathcal{Q}(r) = \sum_{j=0}^{d}\sum_{i=0}^{2^j-1} k_i'^j$. At each depth j of T_{mtp}, $\bigcup_{i=0}^{2^j-1}\mathcal{P}_i^j$ is a subset of all keys and its total length is less than the total length of all keys. Therefore, we have

$$\sum_{i=0}^{2^j-1} \hat{c}_i^j n_i^j \le N\bar{K}.$$

Replacing the average key length $\hat{c}_i^j$ by the smallest key length $k_i'^j$ for each i, we obtain

$$\sum_{i=0}^{2^j-1} k_i'^j n_i^j \le N\bar{K}. \tag{45}$$

By applying the left part of 44 and canceling N from both sides, we get

$$\sum_{i=0}^{2^j-1} k_i'^j \left(\frac{1}{3}\right)^{j+1} \le \bar{K}.$$

Therefore, for each depth j, we have

$$\begin{aligned}\sum_{i=0}^{2^j-1} k_i'^j &\le 3^{j+1}\bar{K} \\ &\le 3^{j-d}3^{d+1}\bar{K} \\ &\le 3^{j-d}3^{\log_2 h+1}\bar{K} \quad \text{, since } d \le \log_2 h \\ &\le 3^{j-d+1}h^{\log_2 3}\bar{K} \\ &\le 3^{j-d+1}\left(\left(\frac{B}{\bar{K}}\right)^{\log_3 2}\right)^{\log_2 3}\bar{K} \\ &= 3^{j-d+1}B.\end{aligned}$$

Since $0 \le j \le d$, we have that

$$\begin{aligned}\mathcal{Q}(r) = \sum_{j=0}^{d}\sum_{i=0}^{2^j-1} k_i'^j &\le \sum_{j=0}^{d} 3^{j-d+1}B \\ &\le 3B + B + B/3 + \cdots \\ &\le 9B/2 \\ &\le 5B.\end{aligned}$$

□

LEMMA 7.2. *Suppose that $\bar{K} > B/3$. Then the root of a fixed-topology atomic-key B-tree contains a single key whose length is less than $3\bar{K}$ and therefore fits in at most $\lceil 3\bar{K}/B \rceil$ blocks.*

PROOF. Since $\bar{K} > B/3$, $h < 2$, $d = 0$ and $g = 1$. The rest of the proof is identical to Lemma 2.2. □

Performance of A Fixed-Topology Atomic-Key B-Tree

The following theorem bounds the expected search cost in a fixed-topology atomic-key B-tree using an induction argument. While this proof is similar to the proof of Theorem 2.3, it now has to deal with the following two issues:

(1) Unlike the atomic-key B-tree in Section 2, keys are not repeated in a fixed-topology atomic-key B-tree.
(2) The pivots of the root node in a fixed-topology atomic-key B-tree divide the array into sets that have a wider size range compared to an atomic-key B-tree. This changes the set of constraints upon which we prove the induction hypothesis. (Namely, Equation (11) in the proof of Theorem 2.3 is no longer correct in a fixed-topology atomic-key B-tree.) Therefore, we need to adapt the argument for the new sets of constraints to bound the expected search cost of a fixed-topology atomic-key B-tree.

THEOREM 7.3. *A fixed-topology atomic-key B-tree with N elements and average key size $\bar{K}$ has an expected search cost of $O(\lceil \bar{K}/B \rceil \log_{1+\lceil B/\bar{K} \rceil} N)$ block transfers when all keys are searched with equal probability. A scan of L contiguous elements of average size $\bar{K}_L$ takes $O(1 + L\bar{K}_L/B)$ additional memory transfers after the first element in the range has been examined. The data structure consumes $O(N\bar{K})$ space, that is, linear space.*

PROOF. We prove the claim by induction on N. Similar to Theorem 2.3, we prove that a fixed-topology atomic-key B-tree with N elements and average key size $\bar{K}$ has an expected search cost of $O((1 + \bar{K}/B) \log_{2+B/\bar{K}} N)$ block transfers when all leaves are searched with equal probability. Equations (5) and (7) show that the theorem statement follows from this bound.

