Cache-Oblivious String B-trees

Michael A. Bender
Stony Brook University
bender@cs.sunysb.edu

Martin Farach-Colton
Rutgers
farach@cs.rutgers.edu

Bradley C. Kuszmaul
MIT CSAIL
bradley@mit.edu

ABSTRACT

B-trees are the data structure of choice for maintaining searchable data on disk. However, B-trees perform suboptimally

- when keys are long or of variable length,
- when keys are compressed, even when using *front compression*, the standard B-tree compression scheme,
- for range queries, and
- with respect to memory effects such as disk prefetching.

This paper presents a *cache-oblivious string B-tree* (COSB-tree) data structure that is efficient in all these ways:

- The COSB-tree searches asymptotically optimally and inserts and deletes nearly optimally.
- It maintains an index whose size is proportional to the front-compressed size of the dictionary. Furthermore, unlike standard front-compressed strings, keys can be decompressed in a memory-efficient manner.
- It performs range queries with no extra disk seeks; in contrast, B-trees incur disk seeks when skipping from leaf block to leaf block.
- It utilizes all levels of a memory hierarchy efficiently and makes good use of disk locality by using cache-oblivious layout strategies.

Categories and Subject Descriptors: E.1 [**Data Structures**]: Arrays, Trees; E.5 [**Files**]: Sorting/searching; H.3.3 [**Information Storage and Retrieval**]:

General Terms: Algorithms, Experimentation, Performance, Theory.

Keywords: cache oblivious string B-tree, locality preserving front compression, packed-memory array, range query, rebalance.

1. INTRODUCTION

For over three decades, the B-tree [4, 16] has been the data structure of choice for maintaining searchable data on disk. B-trees maintain an ordered set of *keys* and allow insertions, deletions, searches, and range queries. Most implementations employ B^+-trees [16, 26], in which the full keys are all stored in the leaves, but for convenience we refer to all the variations as "B-trees."

Traditional B-trees perform suboptimally in several respects:

- The theoretical and practical performance of B-trees degrades when keys are long or vary in length.
- Keys often share large prefixes, and are thus typically stored using *front compression* [5, 15, 26, 32] within blocks. Front compression exhibits a tradeoff between the compression factor and the memory locality for decompression. B-trees make this tradeoff suboptimally.
- Range queries use disk hardware inefficiently because each leaf block fetched may require a random disk seek. Random block accesses perform two orders of magnitude more slowly than sequential block accesses for disk. Seeks to nearby tracks are nearly an order of magnitude faster than random seeks.
- B-trees do not take advantage of memory effects such as disk prefetching, especially when variable amount of prefetching is performed.

Inefficiency in each of these respects can reduce performance significantly. Although some of these issues are addressed individually in the literature, as discussed below, there are no previously known search structures that address all these issues effectively.

This paper presents a *cache-oblivious string B-tree* (COSB-tree) data structure that is efficient in all four respects. The rest of this section states our results.

Variable-length keys. Traditional B-trees do not handle large keys well. Typically, they pack small keys in blocks, but for large keys the pack a pointer to the key, which is stored elsewhere. They choose arbitrarily which key to promote to a parent when a block is split and often bias their choice toward promoting long keys. In principle, it is better to store short keys near the top of the tree, but it is also better to split the search space into nearly even pieces. There are no known techniques for addressing both issues simultaneously in a dynamically changing B-tree.

The *string B-tree* [19] handles keys of unbounded size efficiently. In the string B-tree, an *insertion* of a new key κ uses $O(1 + \|\kappa\|/B + \log_B N)$ block transfers, where $\|\kappa\|$ is the length of key κ and N is the number of keys in the tree. This block transfer complexity is nearly optimal in the traditional Disk Access Machine model [1], where there is a single fixed memory-transfer size, B, since it trivially takes at least $\|\kappa\|/B$ transfers to read κ, and $O(\log_B N)$ is the cost of insertion in a B-tree, even when all keys have unit length. See Figure 1 for a glossary of notation.

κ	Key	$\mathcal{D}$	Set of keys (a dictionary)
Q	Query result (a key set)	$\lvert\mathcal{D}\rvert$	Number of keys in $\mathcal{D}$
B	Block size	$\|\mathcal{D}\|$	Sum of key lengths in $\mathcal{D}$
N	$N = \lvert\mathcal{D}\rvert$	$\langle\langle\mathcal{D}\rangle\rangle$	Size of front compressed $\mathcal{D}$

Figure 1: Glossary of symbols.

A *range query* for keys κ and κ' returns all keys μ such that $\kappa \le \mu \le \kappa'$ lexicographically. In a string B-tree, a range query uses $O(1 + (\|\kappa\| + \|\kappa'\| + \|Q\|)/B + \log_B N)$ block transfers, where Q is the set of result keys and $\|Q\|$ is the sum of their lengths.[1] A *search*

This research was supported in part by the Singapore-MIT Alliance and NSF Grants ACI-0324974, CCR-0208670, and CCR-9820879.

PODS'06 June 26–28, 2006, Chicago, Illinois USA

[1] The complexity of range queries, as reported in [19], has an ad-

for a single key κ is a special case of a range query and therefore uses $O(1+\|\kappa\|/B+\log_B N)$ block transfers.

The string-B-tree paper [19] also describes a *prefix-search* operation, but a prefix search is a special case of a range query. A slower version of the string B-tree [19] also supports *substring queries*. Since our goal is to support heterogeneous key sizes in traditional database applications, we do not address substring queries here and compare our results to the faster string B-tree that does not support substring queries.

The string B-tree is deterministic and the performance bounds given are worst-case (that is, not amortized), but it does not support compression, disk prefetching, or disk-seek-efficient range queries. The cache-oblivious string B-tree we present here also supports keys of unbounded length efficiently. It is amortized for updates and randomized, but is efficient with respect to compression, prefetching, and disk seeks.

Compression. Many practical B-trees (e.g. [30]) employ *front compression* [5, 15, 26, 32] within blocks to reduce the amount of memory required for the keys, but string B-trees [19] do not. Thus, the size of a string B-tree is $\Theta(\|\mathcal{D}\|)$. Front compression reduces space by storing each key κ as a pair $\langle \ell, s\rangle$, where ℓ is the length of the longest common prefix between κ and κ's predecessor, and s is the suffix of κ starting at position $\ell+1$. This strategy pays off when keys are stored lexicographically, which maximizes the average longest common prefix between adjacent keys. Although front compression is not optimal with respect to the entropy bound of the strings, it is used in many implementations of B-trees.

To decode a key, one decodes the previous key. This procedure might require scanning back to the beginning of the entire dictionary. To mitigate this problem, each node of a B-tree is front compressed separately. "Blocked" front compression may yield a poor compression rate compared to front compression of the entire dictionary, however. Thus, there is a tradeoff between the effectiveness of front compression and the cost of decompression. Larger blocking improves the former and worsen latter.

We introduce a modified front-compression scheme for the cache-oblivious string B-tree that simultaneously achieves a decompression complexity that is linear in key length and an overall compression that is within an arbitrarily small constant of the optimal front compression. Specifically, we compress dictionary D into $(1+\varepsilon)\langle\!\langle D\rangle\!\rangle$ bits, where pure front compression uses $\langle\!\langle D\rangle\!\rangle$ bits, and we decode a key κ in $O(\|\kappa\|/\varepsilon B)$ memory transfers, for any $\varepsilon > 0$. Using our improved front-compression scheme, our cache-oblivious string B-tree uses space $O(\langle\!\langle D\rangle\!\rangle)$ to store dictionary D.

Cache-obliviousness, disk prefetching, and range queries. Both traditional and string B-trees are based on the assumption that there is a single block-transfer size B for which the data structure should be optimized. For example, many B-trees used in practice are optimized assuming that the unique block-transfer size is 4096 bytes. If the keys are of constant size, then a B-tree achieves a fanout of $\Theta(B)$, which implies that a search uses $O(1+\log_B N)$ block transfers.

