Performance Guarantees for B-trees with Different-Sized Atomic Keys*

Michael A. Bender
Dept. of Computer Science
Stony Brook University
Stony Brook, NY 11794, USA
bender@cs.sunysb.edu
and
Tokutek, Inc.

Haodong Hu
Windows Division
Microsoft
Redmond, WA, 98052, USA
hu_hd@hotmail.com

Bradley C. Kuszmaul
MIT CSAIL
Cambridge, MA 02139, USA
bradley@mit.edu
and
Tokutek, Inc.

ABSTRACT

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, however, B-tree operations have no nontrivial performance guarantees.

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. The paper proves that 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 a search tree with minimal expected cost.

*Supported in part by the Singapore-MIT Alliance, DOE Grant DE-FG02-08ER25853, and NSF Grants ACI-0324974, CCF-0540897/05414009, CCF-0541209, CCF-0621439/0621425, CCF-0621511, CCF-0634793/0632838, CCF-0937822, CCF-0937860, CNS-0540248, CNS-0615215, and CNS-0627645.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PODS'10, June 6–11, 2010, Indianapolis, Indiana, USA.
Copyright 2010 ACM 978-1-4503-0033-9/10/06 ...$10.00.

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—*Indexing Methods*

General Terms: Algorithms, Theory

Keywords: B-tree with different-sized keys, atomic keys, dynamic programming

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 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 short keys as pivots near the top of the tree because short keys means a larger branching factor and a more efficient search. However, one cannot simply choose to put the shortest 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 short 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. If most nodes in a root-to-leaf path employ short 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 $O(\log_2 N)$.

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, 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 the comparison function.

We analyze the performance using the Disk Access Machine (DAM) model [1], 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, 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*** [15] 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*** [4,11,28], 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 [21].

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.

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 departure 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.

Outline

The rest of this paper is organized as follows. Section 2 discusses a greedy algorithm for building static trees. Section 3 analyzes the search cost for static trees. Section 4 shows how to build the trees efficiently. Section 5 shows how to build dynamic trees. Section 6 presents a dynamic program for

[1]Interestingly, the reference manual to Oracle Berkeley DB [24] 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.

building an optimal static search tree. Section 7 discusses work related to atomic B-trees, and Section 8 concludes with a brief discussion of open problems.

2. STATIC ATOMIC-KEY B-TREE

In this section we give a greedy algorithm for constructing a static atomic-key B-tree on N different-size keys. The idea is to store short keys near the top of the tree and long keys near the bottom while simultaneously choosing pivot keys that are spread out in the key-space.

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 leftmost N/f keys, the second set $\mathcal{C}_1$ contains the next 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.

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 keys in the ith set $\mathcal{C}_i$ and let k_i' be the minimum-length key in $\mathcal{C}_i$.

LEMMA 1. *Suppose that $\bar{K} \le B/3$. Then the root node has size less than B and thus fits within a single block.*

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

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

Because the total length of all N keys is the sum of the lengths of the keys in each set $\mathcal{C}_i$ $(0 \le i \le f-1)$, 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 shortest key length k_i' for each i, we obtain

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

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

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

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

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

as promised. □

LEMMA 2. *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 $\{\mathcal{S}_i\}_{1\le i\le f-1}$. 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 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 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 5. *The expected search cost can not be achieved for worst-case searches. Specifically*

1. *There exists a set of N keys with average key size $\bar{K}$ and maximum length at most B, for which in any tree layout the worst-case search cost (or the tree height) is a factor of $\Omega(\log(B+1))$ worse than the expected search cost in a good tree.*
2. *There exists a set of N keys with average key size $\bar{K}$ for which in any tree layout the worst-case search cost (or the tree height) is a factor of $\Omega((N/B)\log_{B+1} N)$ greater than the expected search cost in a good tree.*

Proof. For Part 1, Consider a key set in which the first $\sqrt{N}$ elements have length B 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 B that requires $\Omega(\log_2 N)$ memory transfers to reach. In contrast, if all pivots of length B are deeper than any pivot of length 1, then the expected search cost is $O(\log_{1+B} N)$ memory transfers.