In a fixed-topology atomic-key B-tree, the root node R comprises g keys and has $g+1$ children $\{T_i\}_{1 \le i \le g+1}$, each of which is a subtree on the set $\mathcal{S}_i$.

By induction hypothesis, for subtrees T_i of size $|\mathcal{S}_i|$ (less than N), the search cost is

$$c\left(1 + \bar{K}_i/B\right) \log_{2+B/\bar{K}_i} |\mathcal{S}_i|,$$

for some constant $c > 0$.

By Lemmas 7.1 and 7.2, the number of block transfers to fetch the root node R is at most $5 + 3\bar{K}/B$. By recursion on the subtrees of the root, the expected search cost of the tree is bounded by

$$5 + \frac{3\bar{K}}{B} + \frac{c}{N} \sum_{i=1}^{g+1} |\mathcal{S}_i| \left(1 + \frac{\bar{K}_i}{B}\right) \log_{2+B/\bar{K}_i} |\mathcal{S}_i|.$$

We need to prove that for the same constant c,

$$5 + \frac{3\bar{K}}{B} + \frac{c}{N} \sum_{i=1}^{g+1} |\mathcal{S}_i| \left(1 + \frac{\bar{K}_i}{B}\right) \log_{2+B/\bar{K}_i} |\mathcal{S}_i| \le c \left(1 + \frac{\bar{K}}{B}\right) \log_{2+B/\bar{K}} N, \tag{46}$$

subject to the following constraints:

— The tree contains N keys, but those in the root node R are not repeated in the subtrees,

$$\sum_{i=1}^{g+1} |\mathcal{S}_i| < N,$$

— the total length of all keys is $N\bar{K}$,

$$\sum_{i=1}^{g+1} |\mathcal{S}_i| \bar{K}_i < N\bar{K},$$

— and by construction, for all i,

$$N \left(\frac{1}{3}\right)^{d+1} \le |\mathcal{S}_i| \le N \left(\frac{2}{3}\right)^{d+1}.$$

Let $x_i = \bar{K}_i/B$, $x = \bar{K}/B$, and $t_i = |\mathcal{S}_i|/N$. Thus, we need to show that for some constant $c > 0$,

$$(5 + 3x) + c \sum_{i=1}^{g+1} t_i (1 + x_i) \log_{2+1/x_i} (t_i N) \le c(1 + x) \log_{2+1/x} N\,, \tag{47}$$

subject to the constraints:

$$\sum_{i=1}^{g+1} t_i < 1\,, \tag{48}$$

$$\sum_{i=1}^{g+1} t_i x_i < x\,, \tag{49}$$

$$\forall i,\ \left(\frac{1}{3}\right)^{d+1} \le t_i \le \left(\frac{2}{3}\right)^{d+1}. \tag{50}$$

We first simplify the left-hand side of (47). By (50), we obtain

$$\begin{aligned}(5+3x) + c\sum_{i=1}^{g+1} t_i(1+x_i)\log_{2+1/x_i}(t_iN) &\le (5+3x) + c\sum_{i=1}^{g+1} t_i(1+x_i)\log_{2+1/x_i}\left(N\left(\frac{2}{3}\right)^{d+1}\right)\\ &\le (5+3x) + c\ln\left(N\left(\frac{2}{3}\right)^{d+1}\right)\sum_{i=1}^{g+1}\frac{t_i(1+x_i)}{\ln(2+1/x_i)}.\end{aligned}$$

We intend to use Claim 3.1 to bound the $\sum_{i=1}^{g+1}\frac{t_i(1+x_i)}{\ln(2+1/x_i)}$ term. However, unlike the proof of Theorem 2.3, Equation (48) is not tight as required by Claim 3.1. Since the root has g pivots in it, we can rewrite Equation (48) as an equality

$$\frac{g}{N} + \sum_{i=1}^{g+1} t_i = 1. \tag{51}$$

Also, we define $z > 0$ to be a positive constant such that

$$z\frac{g}{N} + \sum_{i=1}^{g+1} t_i x_i = x. \tag{52}$$

Since $z > 0$ and $g/N > 0$, we have that

$$\frac{\frac{g}{N}(1+z)}{\ln(2+1/z)} > 0,$$

and therefore

$$\sum_{i=1}^{g+1}\frac{t_i(1+x_i)}{\ln(2+1/x_i)} \le \sum_{i=1}^{g+1}\frac{t_i(1+x_i)}{\ln(2+1/x_i)} + \frac{\frac{g}{N}(1+z)}{\ln(2+1/z)}.$$