Real memory systems are not so simple. The memory hierarchy is composed of several levels of cache, main memory, and disk, but there are other levels between these. For example, the disk cache sits between main memory and the rest of the disk. When the disk system services a request for a disk block, the disk first checks whether the requested block is in the disk cache, and if so, returns the block. Otherwise, the disk seeks to the appropriate track, reads the block, and prefetches the track. Thus, the *effective* block-transfer size may be much larger than a disk block. Indeed, sequential disk-block accesses typically run over two orders of magnitude faster than random disk-block accesses. However, disk tracks vary in size and heavy loads can pollute the relatively small disk cache quickly, evicting blocks before they have a chance to be requested. Therefore the effective block-transfer size between these two levels of memory is highly variable.

The *cache-oblivious (CO)* model [21] enables us to design algorithms that achieve data locality simultaneously for all block sizes. Such algorithms are on-line optimal with respect to changing block sizes in the disk system as well as simultaneously optimal at all levels of the memory hierarchy. In the cache-oblivious model, the objective is to minimize the number of transfers between two levels of the hierarchy. However, unlike traditional external-memory models [1], the parameters B, the block size, and M, the main-memory size, are unknown to the coder or the algorithm. The main idea of the CO model is that if it can be proved that some algorithm performs a nearly optimal number of memory transfers in a two-level model with unknown parameters, then the algorithm also performs a nearly optimal number of memory transfers on any unknown, multilevel memory hierarchy. Cache-oblivious data structures are more portable than traditional external-memory data structures, since they avoid any tuning parameters. CO algorithms do not try to measure the machine's cache and adjust their behavior accordingly. Rather, they are optimized for every level of granularity throughout their execution without any tuning.

It has been shown [9–11, 13] how to implement cache oblivious B-trees for fixed-size keys. The cache-oblivious B-tree supports efficient range queries because of the *packed-memory array (PMA)* structure [9], which maintains the search keys tightly packed in order in memory. Thus, a range query consists of one memory-optimal search, followed by a scan within an array. This scan does not incur any more random disk seeks since the items being scanned are physically in order on disk. In contrast, B-trees may have their relatively-small leaf blocks scattered throughout a disk in any order, and if the effective block size is large, then range queries are far from optimal.

Recently, Brodal and Fagerberg [12] describe a static cache-oblivious string B-tree. It supports cache-oblivious searches with the same bounds as the original string B-tree but does not allow updates. Their paper works by physically laying out the tree in memory with duplications, and it seems difficult to make it dynamic.

In this paper we present a B-tree structure that supports variable-size key insertions, deletions and searches, near optimal front compression and decoding, and is cache-oblivious.

Experimental motivation. Cache-oblivious data structure have interesting theoretical properties, as outlined above. Here we present experimental validation for their on-disk performance. Previous work has focus on their in-memory performance.

We implemented B-trees from the literature for fixed size keys. We placed versions of each data structure into a memory-mapped file, taking care that the data structure was significantly larger than main memory. For static B-trees we employed a breadth-first layout: The root block appears first in the file, followed by the children of the root, followed by the first child's children, and so forth. For static CO B-trees, we used a van Emde Boas layout [29]. The static trees were packed 100% full with data, and since the trees are static, we did not even allocate space for pointers to the children.

ditive $|Q|$ term, which appears because the keys are stored in no particular order to speed up insertions. This data layout means that actually obtaining the various strings may require one extra block transfer per key. Using now-standard techniques, the bound can be improved to the bound we show here.

Data structure	Average time per search	
	small-machine	big-machine
CO B-tree	12.3ms	13.8ms
Btree: 4KB Blocks:	17.2ms	22.4ms
16KB blocks:	13.9ms	22.1ms
32KB blocks:	11.9ms	17.4ms
64KB blocks:	12.9ms	17.6ms
128KB blocks:	13.2ms	16.5ms
256KB blocks:	18.5ms	14.4ms
512KB blocks:		16.7ms

Figure 2: Performance measurements of 1000 random searches on static trees.

We implemented a dynamic B+tree [16] and a dynamic CO B-tree [10, 11].

We ran our experiments on two different machines. The small machine is a 300MHz Pentium II with 128MB of RAM and a 4.3GB ATA disk running Redhat 8.0, Linux Kernel 2.4.20. The large machine is a 4-processor 1.4GHz Opteron 840 with 16GB of memory and a 72GB IBM Ultrastar 10,000RPM SCSI-320 disk running SUSE Linux 2.4.19.

Figure 2 shows the results of our experiments on static trees. We measured the time to perform 1000 searches on random keys. For each measurement, before starting the first search, we flushed the filesystem cache by remounting the filesystem.

For static trees, it is clear that the advertised disk-block size of 4096 bytes is far too small, underperforming big-block B-trees and CO B-trees by 30–50%. But very large blocks perform poorly as well. On the small machine the optimal block size is 32KB, and on the big machine it is 256KB. On the small machine, the B-tree managed to outperform the CO B-tree by 3% in the best case, but in all other cases the CO B-tree outperformed the B-trees. We conclude that although there are some situations where a carefully tuned static B-tree can squeeze out an advantage against a static CO B-tree, static CO B-trees provide much more robust performance and can usually outperform even carefully tuned static B-trees.

Figure 3 shows the results for dynamic CO B-trees based on the PMA construction of [9]. This dynamic CO B-tree has good amortized performance, but very occasionally must rebalance the entire tree, which is expensive when the tree does not fit in main memory. Figure 3 shows that for inserting the first 440,000 random elements, the CO B-tree outperforms any of the traditional B-trees. Big-block B-trees perform poorly for insertions. But sometime before the 450,000th insertion, the CO B-tree reorganizes its whole data structure, at which point it falls behind the small-block B-trees by about a factor of two. For range-queries and random searches where all the leaves of the tree are scanned in order, the big-block B-trees outperform the small-block B-trees, and slightly beat the CO B-tree. For many applications, the big-block B-trees would have unacceptable costs for insertions, and the small-block B-trees are not as fast as the CO B-tree. This suggests that CO B-trees could be a practical way to improve performance of databases and file systems.

As a sanity check, we compared the performance of our traditional and CO B-trees to the Berkeley DB [30], a high-quality commercially available B-tree. The Berkeley DB with the default buffer-pool allocation is much slower than our implementation, but is comparable once the parameters are tuned. Berkeley DB supports variable-sized keys, crash recovery, and very large databases, none of which our implementation supports, and so one should not read too much into these data. It simply suggests that we did a reasonable job implementing our B-trees.

Our experiment is biased in favor of the B-trees because the B-trees were "young," that is, blocks are allocated sequentially. The

	Block Size	insert 440,000 random values	insert 450,000 random values	range query of all data	1000 random searches
	CO B-tree	15.8s		4.6s	5.9s
	CO B-tree		54.8s	9.3s	7.1s
Sequential block allocation:	2K		19.2s	24.8s	12.6s
	4K		19.1s	23.1s	10.5s
	8K		26.4s	22.3s	8.4s
	16K		41.5s	22.2s	7.7s
	32K		71.5s	21.4s	7.3s
	64K		128.0s	11.5s	6.5s
	128K		234.8s	7.3s	6.2s
	256K		444.5s	6.5s	5.3s
Random block allocation:	2K		3928.0s	460.3s	24.3s
Berkeley DB (256 KB pool):			1201.1s		
Berkeley DB (64 MB pool):			76.6s		

Figure 3: Timings for memory-mapped dynamic trees. The keys are 128 bytes long. The range query is a scan of the entire data set after the insert. Berkeley DB was run with the default buffer pool size (256KB), and with a customized loader that uses 64MB of buffer pool. These experiments were performed on the small machine.

dynamic CO B-tree data structure ages well, whereas B-trees age poorly, a fact well documented in the context of filesystems [31]. We simulated an aged B-tree in which the blocks are randomly placed on disk (shown as "Random block allocation" in Figure 3, and found that all operations, including insertions and range queries can slow down dramatically, sometimes by two orders of magnitude. (Perhaps this setup should be called "super-aged", since real B-trees are unlikely ever to allocate their blocks completely randomly.) We view the fact that the CO B-trees do not age as a significant advantage for databases and filesystems.