For Part 2, Consider a key set in which the first element has length N and the remaining $N-1$ elements have length 1. Then the cost to search for the first element is $\Theta(N/B)$ memory transfers, and the expected search cost is $\Theta(\log_{1+B} N)$ memory transfers. $\square$

3. SEARCH ANALYSIS

In this section we prove Theorem 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 leaves are searched with equal probability.

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 $\{T_i\}_{1\le i\le f-1}$, each of which is a subtree on the set $\mathcal{S}_i$.

Assume for 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 1 and 2, when $\bar{K}\le B/3$, 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 3, we show that for the same constant c,

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

subject to the following constraints:

- The tree contains N keys, i.e.,

$$\sum_{i=1}^{f-1}|\mathcal{S}_i|=N,\tag{6}$$

- the total length of all keys is

$$\sum_{i=1}^{f-1}|\mathcal{S}_i|\bar{K}_i=N\bar{K},\tag{7}$$

- and by construction, for all i,

$$0<|\mathcal{S}_i|<2N/f.\tag{8}$$

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, (5)-(8) become respectively (9)-(12).

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

$$\begin{aligned}1+3x+c\sum_{i=1}^{f-1}t_i(1+x_i)\log_{2+1/x_i}(t_iN)\\ \le c(1+x)\log_{2+1/x} N\,,\end{aligned}\tag{9}$$

subject to the constraints:

$$\sum_{i=1}^{f-1}t_i=1\,,\tag{10}$$

$$\sum_{i=1}^{f-1}t_ix_i=x\,,\tag{11}$$

$$\forall i, 0<t_i<2/f\,.\tag{12}$$

Before proceeding further, we prove the following claim.

Claim 6. *Let $x_i>0$ and $x>0$, and let t_i be constrained as follows:*

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

Then the following inequality holds:

$$\sum_{i=1}^{f-1}\frac{t_i(1+x_i)}{\ln(2+1/x_i)}\le\frac{1+x}{\ln(2+1/x)}\,.\tag{13}$$

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

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

Then (13) can be rewritten as

$$\sum_{i=1}^{f-1}t_ih(x_i)\le h\left(\sum_{i=1}^{f-1}t_ix_i\right).$$

The above inequality holds as long as the function $h(x)$ is concave.

To prove that the function $h(x)$ is concave, we show that its second derivative is negative. The first derivative is

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

and the second derivative is

$$h''(x)=\frac{2+2x-(3x+1)\ln(2+1/x)}{(2x^2+x)^2\ln^3(2+1/x)}\,.\tag{14}$$

Because $x > 0$, $\ln(2 + 1/x) > 0$. Thus, the numerator of (14) 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

$$\ln(2 + y) > \frac{2y + 2}{y + 3} = 2 - \frac{4}{y + 3} \quad (y > 0). \tag{15}$$

The derivative of the left side of (15) is

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

and the derivative of the right side of (15) 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 (15) has an initial value greater than the right side of (15), mean that (15) holds.

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

Our objective is to establish (9)-(12). We first simplify the lefthand side of (9). By (12), we obtain

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

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

$$\begin{aligned} & 1 + 3x + c\sum_{i=1}^{f-1} t_i(1 + x_i)\log_{2+1/x_i}(t_i N) \\ \leq\ & 1 + 3x + c\ln(2N/f)\sum_{i=1}^{f-1}\frac{t_i(1 + x_i)}{\ln(2 + 1/x_i)}. \end{aligned} \tag{16}$$

From Claim 6 and (16), we obtain

$$\begin{aligned} & 1 + 3x + c\sum_{i=1}^{f-1} t_i(1 + x_i)\log_{2+1/x_i}(t_i N) \\ \leq\ & 1 + 3x + c\ln(2N/f)\frac{1 + x}{\ln(2 + 1/x)}. \end{aligned} \tag{17}$$

Reorganizing the above inequality, we obtain

$$\begin{aligned} & 1 + 3x + c\sum_{i=1}^{f-1} t_i(1 + x_i)\log_{2+1/x_i}(t_i N) \\ \leq\ & c(1 + x)\log_{2+1/x} N + 1 + 3x \\ & -c(1 + x)\log_{2+1/x}(f/2). \end{aligned} \tag{18}$$

To prove the theorem, we need find the constant c such that the right part in (18) is less than $c(1 + x)\log_{2+1/x} N$, that is

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

Therefore, we derive that

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

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

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

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

Claim 7. *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*

$$c \geq 3\frac{\ln(2 + 1/x)}{\ln(f/2)}. \tag{19}$$

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 $c \geq 3\ln 5/\ln(3/2)$.