By applying Claim 3.1, we get that

$$\sum_{i=1}^{g+1}\frac{t_i(1+x_i)}{\ln(2+1/x_i)} + \frac{\frac{g}{N}(1+z)}{\ln(2+1/z)} \le \frac{1+x}{\ln(2+1/x)},$$

and hence

$$(5+3x) + c\sum_{i=1}^{g+1} t_i(1+x_i)\log_{2+1/x_i}(t_iN) \le (5+3x) + c\ln\left(N\,(2/3)^{d+1}\right)\frac{1+x}{\ln(2+1/x)}.$$

Reorganizing the above inequality, we obtain

$$\begin{aligned}(5+3x) + c\sum_{i=1}^{g+1} t_i(1+x_i)\log_{2+1/x_i}(t_iN) &\le c(1+x)\log_{2+1/x} N + (5+3x)\\ &\quad + c(1+x)\log_{2+1/x}\left((2/3)^{d+1}\right).\end{aligned} \tag{53}$$

To prove the theorem, we must find a constant c such that the right part in Equation (53) is less than $c(1+x)\log_{2+1/x} N$, that is

$$(5+3x) + c(1+x)\frac{(d+1)\ln(2/3)}{\ln(2+1/x)} \leq 0.$$

Therefore, we derive that

$$c \geq \frac{-1}{\ln(2/3)}\frac{(5+3x)\ln(2+1/x)}{(1+x)(d+1)} \geq \frac{1}{\ln(3/2)}\frac{(5+3x)\ln(2+1/x)}{(1+x)(d+1)},$$

since $-1/\ln(2/3) = 1/\ln(3/2)$.

The following claim proves that such a c independent of x, d, and h exists.

CLAIM 7.4. *For* $x = \bar{K}/B$ *and* $h = \max\left\{1, \left(\frac{B}{\bar{K}}\right)^{\log_3 2}\right\}$, *and* $d = \max\left\{j \mid 2^j \leq h\right\}$, *there exists a constant* c *independent of* x, d, *and* h, *such that*

$$c \geq \frac{1}{\ln(3/2)}\frac{(5+3x)\ln(2+1/x)}{(1+x)(d+1)}. \tag{54}$$

Specifically, let $c = \frac{5\ln(5)}{\ln(3/2)}$.

This finishes the proof of the theorem, because we have established Equation (46).

□

It only remains to prove Claim 7.4.

PROOF OF CLAIM 7.4. There are two cases.

The first case is when $B/\bar{K} < 3$, meaning that $h < 2$, $d = 0$, and $1/x < 3$. Then, we can choose $c \geq 5\ln 5/\ln(3/2)$ because

$$\frac{1}{\ln(3/2)}\frac{(5+3x)\ln(2+1/x)}{(1+x)(d+1)} \leq \frac{1}{\ln(3/2)}\frac{5\ln(2+1/x)}{(d+1)}.$$

The second case is when $B/\bar{K} \geq 3$. We have

$$h = \left(\frac{B}{\bar{K}}\right)^{\log_3 2} = \left(\frac{1}{x}\right)^{\log_3 2},$$

and that

$$\frac{\log_2 h}{2} \leq d \leq \log_2 h.$$

Hence, we get

$$\frac{\ln(2+1/x)}{d+1} \leq \frac{2\ln(2+1/x)}{\log_2 h + 1},$$

since

$$\frac{1}{d+1} \leq \frac{2}{\log_2 h + 1}.$$

And

$$\frac{\ln(2+1/x)}{d+1} \leq \frac{2\ln(2+1/x)}{\log_3(2)\log_2(1/x)+1} \leq \frac{2\ln(2+1/x)}{\frac{\log_3(2)}{\ln 2}\ln(1/x)+1},$$