Summary of Results. In this paper we present a solution to the variable-key-length indexing problem. Our new data structure, the *cache-oblivious string B-tree* is simultaneously efficient for all block sizes and has the following performance:

- Insertions require $O(1+\|\kappa\|(\log^2\langle\!\langle\mathcal{D}\rangle\!\rangle)/B+\log_B N)$ memory transfers with high probability (w.h.p.).
- Searches and successor/predecessor queries, require an optimal $O(1+\|\kappa'\|/B+\log_B N)$ memory transfers w.h.p..
- Range queries require an optimal $O(1+(\|\kappa\|+\|\kappa'\|+\langle\!\langle Q\rangle\!\rangle)/B+\log_B N)$ block transfers w.h.p.. The result set Q is returned in compressed representation and can be decompressed in an additional $O(\|Q\|/B)$ memory transfers, which is optimal for front compression. Because COSB-trees store all keys in order on disk, range queries involve no extra disk seeks.
- The space usage is $O(\langle\!\langle D\rangle\!\rangle)$. In contrast, string B-trees and per-block front-compressed B-trees use more space, $O(\|D\|)$ and $O(\min\{\|D\|, B\langle\!\langle D\rangle\!\rangle\})$, respectively.
- The COSB-tree is cache oblivious. Thus, it is on-line optimal with respect to disk prefetching and efficient at all levels of the memory hierarchy.

An important component of the COSB-tree, of independent interest, is the *front-compressed packed-memory array* (FC-PMA) data structure. The FC-PMA maintains a collection of strings $\mathcal{D}$ stored in order, with a modified front compression. The FC-PMA has the following properties:

- For any ε, the space usage of the FC-PMA can be set to $(1+\varepsilon)\langle\!\langle\mathcal{D}\rangle\!\rangle$, while enabling a string κ to be reconstructed with $O(1+\|\kappa\|/(\varepsilon B))$ memory transfers.
- Inserting and deleting a string κ into an FC-PMA requires $O(\|\kappa\|(\log^2\langle\!\langle\mathcal{D}\rangle\!\rangle)/(\varepsilon B))$ memory transfers.

The advantage of the COSB-tree, as summarized above, is that keys are kept physically in sorted order on disk, so that range

queries use a minimum number of disk seeks. If we relax our conditions to match those of the string B-tree, that is, range queries return pointers to keys and the strings are not kept compressed, we can match the string B-tree bounds in an amortized sense. That is, we can achieve $O(1+\|\kappa\|/B+\log_B N)$ amortized transfers to insert κ by using the amortized scanning structure from [6]. In many applications, range queries must retrieve the strings, so we present the version outlined above, without the structure from [6]. Moreover, in many applications, keeping order on disk is worth some loss in theoretical bounds, in order exploit to prefetching mechanisms by the disk and operating system.

Roadmap. The rest of this paper is organized as follows. In Section 2, we describe a internal-memory algorithm for dictionary matching that forms the basis for the COSB-tree. In Section 3, we give a static COSB-tree, and explain the static version of locality-preserving front compression. In Section 4, we show how to dynamize this static structure, including an explanation of the FC-PMA .

2. DICTIONARY MATCHING IN INTERNAL MEMORY

In this section we review a internal memory (RAM) data structure for the dictionary-matching problem [3,28], which we develop into the COSB-tree. In the *dictionary-matching problem* the goal is to preprocess a dictionary $\mathcal{D}$ of keys $\{\kappa_1, \kappa_2, \ldots, \kappa_N\}$ to answer the following queries:

- MEMBER(κ): Determine whether $\kappa \in \mathcal{D}$.
  RAM time: $O(\|\kappa\| + \log N)$.
- PRED(κ): Return the maximum $\kappa' \in \mathcal{D}$ such that $\kappa' < \kappa$.
  RAM time: $O(\|\kappa\| + \|\text{PRED}(\kappa)\| + \log N)$.
- SUCC(κ): Return the minimum $\kappa' \in \mathcal{D}$ such that $\kappa' > \kappa$.
  RAM time: $O(\|\kappa\| + \|\text{SUCC}(\kappa)\| + \log N)$.

We solve this problem using divide-and-conquer by exploiting the following observation: Let $\mathcal{T}$ be the compacted trie[2] of $\mathcal{D}$. Then there is a *centroid* vertex ρ in $\mathcal{T}$ that has at least $N/3$ and at most $2N/3$ descendants. Throughout, we identify a trie node with the string obtained by tracing from the root to that node. To answer MEMBER(κ) we determine whether ρ is a prefix of κ (in which case, we say that κ *matches* ρ). If κ matches ρ, then we recurse into the trie rooted at ρ, the so-called *down trie*. Otherwise, we recurse into the trie obtained by excluding ρ and its subtree, the so-called *up trie*. Either way, we eliminate a constant fraction of the trie from consideration. The *centroid tree* of $\mathcal{T}$ is obtained by making ρ the root, where ρ's children are the recursively defined centroid trees of ρ's up and down tries.

To achieve the required time bounds, we cannot simply compare the letters of ρ and κ to determine if they match. Each comparison could take time $\Omega(\|\kappa\|)$, yielding a run time that could be $\Omega(\|\kappa\| \log N)$. Instead, we employ a hash function $\mathcal{H}$ (such as for Karp-Rabin fingerprinting [24], CRC, or MD5) that maps strings into integers. Our requirement is that we can compute the fingerprint of every prefix of ρ in time $O(\|\rho\|)$ in a RAM model and that unequal strings collide with polynomially small probability.

To speed up matching, we preprocess the dictionary by computing the hashes of all compacted trie nodes. Then, to test membership of κ, we compute the hashes of all the prefixes of κ. Now, whenever we want to match a prefix of κ with a node in $\mathcal{T}$, we compare the hash values. The algorithm, as stated, is Monte Carlo; we might get a false positive on the matching. To make this algorithm Las Vegas, we do a character-by-character match only when the hash values match. Since the probability of a mismatch is low, this does not increase our running time, with high probability.

[2]A compacted trie is a trie where all nonbranching paths are reduced to edges.

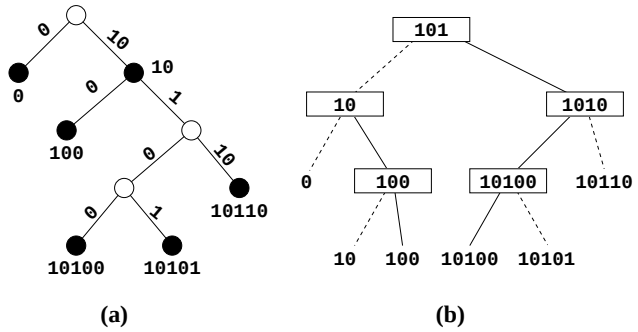

Figure 4: An example of the RAM data structure. Part (a) shows the compressed trie representation of a dictionary. Each node in the compressed trie corresponds to a string. The strings that are in the dictionary are represented by black nodes, whereas the strings not in the dictionary are represented by white nodes. For example, the string `0` is not in the dictionary. Edges are labeled with strings. Part (b) shows the centroid tree for the trie. Internal nodes of the tree are shown as rectangles. Each internal node of the tree is labeled with a string τ. Each internal node has a solid line linking it to the root of its down trie (the leaves of which have τ as a prefix), and a dotted line linking it to the root of its up trie (the leaves of which do not have τ as a prefix). The leaves are shown as unadorned strings, and are shown in order (lexicographically from left to right). Each leaf string maintains pointers (not shown) to its predecessor and successor. Each internal node maintains a pointer to the leftmost and rightmost matching leaf from the entire dictionary (not shown.)

Figure 4 shows an example of the RAM data structure. Figure 4 (a) shows a compressed trie containing several strings, and Figure 4 (b) shows the centroid tree. To search the trie for $\kappa =$`10101`, follow the `10` edge from the root, then follow the `1` edge, then follow the `0` edge, and then follow the `1` edge. In contrast, to search the centroid tree for the same key, start at the root, where κ matches $\tau =$`101` so follow the solid line. Key κ matches `1010` so follow the solid line. It does not match `10100` so follow the solid line which leads to a leaf.