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

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

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

$$\begin{aligned} \frac{\ln(2 + 1/x)}{\ln(f/2)} &\leq \frac{\ln(3 + \lfloor 1/x\rfloor)}{\ln(f/2)} \leq \frac{\ln(2\lfloor 1/x\rfloor)}{\ln(f/2)} \\ &= \frac{\ln(2f)}{\ln(f/2)} \leq \frac{\ln 6}{\ln(3/2)}. \end{aligned}$$

The first inequality is by $1/x \leq 1 + \lfloor 1/x\rfloor$; The second inequality follows from $\lfloor 1/x\rfloor \geq 3$; 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 $f \geq 3$. Thus, we can choose $c = 3\ln 6/\ln(3/2)$ to satisfy (19) for both cases. □

In conclusion, by (18) and for the constant c from Claim 7, we establish

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

4. BUILDING AN ATOMIC KEY B-TREE

This section presents a construction algorithm for atomic-key B-trees, proving Theorem 4. 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. Then recursively build each subtree. The algorithmic challenge is to select the pivots quickly.

By way of comparison, a naive 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 5 indicates.

We now give a faster way to select pivot elements. The trick is to preprocess the keys to answer two different kinds of queries.

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 [16], and the inputs can be preprocessed in linear time to answer constant-time queries [6, 8, 19, 26].

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 [10]. 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 [6] 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, [13] gives a cache-oblivious solution to the 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 first fits 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 8. *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 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 8 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 9. *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 Section 2. 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 10. *A dynamic atomic-key B-tree with N elements and average key size $\bar{K}$ has the following performance bounds:*

1. *The expected search cost is $O(\lceil \bar{K}/B \rceil \log_{1+\lceil B/\bar{K} \rceil} N)$ transfers when all leaves are searched with equal probability.*
2. *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. *The tree has linear size, i.e., size $O(N\bar{K})$.*
4. *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. *The cost to insert or delete an element κ 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, and κ fits within a constant fraction of memory.
6. *The 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

$$f = \max\left\{2, \left\lfloor \frac{B}{\bar{K}} \right\rfloor\right\}, \quad (20)$$

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 11. *Suppose that $\bar{K} \leq B/2$, and let $\beta > 0$ be a constant. If we choose a representative key r_i from $\mathcal{C}_i$ such that $r_i \leq \beta\hat{c}_i$, 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,

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

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 is has size $O(\bar{K})$.

LEMMA 12. *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

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

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 13. *Let $\beta > 1$ be a constant. 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

$$|\mathcal{C}_i| = |\mathcal{C}_i^s| + |\mathcal{C}_i^\ell| = N/f\,. \quad (21)$$

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,

$$\hat{c}_i|\mathcal{C}_i| \geq \beta\hat{c}_i|\mathcal{C}_i^\ell| + |\mathcal{C}_i^s| > \beta\hat{c}_i|\mathcal{C}_i^\ell|. \tag{22}$$

By (21) and (22), 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 with 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 ***(β, γ)-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 13.

Corollary 14. *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 15. *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 14, which bounds the number of elements before r_1,

$$\left(1 - \frac{1}{2\beta}\right)\frac{N}{f} \leq |\mathcal{S}_1| \leq \left(1 + \frac{1}{2\beta}\right)\frac{N}{f}. \tag{23}$$

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 14,

$$\left(1 - \frac{1}{\beta}\right)\frac{N}{f} \leq |\mathcal{S}_i| \leq \left(1 + \frac{1}{\beta}\right)\frac{N}{f}. \tag{24}$$

The lemma follows from (23) and (24). □

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

Lemma 16. *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 10.

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

Lemma 17. *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 smaller than r_i (those outside the range from which r_i was chosen) and at least $N/(\beta f)$ elements greater than or equal to r_i (those in the range from which r_i was chosen).