since

$$\log_2(1/x) = \frac{\ln(1/x)}{\ln 2}.$$

Therefore,

$$\frac{\ln(2+1/x)}{d+1} \leq \frac{2\ln(2+1/x)}{\log_3(e)\ln(1/x)+1}$$

since

$$\frac{\log_3(2)}{\ln 2} = \log_3(e).$$

Since the function $\ln(2+1/x)/\ln(1/x)$ is increasing in x and $x \leq 1/3$, we have that

$$\frac{\ln(2+1/x)}{d+1} \leq \frac{2\ln(5)}{\log_3(e)\ln(3)+1} \leq \frac{2\ln(5)}{1+1} \leq \ln(5).$$

Hence, we can choose

$$c \geq \frac{5\ln(5)}{\ln(3/2)},$$

as in case 1. □

Building Cost of a Fixed-Topology Atomic-Key B-Tree

To build a fixed-topology atomic-key B-tree we utilize the average-length and minimum-length data structures from Section 4. However, unlike the exposition given above, we do not build the binary tree first and then go on to block it as a B-tree. Instead, we use the binary tree analogy to figure out which nodes will be placed in the root of the B-tree and which sections will become its subtrees.

Based on Lemma 7.1 and Lemma 7.2, a similar argument to Lemma 4.1 shows the following lemma.

LEMMA 7.5. *The root node of a fixed-topology atomic-key B-tree can be built using* $O(g + \bar{K}/B)$ *block transfers.*

Finally, a similar argument to Theorem 2.4 shows that a fixed-topology atomic-key B-tree can be built in $O(\text{linear scan})$ block transfers, which is optimal.

THEOREM 7.6. *On a pre-sorted set of keys, a fixed-topology atomic-key B-tree can be built in* $O(1 + N\bar{K}/B)$ *block transfers and* $O(N)$ *processor operations.*

PROOF. The block transfer argument is the same argument from Theorem 2.4 by using Lemma 7.5. As for the processor operations, we first notice that the average-length and minimum-length query techniques of Section 4 support $O(1)$ processor operations per query. Furthermore, notice that during the recursive construction of the fixed-topology atomic-key B-tree, each block transfer that contains x keys incurs at most $O(x)$ processor operations. Therefore, the whole tree can be built in $O(N)$ processor operations. □

Cache-Oblivious Layout

To produce the cache oblivious layout, we use the "split-and-refine" technique [Alstrup et al. 2002], which converts any family of layouts with known block sizes into a layout with unknown block size, while increasing the expected number of I/Os in a search by at most a constant factor. We use the construction algorithm for the fixed-topology

atomic-key B-tree as a black box, and we compute different layouts for several carefully chosen block sizes.

We start with the block size $B_0 = 2^{\lceil \log N\bar{K} \rceil}$ and we refer to it as the *level of detail* 0. Since $B_0 \geq N\bar{K}$, the fixed-topology atomic-key B-tree is just one node and its search cost is 1. We halve the block size repeatedly until we reach a block size B_1 for which the expected search cost of the fixed-topology atomic-key B-tree for B_1 is between 2 and 4. We compute the fixed-topology atomic-key B-tree for B_1 and refer to the blocked partition of the tree as *level of detail* 1. By analogy, B_{i+1} is derived by halving B_i repeatedly until the expected search cost of the fixed-topology atomic-key B-tree for B_{i+1} is between 2 and 4 times the expected search cost of the fixed-topology atomic-key B-tree for B_i. As before, we refer to the blocked partition of this fixed-topology atomic-key B-tree as *level of detail* $i + 1$.

The partitions of different levels of detail are inconsistent with each other because a block at one level might not be wholly contained in a block at smaller levels of detail. We intend to create a *recursively consistent* structure among these partitions. Therefore, we *refine* the level of detail i according to the partitions of all smaller levels of detail $< i$. This means that the separation of nodes in the smaller levels of detail precede the bigger levels of detail.