Now we describe how to perform a successor or predecessor query efficiently. If $\kappa \in \mathcal{D}$, we can perform a membership query to find κ in the dictionary and then use a doubly linked list to find the predecessor or successor. But if $\kappa \notin \mathcal{D}$, we must do something else. Consider tracing down from the root of $\mathcal{T}$ with κ. If κ is not in $\mathcal{D}$, then at some point we match as far as some node τ and into the edge between τ and one of its children ρ, but we do not match as far as ρ. If the first mismatch between κ and the trie is because κ has a `0` where the $\overline{\tau\rho}$ edge has a `1`, then κ is lexicographically less than all strings below ρ, and its successor is ρ's leftmost trie descendant (in the entire trie). Otherwise, by symmetry, κ's predecessor is ρ's rightmost trie descendant.

Thus, we can always find either a predecessor or successor if every node in the centroid tree keeps track of its lexicographically least and greatest trie descendants. If we want the predecessor but find the successor, or vice versa, we can traverse the linked list of leaves forward, respectively backward, to find the desired key.

For example, in Figure 4, consider the problem of searching for $\kappa =$`1010` which is not actually a member of the dictionary. As we descend the trie we find κ matches `101`, and does not match `1010`, leading us to leaf `10110` which is not what we want. We backtrack up the tree to the child of the last node that matched ($\tau =$`101`). That

node has a pointer to the leftmost key that has prefix **101**. That key is **10100**, which is the successor of **1010** in the dictionary.

We conclude with the following lemma:

LEMMA 1 ([3]). *The dictionary-matching problem can be solved within the bounds $O(\|\kappa\| + \log N)$ for* MEMBER(κ), $O(\|\kappa\| + \|\text{PRED}(\kappa)\| + \log N)$ *for* PRED(κ), *and* $O(\|\kappa\| + \|\text{SUCC}(\kappa)\| + \log N)$ *for* SUCC(κ) *on a RAM.*

In fact, on a RAM, we can solve this problem trivially by direct trie traversals without the $\log N$ additive factor, but we will employ this centroid method in the following to achieve good data locality.

3. STATIC COSB-tree

In this section we present a static cache-oblivious string B-tree. Our data structure preprocesses a set $\mathcal{D}$ of N keys $\{\kappa_1, \kappa_2, \ldots, \kappa_N\}$ to answer the following query types efficiently in the cache-oblivous model:

- MEMBER(κ): Determine whether $\kappa \in \mathcal{D}$.
  Memory transfers: $O(1 + \log_B N + \|\kappa\|/B)$.
- PRED(κ): Return κ' where κ' is the predecessor of κ in $\mathcal{D}$.
  Memory transfers: $O(1 + \log_B N + \|\kappa\|/B + \|\kappa'\|/B)$.
- SUCC(κ): Return κ' where κ' is the successor of κ in $\mathcal{D}$.
  Memory transfers: $O(1 + \log_B N + \|\kappa\|/B + \|\kappa'\|/B)$.
- RANGE-QUERY(κ_i, κ_j): Given two keys $\kappa_i, \kappa_j \in \mathcal{D}$, return a compressed representation of all keys in the set $Q = \{\kappa \in \mathcal{D} | \kappa_i \le \kappa \le \kappa_j\}$.
  Memory transfers: $O(1 + \log_B N + (\|\kappa_i\| + \|\kappa_j\| + \langle\langle Q \rangle\rangle)/B)$, plus $O(1 + \|Q\|/B)$ to uncompress results.

We first present a relatively simple COSB-tree, which only supports queries on uncompressed data. We then present a variation on front compression [5, 15, 32] that permits uncompressing a key κ with only $O(\|\kappa\|/B)$ block transfers.

3.1 Static COSB-tree with no compression

A COSB-tree with no compression is made up of two pieces, an array of keys stored in lexicographic order and a centroid tree for faster searching of the data. The centroid tree is just as described in Section 2, except that the leaves of the tree point to locations in the array of keys, and the tree is laid out to achieve good cache-oblivious performance. The centroid tree has depth $O(\log N)$, but not all leaves have the same depth. There are several ways [2, 9, 17, 22, 29] to lay out such a tree in memory to achieve optimal cache-oblivious searching, that is, with $O(\log_B N)$ memory transfers. However, these techniques require the tree to be processed in a batch, whereas we need a layout that will lend itself to dynamization in Section 4.

The *weight* of a node in a tree is defined to be the total number nodes in the subtree rooted at the node. We exploit the fact that centroid trees are *weight balanced*, that is, for each node, 1 plus the weight of the left subtree is within a constant factor of 1 plus the weight of the right subtree of that node. The constant turns out to be 2 for centroid trees. The rest of this subsection describes a modified van Emde Boas (vEB) layout for weight-balanced binary trees.

The standard approach for laying out a tree in memory is to cut the tree along a frontier so that the top tree and each of the bottom trees have size roughly $\sqrt{N}$. The original layout in [29] did this partitioning by selecting bottom-tree roots by height. The difficulty in applying this method here is that centroid-tree leaves have nonuniform depth. Nonetheless, it is possible to adapt height-based partitioning to centroid trees, but we do not know how to maintain such a layout dynamically. Instead, we use the weight of a node to select it for the frontier, as follows.

Given integer w, we say that a node is *selected by* w if both that node and its sibling have weight at least w, and neither of their children are selected by w. That is, we select the deepest nodes that have weight at least w and whose siblings also have weight at least w. Selected nodes have the following property:

LEMMA 2. *All nodes selected by w have weight at least w and at most $3w$.*

Proof. If any node u has weight greater than $3w$, then both of u's children have weight at least w, because centroid trees are weight balanced with a constant of 2. If both of u's children have weight at least w, then the children would be selected, rather than u. □

Define the *hyperfloor* of x, denoted $\lfloor\lfloor x \rfloor\rfloor$, to be $2^{\lfloor \lg x \rfloor}$. Thus, the hyperfloor rounds x down to the nearest power of 2. Let the *hyper-hyperfloor* be $\lfloor\lfloor\lfloor x \rfloor\rfloor\rfloor = 2^{\lfloor\lfloor \lg x \rfloor\rfloor}$. Thus, the hyperhyperfloor rounds x down to the nearest power of a power of 2.

To lay out a centroid tree, we select nodes by weight $w = \lfloor\lfloor\lfloor N \rfloor\rfloor\rfloor$. We call the resulting nodes the roots of the *bottom recursive subtrees* $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_z$, and call the remaining tree, above, the *top recursive subtree* $\mathcal{C}_0$. We now lay out $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_z$ in memory in that order, recursively with selection weight $\sqrt{\lfloor\lfloor\lfloor N \rfloor\rfloor\rfloor}$.

For this static construction, it would also work to select nodes by weight $N^{1/2}, N^{1/4}, N^{1/8}$ and so forth, rather than arranging for the weights to always be powers of powers of 2. We use powers of powers of 2 because it is convenient in Section 4. The key insight for either construction is that the selection weights must be all the same for recursive subtrees at a given *level of detailed*, defined as follows. Each level of detail is a partition of the tree into disjoint recursive subtrees. At the coarsest level of detail the entire tree forms the unique recursive subtree. At the finest level of detail, 0, each node forms its own recursive subtree with selection weight 2^{2^0}. In general, at level-of-detail k we view the tree as partitioned into recursive subtrees with selection weight 2^{2^k}. The key property of the layout is that, at any level of detail, each recursive subtree is stored in a contiguous block of memory.

It is straightforward to lay out trees $\mathcal{C}_1, \ldots, \mathcal{C}_z$ recursively because they are as weight-balanced as $\mathcal{C}$, i.e., 2-balanced. However, $\mathcal{C}_0$ is only 4-balanced. If we were to lay out $\mathcal{C}_0$ in the same way, then $\mathcal{C}_0$'s recursive subtree would only be 16-balanced. Instead, we employ the following strategy for laying out recursive subtrees that do not contain leaves of $\mathcal{C}$. Suppose that we want to find recursive subtrees with selection weight 2^{2^i} above nodes with selection weight 2^{2^j}. Then we select nodes weight $2^{2^i+2^j}$ to be the roots of the bottom recursive subtrees.