Since then, at most $N/(3\beta f)$ elements smaller than r_i were inserted and at most $N/(3\beta f)$ elements larger than r_i were deleted. Thus, at most $(1 - 2/(3\beta))N/f$ elements are smaller 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 small 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 by finding representative elements as described in Section 4.

The combination of chunks and splits ensures that Properties 2 and 3 of Theorem 10 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 10 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 17), 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.

A node v is rebalanced when there have been $N/(3\beta f)$ inserts or deletes into the subtree rooted at v since its 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 18. *Consider a dynamic atomic-key B-tree with N elements and average element size $\bar{K} = O(B)$. 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 the interval between two consecutive rebalances of a node. Denote the average element size during the first rebalance as $\bar{K}_1$. We give an amortized analysis for paying for the rebalance by the inserts/deletes in between.

The cost for the rebalance is $O(1 + N\bar{K}_1/B)$. We charge the rebalance cost to the $\Theta(N/f)$ inserts/deletes that take place before the second rebalance. Thus, the amortized rebalance cost per insertion is

$$O(f\bar{K}_1/B) = O(1 + \bar{K}_1/B) = O(1). \qquad (25)$$

□

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

LEMMA 19. *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 17, 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 18, 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 19 establishes Properties 5 and 6 of Theorem 10 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 19 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 (25)) 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 is $O(1 + N\min\{\bar{K}, B\}/B)$.

Insertions and Deletions

With indirection we obtain the following generalizations of Lemmas 18 and 19.

LEMMA 20. *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. Consider the interval between two consecutive rebalances of a node. Denote the average element size during the first rebalance as $\bar{K}_1$.

The cost for the rebalance is $O(1+N\min\{\bar{K}_1, B\}/B)$. We charge the rebalance cost to the $\Theta(N/f)$ inserts/deletes that

take place before the second rebalance. If $\bar{K}_1 = O(B)$, the amortized rebalance cost per insertion is

$$O(f\bar{K}_1/B) = O(1 + \bar{K}_1/B) = O(1).$$

If $\bar{K}_1 = \Omega(B)$, the amortized rebalance cost per insertion is

$$O(f) = O(1).$$

□

LEMMA 21. *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 19. By Lemma 21, 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 21 establishes Properties 5 and 6 of Theorem 10.

6. OPTIMAL DYNAMIC PROGRAM

This section presents a dynamic program that produces an optimal static search tree for N atomic keys, $\kappa_1, \ldots, \kappa_N$, where each key κ_i has a probability p_i of being queried. The optimal structure minimizes the expected number of blocks transfered 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 optimally joining an optimal tree for the set $\kappa_{r+1}, \ldots, \kappa_j$ with space less than $S - |\kappa_r|$ in the root and an optimal tree for the set $\kappa_i, \ldots, \kappa_{r-1}$.

Figure 1 illustrates the dynamic program.

For keys $i, \ldots, j$, let

$$K_{i,j} = \sum_{i \le \ell \le j} |\kappa_\ell|$$

and

$$P_{i,j} = \sum_{i \le \ell \le j} p_\ell.$$

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.

Overloading notation, let p be 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. Define the search cost in terms of a function $F(i, r, j, k)$, as follows:

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

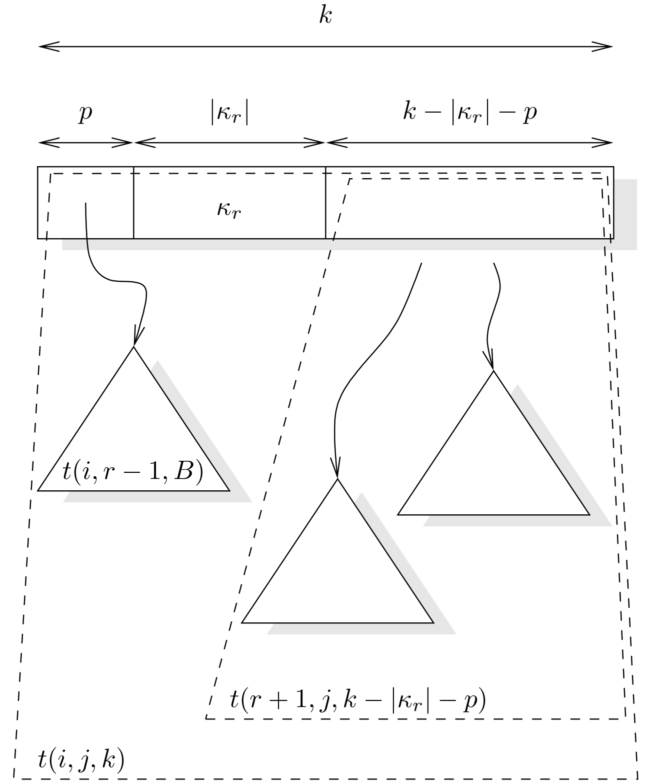

Figure 1: The dynamic program produces a tree $t(i, j, k)$ by finding an optimal way of allocating the root block of size k among a pointer of size p to a left subtree, a key κ_r, and a right subtree. The left subtree has root block of size B and holds keys $\kappa_i \ldots \kappa_{r-1}$, and incurs 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 $k - |\kappa_r| - p$ to hold its root.

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

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

7. RELATED WORK

The B-tree [3, 12, 21] 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 [23] 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 [14] and Larmore and Hirshberg [22] develop faster algorithms for finding these elements.

Several papers [5, 18, 20, 27] 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 [25] 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*** [15] of Ferragina and Grossi 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. Refs. [7, 9] give cache-oblivious (i.e., memory-hierarchy universal) string dictionaries with similar performance. 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.

Another related problem on variable-sized keys is the following. For a given probability distribution on the leaf nodes, how to lay out a fixed-topology tree in 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 [17] optimal and near optimal algorithms for the problem. Alstrup et al. [2] give faster algorithms for the problem and also give cache-oblivious solutions.

8. CONCLUSION

As mentioned in Section 7, 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 programing may be optimal or near-optimal, but the analysis is not parameterized by the 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.

Finally, one major open question is whether there exists a cache-oblivious atomic-key B-tree that attains the bounds presented here.

Acknowledgments

We would like to thank Martin Farach-Colton, Simai He, Margo Seltzer, and Marc Tchiboukdjian for helpful discussions in early stages of this work.

9. REFERENCES

[1] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[2] Stephen Alstrup, Michael A. Bender, Erik D. Demaine, Martin Farach-Colton, J. Ian Munro, Theis Rauhe, and Mikkel Thorup. Efficient tree layout in a multilevel memory hierarchy. arXiv:cs.DS/0211010, November 2002. `http://www.arXiv.org/abs/cs.DS/0211010`.

[3] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, February 1972.

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

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

[6] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Proceedings of Latin American Theoretical INformatics (LATIN)*, pages 88–94, 2000.

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

[8] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993.

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

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

[11] William A. Clark IV, Kent A. Salmond, and Thomas A Stafford. Method and means for generating compressed keys. US Patent 3,593,309, 3 January 1969.

[12] Douglas Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[13] Erik D. Demaine, Gad M. Landau, and Oren Weimann. On cartesian trees and range minimum queries. In *Proc. 36th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 5555 of *Lecture Notes in Computer Science*, pages 341–353. Springer, 2009.

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

[15] Paolo Ferragina and Roberto Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.

[16] Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th Annual ACM Symposium on Theory of Computing (STOC)*, pages 135–143, 1984.

[17] Joseph Gil and Alon Itai. How to pack trees. *J. Algorithms*, 32(2):108–132, 1999.

[18] L. Gotlieb. Optimal multi-way search trees. *SIAM J. Comput.*, 10(3):422–433, 1981.

[19] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[20] Shou-Hsuan Stephen Huang and Venkatraman Viswanathan. On the construction of weighted time-optimal B-trees. *BIT*, 30(2):207–215, 1990.

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

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

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

[24] Oracle. Oracle Berkeley DB programmer's reference guide, release 4.8. `http://www.oracle.com/technology/documentation/berkeley-db/db/index.html`, August 2009.

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

[26] Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.

[27] Vijay K. Vaishnavi, Hans-Peter Kriegel, and Derick Wood. Optimum multiway search trees. *Acta Inf.*, 14:119–133, 1980.

[28] R. E. Wagner. Indexing design considerations. *IBM Syst. J.*, 12(4):351–367, 1973.