To achieve this refinement for all different levels of detail at once, we incorporate a simple sorting mechanism. For every level of detail, we number each block in the partition produced by the fixed-topology atomic-key B-tree by a unique number. Each node in the block gets the same label as the block. For each key, we collect these labels into a *meta-label* with at most $\lceil \log N\bar{K} \rceil + 1$ components. The significance of each component in the meta-label is determined by the level of detail the label is coming from, where the most significant component is level of detail 0. A simple radix sort on these meta-labels achieves the simultaneous refinement of all levels of detail for all keys. We then layout elements in memory according to this ordering.

The following theorem is a rephrased version of Theorem 4 from [Alstrup et al. 2002].

THEOREM 7.7 (REPHRASED FROM [ALSTRUP ET AL. 2002]). *Given a black-box algorithm to build a layout for a fixed block size B, the split-and-refine algorithm technique creates a cache-oblivious layout. On any fixed given block size B, the expected search cost for the cache-oblivious layout is within a constant factor of the expected search cost of the black-box layout.*

Hence we get the following corollary that characterizes the performance and construction cost of a cache-oblivious static atomic-key B-tree.

COROLLARY 7.8. *The split-and-refine algorithm technique creates a cache-oblivious static atomic-key B-tree that for any given fixed B has the same expected search time as a fixed-topology atomic-key B-tree asymptotically. This layout can be built in in $O(N \log N\bar{K})$ processor operations.*

PROOF. There are at most $L = \lceil \log N\bar{K} \rceil + 1$ calls to the black-box algorithm for building a fixed-topology atomic-key B-tree. If the black-box layout can be built in $T(N, B)$ and there are L levels of detail in the split-and-refine layout, then the building time of the cache-oblivious layout is $O\left(\sum_{\ell=0}^{L} T\left(N, 2^{\ell}\right)\right)$. Hence, by Theorem 7.6, the total number of processor operations from these calls is

$$O\left(\sum_{\ell=0}^{L} O\left(N\right)\right) = O(N \log N\bar{K}).$$

Furthermore, the radix sort algorithm takes $O(N \log N\bar{K})$ processor operations as well because each key has $\lceil \log N\bar{K} \rceil + 1$ components. □

8. RELATED WORK

The B-tree [Bayer and McCreight 1972; Comer 1979; Knuth 1973] has been the most important data structure for storing on-disk data for four decades. Most algorithmic descriptions of B-trees assume unit-size keys but there has been work on variable-size keys since the 70s.

McCreight [McCreight 1977] studies the problem of how to provably optimize the search performance of B-trees when records have variable sizes. Since all leaves have the same depth, the problem is how to assign records to pages to minimize the height of the resulting tree. The approach is to minimize the sum of the key lengths of elements from one level to be "promoted" to the parents. As long as the records of each size are uniformly distributed within the file, their construction algorithm results in low-height B-trees. Diehr and Faaland [Diehr and Faaland 1984] and Larmore and Hirshberg [Larmore and Hirschberg 1985] develop faster algorithms for finding these elements.

Several papers [Gotlieb 1981; Huang and Viswanathan 1990; Vaishnavi et al. 1980; Becker 1994] give dynamic programs for constructing optimal B-trees and K-ary trees with unit-size elements. There are "element weights" indicating the probability that a given element in the tree is the target element and "gap weights" indicating the probability that that the target element lies between two contiguous elements in the tree. Rosenberg and Snyder [Rosenberg and Snyder 1981] study the tradeoff between space and time optimality in B-trees.

There exist optimal dynamic dictionaries designed to store different-sized keys. The ***string B-tree*** [Ferragina and Grossi 1999] supports searches, inserts, and deletes of a key κ in $O(|\kappa| / B + \log_B N)$ block transfers. Thus, the additional cost to access a key κ is just the additive cost, $O(1 + |\kappa| / B)$, to read key κ plus the cost to search in a B-tree, which is optimal. There also exist cache-oblivious (i.e., memory-hierarchy universal) string dictionaries with similar performance [Bender et al. 2006; Brodal and Fagerberg 2006]. These data structures support strings, not atomic keys. That is, key comparisons do not happen in a single step. Rather, different parts of the keys are compared at different times, and the keys can be chopped up and distributed among different parts of the data structure.