LEMMA 3. *Subtrees containing leaves have size one to three times their selection weight. Subtrees that do not contain leaves have size between one third and three times their selection weight.*

LEMMA 4. *This nonuniform layout of a weight 2-balanced binary tree incurs $O(\log_B N)$ block transfers on a root-to-leaf traversal.*

THEOREM 5. *This static COSB-tree represents a set $\mathcal{D}$ of N elements, and supports member, predecessor, successor, and range queries. The operation* MEMBER(κ) *runs in $O(1 + \log_B N + \|\kappa\|/B)$ memory transfers w.h.p., and* PRED(κ), *and* SUCC(κ) *run in $O(1 + \log_B N + \|\kappa\|/B + \|\kappa'\|/B)$ memory transfers w.h.p., where κ' is the predecessor (resp. successor) of κ. The operation* RANGE-QUERY(κ, κ') *runs in $O(1 + \log_B N + (\|\kappa\| + \|\kappa'\| + \|Q\|)/B)$ transfers, where Q is set of keys in the result. These results hold in the cache-oblivious model with the tall-cache assumption.*

Proof. There are two cases:

Case 1: $\|\kappa\| = O(M)$, i.e., the key is small compared to memory. Computing the Karp-Rabin fingerprints takes $1+\kappa/B$ memory transfers, and all keys remain in internal memory while we search in the centroid tree.

Case 2: $\|\kappa\| = \Omega(M)$, i.e., the key is large compared to memory. In this case, the Karp-Rabin fingerprints that we compute cannot fit in memory at the same time. Thus, since we query $O(\log N)$ fingerprints, the number of memory transfers is $O(\log N + \log_B N + \|\kappa\|/B + 1)$. However, the $O(\log N)$ term is dominated as long as $\log N < \|\kappa\|/B$. Since the CO model is transdichotomous ($B = \Omega(\log N)$) [20] and assuming the cache is tall ($M = \Omega(B^2)$) [21], $\log N \leq M/B \leq \|\kappa\|/B$.

The scan bounds are trivially obtained. □

3.2 Locality-preserving front compression

In this subsection we show how to add compression to our static COSB-tree. We develop a new strategy for achieving front compression without high decoding cost. The front-compressed data then replaces the array of keys used in the static COSB-tree above.

Front compression works as follows: Given a sequence of keys $\kappa_1, \kappa_2, \ldots, \kappa_i$ to store, a naive representation requires $\sum_j \|\kappa_j\|$ memory. Instead, we let π_{j+1} be the longest common prefix of κ_j and κ_{j+1}. In this case, we can remove nearly $\sum_j \|\pi_j\|$ memory from the representation by representing the keys as

$$\kappa_1, \|\pi_2\|, \sigma_2, \|\pi_3\|, \ldots, \|\pi_i\|, \sigma_i$$

where σ_j is the suffix of κ_j after removing the first π_j bits. To *decode* κ_j, one concatenates the first π_j bits from κ_{j-1} to σ_j. Finding the first π_j bits of κ_{j-1} may require further decoding, possibly resulting in expensive decoding. Front compression, which is a lossless compression scheme, requires the the same space as the (uncompacted) trie for $\mathcal{D}$ [26]. The total size of a front-compressed set of keys $\mathcal{D}$ is written as $\langle\!\langle \mathcal{D} \rangle\!\rangle$.

Front and rear compression are described in [14, 15, 32]. Reference [26] describes front compression in an exercise, but provides less detail. Reference [5] argues that front and rear compression are particularly important for secondary indices. Front compression is relevant for compressing the keys stored at the leaves of a search tree, whereas rear compression is essentially used only in the indices, and is subsumed by the string-B-tree techniques presented here and in [19].

Our goal is to achieve $O(1 + \|\kappa\|/B)$ memory transfers to decompress any key in $\mathcal{D}$, but to store $\mathcal{D}$ with $O(\langle\!\langle \mathcal{D} \rangle\!\rangle)$ space. The challenge is that, for front compression, uncompressing a single key may require scanning back through the entire compressed representation. This is a well known problem for front compression. One common strategy is to compress enough keys to fill some predefined block and to start the compression over when that block is full. This idea does not provide any theoretical bounds, however: the compression achieved can be much worse than the best front compression, and a block size may be arbitrarily bigger than $\|Q\|$, so decompression also has no guarantees.

Here we show a *locality-preserving front compression (LPFC)*, which meets our goal. Our modified compression scheme begins with key κ_1. Suppose we have compressed the first $i-1$ keys and now we want to add key κ_i. We set $c = 2 + \varepsilon/2$. We scan back $c\|\kappa_i\|$ characters in the compression to see if we could decode κ_i from just this information. If so, we add π_i, σ_i as before. If not, we add $0, \kappa_i$ to the compression, that is, we do not compress key κ_i at all. Call this sequence the *locality-preserving front compression of* $\mathcal{D}$, denoted LPFC($\mathcal{D}$).

The decoding scheme is just as with standard front compression, and it immediately matches the desired bounds: decoding κ_i touches at most $c\|\kappa_i\|$ contiguous characters, and decoding Q touches $O(\|Q\|)$ contiguous characters. The remaining issue is to show that LPFC achieves a compressed dictionary of size $(1+\varepsilon)\langle\!\langle \mathcal{D} \rangle\!\rangle$.

LEMMA 6. *The total length of the LPFC($\mathcal{D}$) is at most* $(1+\varepsilon)\langle\!\langle \mathcal{D} \rangle\!\rangle$ *and every key* κ_i *can be decoded with* $O(\|\kappa_i\|/\varepsilon B)$ *block transfers.*

Proof. Call any key κ that has been inserted without front compression a *copied key*. Denote as *native* any characters in the compression that are not copied (that is, characters that appear in the full front-compressed version of $\mathcal{D}$). Denote the preceding $c\|\kappa\|$ characters as the *left extent* of κ. Notice that if κ is a copied key, there can be no copied key beginning in the left extent of κ. However, a copied key may end within κ's left extent.

We consider two cases. In the first case, the preceding copied key ends at least $c\|\kappa\|/2$ characters before κ. Then, we say that κ is *uncrowded*. In the second case the preceding copied key κ ends within $c\|\kappa\|/2$ characters of κ. Then, we say that κ is *crowded*.

Partition the sequence of all copied keys just before each uncrowded key. We call each such subsequence a *chain*. Note that each chain begins with an uncrowded key and is followed by a sequence of crowded keys.

Furthermore, the lengths of these crowded keys decrease geometrically. To see this, consider a crowded key κ. Since κ's predecessor in the chain, κ', must begin before κ's left extent, it must have length at least $c\|\kappa\|/2$.

Thus, if κ is uncrowded, the kth crowded key in its chain has length at most $\|\kappa\|(2/c)^k$. The total length of all keys in a chain starting at κ is thus at most $c\|\kappa\|/(c-2)$.

Finally, charge the cost of copying these keys to the $c\|\kappa\|/2$ characters preceding the uncrowded key at the beginning of the chain. This charge is at most $2/(c-2) = \varepsilon$ per character. □

THEOREM 7. *The static COSB-tree with front compression represents a set $\mathcal{D}$ of N elements, and supports member, predecessor, successor, prefix, and range queries. The operation* MEMBER(κ) *runs in* $O(1 + \log_B N + \|\kappa\|/B)$ *memory transfers w.h.p., and* PRED(κ) *and* SUCC(κ) *run in* $O(1 + \log_B N + \|\kappa\|/B + \|\kappa'\|/B)$ *memory transfers w.h.p., where κ' is the predecessor (resp. successor) of κ. The operation* RANGE-QUERY(κ, κ') *runs in* $O(1 + \log_B N + (\|\kappa\| + \|key'\| + \langle\!\langle Q \rangle\!\rangle)/B)$ *transfers. The compressed keys can be decoded for an additional $\|Q\|/B$ transfers. All results hold in the cache-oblivious model with the tall-cache assumption.*

4. DYNAMIC COSB-trees

In this section we dynamize the COSB-tree. We use a combination of cache-oblivious data-structure tools, such as van Emde Boas (vEB) layouts [29] and packed-memory arrays (PMAs) [9, 23], but none of these are strong enough for our purposes. For this paper we need augmented versions of these tools. In the following, we present an overview of the three parts of a dynamic COSB-tree. We then give a detailed description of each in turn.

The Data Structure

The dynamic COSB-tree consists of three pieces. The top piece, called the *centroid tree*, is a dynamic version of the centroid tree described in Section 3, i.e., a binary tree of depth $O(\log N)$. The

centroid tree is embedded into a packed-memory array with a dynamic cache-oblivious layout, so that a root-to-leaf traversal in the centroid tree requires only $O(\log_B N)$ transfers. The centroid tree is built upon only $\Theta(N/\log N)$ keys. We use the centroid tree to find a key that is within $O(\log N)$ of our target key using $O(\log_B N)$ memory transfers. The reason to build the top tree on a sparse data set is that there is an additive $O(\log N)$ insertion cost in the top level, which is amortized away with this level of indirection.

The centroid-tree leaves point into a middle layer, called the *hashdata*. This is a packed-memory array [9] that contains $O(1)$ words of information for each key. This layer is designed to allow for fast sequential searches of predecessor and successor keys. When we enter the hashdata from the centroid tree, we are within $O(\log N)$ keys of our true successor/predecessor. We finish the search in this local neighborhood by a sequential scan, which uses $O(1+(\|\kappa\|+\log N)/B)$ memory transfers. This local search works by storing, for each key, the fingerprint of the longest common prefix of the key with it predecessor.

Once we have localized our target key, we follow a pointer to the bottom piece, another packed-memory array called the *keydata*. The keydata contains the actual keys, sorted in lexicographic order and compressed. For the compression we use a dynamic variant of our augmented front compression of Section 3. This dynamized data structure supports insertion or deletion of a key κ with $O(1+\|\kappa\|\log^2\langle\langle D\rangle\rangle/B)$ block transfers (amortized), once we have determined where the key belongs.

If we want faster updates, but the data does not need to be physically in sorted order, then we add another level of indirection, a scanning structure (see [6]), which reduces the insertion/deletion cost to $O(1+\|\kappa\|\log^{2+\varepsilon}\log\langle\langle D\rangle\rangle/B)$, for any $\varepsilon>0$. If, as with the original string B-tree, range queries return pointers to keys, not to the keys themselves, then we can use yet another level of indirection in addition to the scanning structure to match the string B-tree bounds in amortized sense while remaining cache-oblivious.

The rest of this section details the three pieces, starting with the keydata, then the centroid tree, and then the hashdata. The section concludes with an explanation of how these pieces fit together to achieve our desired bounds.

Keydata PMA

For the static COSB-tree, we showed how to implement locality-preserving front compression. For the dynamic COSB-tree, we need to support insertions and deletions while maintaining provably good compression. We employ a packed-memory array, which allows us to keeps data in order dynamically. The PMA, as originally described, supports keys of unit length. However, it directly achieves the desired bounds if we break up any long key into unit-length pieces and use the original algorithm.

We already showed how to implement cache-efficient decoding for front compression. We need another idea to implement cache-efficient insertions and deletions. We preserve the decoding invariant: if decoding key κ requires more than $\|\kappa\|/\varepsilon$ elements of the compressed representation to be scanned, then κ should be copied.

However, insertions interfere with this invariant. To see why, observe that when we insert a key, we can easily check its left extent to see if needs to be a copied key, as we did for the static case. The problem comes with keys to the right. Suppose that a key κ^* is inserted within the left extent of some key κ. If κ and κ^* are not copied keys, the insertion of κ^* may increase the decoding cost of κ to above $\|\kappa\|/\varepsilon B$ transfers. A solution would be to copy key κ, that is, to replace its compressed representation with a copied representation. However, such problem keys κ may be large, and so their left extents may be arbitrarily long. We would thus be required to look arbitrarily far to the right of κ^* to find a violation.

We present a modified compression scheme, the *Dynamic Locality-Preserving Front Compression (DLPFC)*, which preserves the compression rate, preserves the locality in the decoding, and enforces locality for insertions. To implement DLPFC, we augment the (static) LPFC with *copied prefixes*. For LPFC, each key could be coded with a pair representing the largest common prefix (lcp) with its predecessor and its suffix beyond the lcp; for DLPFC, we may now also choose to explicitly copy any prefix of the key. For LPFC, we decode keys from the last character forward; for DLPFC, we may simultaneously decode some prefix and some suffix until we meet somewhere in the middle. We will see that copied prefixes can be used to prevent the effects of an insertion from propagating too far forward.

The algorithm proceeds as follows. We first check the left extent of the inserted key κ^* to see if κ^* should be copied. If so, we insert it as a copied string and are done. Otherwise, we need to check the characters to the right of κ^*. Call the first $c\|\kappa^*\|$ characters to the right of the insertion point the *near right extent* of κ^* and the first $3c\|\kappa^*\|$ characters the *far right extent*. The lcp ℓ of κ^* and whichever key is at the end of the far right extent is the minimum lcp in the far right extent. If there is a copied key in the far right extent, or a copied prefix of length at least ℓ, then the the effects of inserting κ^* do not propagate to the end of its far right extent, in which case κ^* is inserted normally. If there is propagation, then we consider the key κ' being touch at the end of the near right extent. Let ℓ' be the lcp of κ^* and κ'. Then we change the representation of κ' to include a copied prefix of its first ℓ' characters. Furthermore, for technical reasons that will become clear in the following, we also include a copied prefix of the first ℓ' characters of κ^* in its representation.

THEOREM 8. Dynamic Locality-Preserving Front Compression *is a compression scheme that can represent a set of N keys $\mathcal{D}$ in size at most $(1+\varepsilon)\langle\langle\mathcal{D}\rangle\rangle+N$ bits so that key κ can be decoded in $O(1+\|\kappa\|/B)$ memory transfers and, given a finger to the location of insertion, key κ^* can be inserted in $O(1+\|\kappa^*\|/B\varepsilon)$ memory transfers in the CO model.*

Proof. Inserting copied prefixes in our algorithm can only improve decoding complexities, and the number of bytes scanned during insertion is linear. Now we must prove that inserting such keys does not cause too much damage to the compression.

If there is a copied prefix that starts within the near right extent of κ^*, then κ^* will not induce a prefix copy, since the effects of inserting κ^* cannot propagate beyond that copied prefix. Thus, any copied prefix after the far right extent of κ^* must have been caused by the insertion of a key κ' that either starts after the far right extent of κ^* or before κ^*. We charge κ^*'s copied prefix only to characters within κ^*'s near right extent, so we do not care about insertions after the far right extent.

Consider now the other types of insertions: a key κ' inserted before κ^* that induces a prefix copy after the far right extent of κ^*. The near (and indeed the far) right extent of κ^* is part of the near right extent of κ', so we need to make sure that we do not charge the same characters twice for prefix copying.

To keep charges from overlapping, we take any copied prefix of length ℓ at the end of a near right extent and charge it to its preceding $c\ell$ characters. Since each such copied prefix is paired with a matching size-ℓ copied prefix at the beginning of the near right extent, each character is charged $2/c$ units.

The copied prefix of κ' is of size at most $\|\kappa^*\|$ by the transitivity of lcp in lexicographically ordered strings. Thus, the charged region for κ' is of length at most $c\|\kappa^*\|$, but it begins at least $3c\|\kappa^*\|$

after κ^*, and thus cannot overlap the near right extent, and particularly the charged region of κ^*. Therefore, no character gets charged twice. As before, set $\varepsilon = 2/c$. □

Centroid tree: Dynamic layout of weight-balanced trees

In this section we show how to maintain the vEB layout of a dynamically changing weight-balanced tree. This approach was already used in the first CO B-tree [8, 9], but now we show how to support faster dynamic updates more efficiently and on more general trees.

Recall that a tree is *weight balanced* if for all nodes, one plus the weight of the left subtree is within a constant factor of one plus the weight of the right subtree. For (static) centroid trees, this constant is 2 in the worst case. For dynamic centroid trees, we need a constant greater than 2.

By allowing for c-weight balance, for constant $c > 2$, we obtain the following guarantee: A node v only gets out of balance every $\Omega(\text{WEIGHT}(v))$ insertions or deletions of nodes that are descendants of v (see, e.g., [27]).