Provably good static and dynamic dictionaries for B-trees with different-sized atomic keys were presented by Bender, Hu, and Kuszmaul [Bender et al. 2010]. Bender et al. also gave a dynamic-programming algorithm to build a B-tree with minimal expected cost, when there is a search probability distribution for each key.

The present paper serves as the journal version for [Bender et al. 2010], but it also contains follow-up work. Specifically, the present paper gives a cache-oblivious data structure for static searching among different-sized atomic keys. Unlike some of the data structures in this paper, the data structures from [Bender et al. 2010] are not cache oblivious, since they are explicitly parameterized by B.

The cache-oblivious results in this paper build upon the problem of how to lay out a fixed-topology tree in memory: Specifically, given a fixed-topology tree and a probability distribution on the leaf nodes for searches, what is the optimal layout in external memory, such that the expected number of memory transfers for a search is minimized. Because the topology of the tree is fixed, the objective is to assign tree nodes to disk blocks to minimize the search cost. Gil and Itai [Gil and Itai 1999] give optimal and near optimal algorithms for the problem. Alstrup et al. [Alstrup et al. 2002] give faster algorithms for the problem and also give cache-oblivious solutions.

9. CONCLUSION

As mentioned in Section 8, much of the related work employs dynamic programs for building various kinds of optimal trees. Often those programs build trees of uniform depth to provide worst-case search bounds, whereas this paper gives expected bounds in terms of average key size. One open question is whether it is possible to build a tree that has a worst-case search time within a constant factor of optimal as well as expected bounds matching those in this paper.

As illustrated in Section 6, some of the problems we consider in this paper may be solved using dynamic programming. Trees built with dynamic program may be optimal or near-optimal, but the analysis is not parameterized by average key size, something that B-tree users often find useful.

This paper focuses on asymptotics, rather than optimizing constants. Honing the constants may be important for squeezing out performance and for building a B-tree variant that also supports front compression.

REFERENCES

Alok Aggarwal and Jeffrey Scott Vitter. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (1988), 1116–1127.

Stephen Alstrup, Michael A. Bender, Erik D. Demaine, Martin Farach-Colton, J. Ian Munro, Theis Rauhe, and Mikkel Thorup. 2002. Efficient Tree Layout in a Multilevel Memory Hierarchy. arXiv:cs.DS/0211010. (November 2002). http://www.arXiv.org/abs/cs.DS/0211010.

Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1, 3 (Feb. 1972), 173–189.

Rudolf Bayer and Karl Unterauer. 1977. Prefix B-trees. *ACM Trans. Database Syst.* 2, 1 (1977), 11–26.

Peter Becker. 1994. A new algorithm for the construction of optimal B-trees. *Nordic J. of Computing* 1, 4 (1994), 389–401.

Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. 2000. Cache-Oblivious B-Trees. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*. 399–409.

Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. 2005. Cache-Oblivious B-Trees. *SIAM J. Comput.* 35, 2 (2005), 341–358.

Michael A. Bender, Ziyang Duan, John Iacono, and Jing Wu. 2002. A Locality-Preserving Cache-Oblivious Dynamic Dictionary. In *Proc. of the 13th Annual Symposium on Discrete Mathematics (SODA)*. 29–38.

Michael A. Bender, Ziyang Duan, John Iacono, and Jing Wu. 2004. A Locality-Preserving Cache-Oblivious Dynamic Dictionary. *Journal of Algorithms* 3, 2 (2004), 115–136.

Michael A. Bender and Martin Farach-Colton. 2000. The LCA Problem Revisited. In *Proc. Latin American Theoretical INformatics (LATIN)*. 88–94.

Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. 2006. Cache-Oblivious String B-trees. In *Proc. 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. 233–242.