This property of insertions means that whenever v falls out of balance, we can afford to scan all of v's subtree for a total amortized cost of $O(\log N)$ work and $O(1+(\log N)/B)$ memory transfers per update (see, e.g., [27, Theorem 5]). In principle, this ability to scan descendants enables us to maintain the vEB layout of the centroid tree dynamically; if a node falls out of balance, then we can afford to rebuild the whole subtree and its vEB layout. We show the following:

LEMMA 9. *There exists a dynamic van Emde Boas layout of a weight-balanced tree in a PMA, where the amortized rebalance cost is $O(1+(\log^2 N)/B)$ per update. This layout has the property that whenever a node v is in a rebalance interval of the PMA, then so are all of v's descendants.*

Proof. We maintain the sorted order of tree nodes in the vEB layout in memory by storing the nodes in a *packed-memory array (PMA)* [9]. The PMA stores N elements in sorted order in a $\Theta(N)$-sized array, subject to insertions or deletions. When we insert/delete an element in the array, we scan left and right to find a neighborhood of the array whose density is "within threshold." Then we *rebalance* the neighborhood, i.e., we spread out the elements uniformly in the range. Thus, in principle, the amortized cost to insert a node v into the weight-balanced tree is $O(1+(\log^2 N)/B)$ to make room in the PMA plus $O(1+(\log N)/B)$ to re-layout the tree.

Unfortunately, this analysis is incomplete because it does not account for the cost to maintain the pointers in the tree as nodes shift around in the PMA. To understand the problem, consider an upper recursive subtree $\mathcal{U}$ and lower recursive subtrees $\mathcal{L}_1\mathcal{L}_2\mathcal{L}_3\ldots$ laid out in order $\mathcal{U}\mathcal{L}_1\mathcal{L}_2\mathcal{L}_3\ldots$. An insert near the "left" part of $\mathcal{L}_1$ may cause many nodes in $\mathcal{U}$ to move around in the PMA. In order to maintain the child pointers, we also maintain parent pointers, but maintaining parent pointers causes trouble. If we move some nodes in $\mathcal{U}$, which are high up in the tree, then we also have to follow the child pointers of these moved nodes to update the parent pointers of the children. Unfortunately, these children may be spread out in the vEB layout causing one memory transfer per child for a total update cost of $O(\log^2 N)$ memory transfers.[3]

[3] In the conference version of the original CO B-tree [8] this problem was partially solved using "dummy nodes"; specifically, as much space as possible was added in the PMA between each top recursive subtree $\mathcal{U}$ and its rightmost bottom recursive subtree $\mathcal{L}_1$ at each level of detail, reducing this above cost by a $\tilde{\Omega}(\sqrt{B})$ factor. Both conference and journal versions [9] ultimately avoid the problem by using indirection. Specifically, a top tree stores only a $\Theta(N/\log^2 N)$ fraction of the elements, so that while modifications of the top tree are expensive, they occur only every $\Omega(\log^2 N)$ updates. Subsequent CO B-trees have avoided arbitrary weight-balanced trees.

Our solution is to use a more flexible PMA [7, 25]. The earliest PMA [9] gives no choice to the user in determining the extent of the rebalance interval; the rebalance intervals are defined by the nodes of an implicit binary tree placed on top of the array. A more sophisticated PMA [7, 25] enables us to choose the neighborhood by *growing left or right arbitrarily* until we find a neighborhood that is within the appropriate density threshold. Then we rebalance this neighborhood. This PMA gives us the flexibility to lay out arbitrary weight-balanced trees. (In contrast, the dynamic vEB layout from [9] only applies to strongly weight-balanced trees (see [9]) and requires indirection for efficiency.)

The main idea of our layout algorithm is that we do not include a node v in a PMA rebalance unless we also include all of v's descendants. This rebalance policy completely fixes the pointer maintenance problems from above. We now give the rebalance policy. Suppose that we insert/delete a leaf in the weight-balanced tree. This leaf is in some lower recursive subtree $\mathcal{L}_i$ with target size 2^{2^0}, which is in the layout $\mathcal{U}\mathcal{L}_1\ldots\mathcal{L}_i\ldots\mathcal{L}_x$. We start with a rebalance interval in the PMA consisting only of $\mathcal{L}_i$. If the density is not within threshold, then we can add $\mathcal{L}_{i-1}$ or $\mathcal{L}_{i+1}$ to the rebalance interval. If the density is still not within threshold, then we add more lower recursive subtrees, and once all lower recursive subtrees $\mathcal{L}_1\ldots\mathcal{L}_i\ldots\mathcal{L}_x$ have been added, we add the upper recursive subtree $\mathcal{U}$ to the rebalance interval. However the rebalance interval may not be within threshold. Let $\mathcal{L}'_j = \mathcal{U}\mathcal{L}_1\ldots\mathcal{L}_i\ldots\mathcal{L}_x$ be the lower recursive subtree with targets size 2^{2^1}, which forms part of the layout $\mathcal{U}'\mathcal{L}'_1\ldots\mathcal{L}'_j\ldots\mathcal{L}'_y$. We repeat the same procedure by growing the rebalance interval starting with $\mathcal{L}'_j$, adding the top recursive subtree $\mathcal{U}'$ last, and proceeding to recursive subtrees with target size 2^{2^2}, 2^{2^3}, 2^{2^4}, etc.

The crucial feature of the vEB layout enabling the dynamic strategy is that all recursive subtrees in a level of detail have asymptotically the same size; see Lemma 3. Thus, we establish the lemma. □

Centroid tree: Modified successor queries

We now show how to implement predecessor/successor queries in the dynamic data structure. Unfortunately the predecessor/successor queries in the static structure do not dynamize easily. The static CO string B-tree maintains a pointer to the largest and smallest descendants of each centroid in the original trie, and an insert of one key means that a large number of nodes in this centroid tree may need updated max/min descendant pointers.

We avoid this problem by changing the specification of the max/min descendant pointers, based on the following structural property of the centroid tree:

LEMMA 10. *Let α be the parent node of γ in the compressed trie. Then, in the centroid tree, either α is a descendant of γ, or γ is a descendant of α.*

Proof. Consider starting at the root of the centroid tree. If neither α or γ is the centroid, then both nodes are in the up-tree or both nodes are in the down-tree, and we recursively consider the new

trie. Otherwise, one node of α and γ is a centroid, and the other node is therefore either in the up-tree or down-tree of that node. □

In the new specification, the predecessor (successor) pointer of a node v points to the lexicographically minimum (maximum) descendant leaf of v (which is a key) in the centroid tree. Now the successor/predecessor pointers may no longer point to the lexicographically minimum/maximum descendants in the subtrie, only in the centroid tree. To understand this distinction consider some subtrie. As we descend in the centroid tree, some of the subsubtries have been matched by down-trees higher up in the centroid tree, and there are $O(\log N)$ such subsubtries. The leftmost descendant of the root of the subtrie is the leftmost descendant of that node in the centroid or the leftmost descendant of one of the ancestors of that node in the centroid tree, since these ancestors represent down-trees that were removed.

We now explain how to answer successor/predecessor queries. In the static case a successor/predecessor query was answered by following the max/min descendant pointers from a single node v. In the dynamic case, we need to look at the max/min pointers for $O(\log N)$ nodes: v and all of v's ancestors in the centroid tree. By Lemma 10, the minimum and maximum descendants of v belongs to this set. We can determine which pointers indicate the minimum and maximum descendant without having to follow the pointers (e.g., by using a tree-labeling scheme and order-maintenance queries [6, 18] to determine which nodes are descendants of v in the trie, and from those that are, looking at the leftmost and rightmost pointing pointers in the PMA). Thus, we answer these queries matching the static performance of Theorem 5.

With this new specification, when we insert a key into the PMA, we only need to update the first and last pointers of $O(\log N)$ centroid nodes. Thus, we achieve following the performance bounds:

LEMMA 11. *We can answer predecessor and successor queries in the same bounds as Theorem 5, where on inserts and deletes of keys only $O(\log N)$ centroid nodes are affected.*

Centroid tree: Rebuilding

When keys are inserted or removed from the centroid tree, parts of the tree may go out of balance and need to be rebuilt. We cannot simply employ an existing cache-oblivious layout strategy for nonuniform trees (e.g., [2, 17, 22]) because we want to rebuild only subtrees, not the entire tree. The centroid tree of Section 3 is always weight balanced to within a factor of two. If we allow the centroid tree to "drift" out of balance (say to within a factor of four), then we have a weight-balanced property and a subtree needs to be rebuilt only if a relatively large number of insertions or deletions have occurred. Here we explain how a given subtree of the centroid tree can be rebuilt without incurring too many block transfers.