Michael A. Bender, Haodong Hu, and Bradley C. Kuszmaul. 2010. Performance Guarantees for B-trees with Different-sized Atomic Keys. In *Proc. 29th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. 305–316.

Omer Berkman and Uzi Vishkin. 1993. Recursive Star-Tree Parallel Data Structure. *SIAM J. Comput.* 22, 2 (1993), 221–242.

Gerth Stölting Brodal and Rolf Fagerberg. 2006. Cache-oblivious string dictionaries. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 581–590.

Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. 2002. Cache Oblivious Search Trees via Binary Trees of Small Height. In *Proc. 13th Annual Symposium on Discrete Algorithms (SODA)*. 39–48.

Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. 1995. External-Memory Graph Algorithms. In *Proc. 6th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 139–149.

William A. Clark, Kent A. Salmond, and Thomas S. Stafford. 1969. Method and Means for Generating Compressed Keys. US Patent 3,593,309. (3 Jan. 1969).

Douglas Comer. 1979. The Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137.

Erik D. Demaine, Gad M. Landau, and Oren Weimann. 2009. On Cartesian Trees and Range Minimum Queries. In *Proc. 36th International Colloquium on Automata, Languages and Programming (ICALP) (Lecture Notes in Computer Science)*, Vol. 5555. Springer, 341–353.

George Diehr and Bruce Faaland. 1984. Optimal pagination of B-trees with variable-length items. *Commun. ACM* 27, 3 (1984), 241–247.

Paolo Ferragina and Roberto Grossi. 1999. The String B-tree: A New Data Structure for String Search in External Memory and its Applications. *J. ACM* 46, 2 (1999), 236–280.

Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-Oblivious Algorithms. In *Proc. 40th Annual Symposium on Foundations of Computer Science (FOCS)*. 285–297.

Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 2012. Cache-Oblivious Algorithms. *ACM Transactions on Algorithms* 8, 1 (2012), 4.

Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. 1984. Scaling and Related Techniques for Geometry Problems. In *Proc. 16th Annual ACM Symposium on Theory of Computing (STOC)*. 135–143.

Joseph Gil and Alon Itai. 1999. How to pack trees. *J. Algorithms* 32, 2 (1999), 108–132. DOI:http://dx.doi.org/10.1006/jagm.1999.1014

Leo Gotlieb. 1981. Optimal Multi-Way Search Trees. *SIAM J. Comput.* 10, 3 (1981), 422–433.

Dov Harel and Robert Endre Tarjan. 1984. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J. Comput.* 13, 2 (1984), 338–355.

Shou-Hsuan Stephen Huang and Venkatraman Viswanathan. 1990. On the Construction of Weighted Time-Optimal B-Trees. *BIT* 30, 2 (1990), 207–215.

Donald E. Knuth. 1973. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison Wesley, Reading, MA.

Lawrence L. Larmore and Daniel S. Hirschberg. 1985. Efficient optimal pagination of scrolls. *Commun. ACM* 28, 8 (1985), 854–856.

Edward M. McCreight. 1977. Pagination of B*-trees with variable-length records. *Commun. ACM* 20, 9 (1977), 670–674.

Oracle. 2009. Oracle Berkeley DB Programmer's Reference Guide, Release 4.8. http://www.oracle.com/technology/documentation/berkeley-db/db/index.html. (August 2009).

Harald Prokop. 1999. *Cache-Oblivious Algorithms*. Master's thesis. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Arnold L. Rosenberg and Lawrence Snyder. 1981. Time- and space-optimality in B-trees. *ACM Trans. Database Syst.* 6, 1 (1981), 174–193.

Baruch Schieber and Uzi Vishkin. 1988. On Finding Lowest Common Ancestors: Simplification and Parallelization. *SIAM J. Comput.* 17, 6 (1988), 1253–1262.

Vijay K. Vaishnavi, Hans-Peter Kriegel, and Derick Wood. 1980. Optimum Multiway Search Trees. *Acta Inf.* 14 (1980), 119–133.

Robert. E. Wagner. 1973. Indexing Design Considerations. *IBM Syst. J.* 12, 4 (1973), 351–367.