First, put all the trie elements, stored in the centroid tree into DFS/Euler-tour order for the trie. This reordering can be done bottom up in $O(\log N)$ scans of the centroid tree, since the tree is only $O(\log N)$ deep. Then given the trie in Euler-tour order, scan the trie to find the centroid, partition the trie into upper and lower trees stored in different parts of memory, and then repeat recursively on each part.

Thus, we have the following performance bounds:

LEMMA 12. *The amortized cost to re-layout a centroid tree is $O(1+(\log^2 N)/B)$ amortized memory transfers. This cost does not include the maintenance of the max/min descendant pointers.*

Centroid tree: Maintaining PMA max/min descendant pointers

We now show how to maintain the max/min descendant pointers from the centroid tree into the PMA when the PMA elements shift around. We show that this update may in fact have an additive cost of $O(\log N)$ memory transfers, meaning that the algorithm for rebalancing centroid trees, as described, has an additional additive cost of $O(\log N)$ memory transfers.

Let $\mathcal{D}$ represent the keys in the down-tree of the root, $\mathcal{U}_L$ represent the keys in the up-tree lexicographically before $\mathcal{D}$, and $\mathcal{U}_R$ represent the keys in the up-tree lexicographically after $\mathcal{D}$. Thus, the lexicographic order is $\mathcal{U}_L\mathcal{D}\mathcal{U}_R$, and the centroid order is $\mathcal{D}\mathcal{U}_L\mathcal{U}_R$. Let $\mathcal{D}$ be further divided into $\mathcal{D}'$, $\mathcal{U}'_L$, and $\mathcal{U}'_R$, let $\mathcal{D}'$ be further divided into $\mathcal{D}''$, $\mathcal{U}''_L$, and $\mathcal{U}''_R$, etc. Thus, the lexicographic order is $\mathcal{U}_L\mathcal{U}'_L\mathcal{U}''_L\mathcal{U}'''_L\mathcal{D}'''\mathcal{U}'''_R\mathcal{U}''_R\mathcal{U}'_R\mathcal{U}_R$, and the centroid order is $\mathcal{D}'''\mathcal{U}'''_L\mathcal{U}'''_R\mathcal{U}''_L\mathcal{U}''_R\mathcal{U}'_L\mathcal{U}'_R\mathcal{U}_L\mathcal{U}_R$. Suppose that $\ldots\mathcal{U}'''_R\mathcal{U}''_R\mathcal{U}'_R\mathcal{U}_R$ each have very few keys in them. Then, an insert in the PMA may move the keys in $\mathcal{U}'''_R\mathcal{U}''_R\mathcal{U}'_R\mathcal{U}_R$, which are in lexicographic order. However, there are pointers to these keys in the centroid tree, and the centroid-tree nodes are stored in centroid order. Thus, a small number of moves in the PMA means that we need to update in what we will show is $O(\log N)$ distinct regions in the centroid tree. Since there is no data locality, this update could use $O(\log N)$ memory transfers.

LEMMA 13. *The amortized cost to update the centroid tree is $O((\log^2 N)/B+\log N)$ amortized memory transfers. The additive $O(\log N)$ memory transfers comes from maintaining the max/min descendant pointers.*

Proof. The centroid tree is stored in memory in centroid order, but we update the pointers of the centroid tree in lexicographic order. We begin by showing that if we scan all centroid-tree nodes in lexicographic order, then the number of memory transfers is $O(N/B+1)$. In a left-to-right scan, we first scan the elements in $\mathcal{U}_L$ (the up-tree whose elements are lexicographically before the centroid), then the elements in $\mathcal{D}$ (the down-tree), and then the elements in $\mathcal{U}_R$ (the up-tree whose elements are lexicographically after the centroid), proceeding recursively within each subtree. In this tree of down-trees, left up-trees, and right up-trees, mark the deepest nodes containing at least B descendants, but where all (three) children contain fewer than B descendants. There are $O(N/B)$ such nodes, and each node causes $O(1)$ memory transfers. To finish counting memory transfers, observe that there are also left or right up-trees with fewer than B descendants that are "aunt/uncle" nodes of marked nodes, i.e., children of ancestors of marked nodes; each of these nodes also causes $O(1)$ memory transfers. However, there are also only $O(N/B)$ of these nodes, because each left (respectively right) up-tree of this form is matched to a sibling right (resp. left) up-tree containing more than B descendants, and there are only $O(N/B)$ such nodes.

We now show how the preceding analysis changes if we do not scan all nodes, only a range of lexicographically contiguous nodes in the centroid tree. Specifically, we may scan the left up-tree without having to scan the sibling down-tree or right up-tree. The same analysis applies if we do not scan all the data, but instead, for every scanned left or right up-tree, we also completely scan the sibling down-tree. We do not retain the same analysis when we scan a left (resp. right) up-tree but not the sibling down-tree or sibling right (resp. left) up-tree. The maximum number of such left or right up-trees that we could scan is $O(\log N)$. For each of these orphan left or right up-trees, we pay an additional $O(1)$ memory transfers, for a total additive cost of $O(\log N)$ memory transfers. □

Hashdata PMA and indirection

The centroid-tree is built on $\Theta(N/B)$ keys, and the hashdata PMA is built upon N keys, storing $O(1)$ information about each key. There are pointers from the leaves of the centroid tree (representing keys) to the elements (again representing keys) in the hashdata PMA, and there are pointers from each element in the hashdata PMA to its associated key in the keydata PMA. There are no back pointers because these would be too expensive to maintain.

The hashdata PMA also stores the fingerprint of the longest common prefix between each key and the previous element, along with the next character in the key. Thus, a search proceeds by finding the predecessor and successor in the centroid tree, searching the hashdata PMA using an additional $O(1+(\log N)/B)$ memory transfers to find the representation of the predecessor and successor in the hashdata PMA, and then jumping into the keydata to return the actual values.

With this extra level of indirection the additive $O(\log N)$ memory transfers in the update cost of the centroid-tree (Lemma 13) is amortized to an $O(1)$ update cost, giving the desired performance bounds.

We now give more details of how the searches work in the hashdata. By searching in the centroid tree, we get pointers to the predecessor and successor in the centroid tree, which gives us a range of $\Theta(\log N)$ possible keys in the hashdata. Note that the predecessor and successor in the centroid tree may be much longer than the search key κ, but we do not need to read these keys unless we actually return them. Instead, we scan from the leftmost key in the range which precedes κ. By comparing the fingerprints and next characters and by scanning κ once, we can determine the predecessor and successor keys.

We thus obtain the following performance bounds:

THEOREM 14. *The dynamic COSB-tree with front compression represents a set $\mathcal{D}$ of N elements, and supports member, predecessor, successor, and range queries. The operation* MEMBER(κ) *runs in $O(1+\log_B N+\|\kappa\|/B)$ memory transfers w.h.p., and* PRED(κ)*, and* SUCC(κ) *run in $O(1+\log_B N+\|\kappa\|/B+\|\kappa'\|/B)$ memory transfers w.h.p., where κ' is the predecessor (resp. successor) of κ. The operation* RANGE-QUERY(κ,κ') *runs in $O(1+\log_B N+(\|\kappa\|+\|\kappa'\|+\langle\!\langle Q\rangle\!\rangle)/B)$ transfers w.h.p.. The compressed keys can be decoded for an additional $\|Q\|/B$ transfers. Finally* INSERT(κ) *and* DELETE(κ) *run in $O(1+\log_B N+\log^2 N\,\|\kappa\|/B)$ memory transfers w.h.p.. All results hold in the cache-oblivious model with the tall-cache assumption.*

As described earlier, we can reduce the bounds by using scanning structures [6], but at the cost of keeping the data out of order and of amortizing scans; details are left for the full version. Finally, if we only need to return pointers to keys, as in the string B-tree, we need not store the keys themselves in a PMA. We call this modification the *pointer COSB-tree* . Our pointer COSB-tree matches the bounds of the string B-tree in the amortized sense while retaining all of the advantages of cache-obliviousness. In summary, we obtain:

THEOREM 15. *The dynamic COSB-tree augmented with a scanning structure achieves the bounds from Theorem 14, except that range queries are amortized, and* INSERT(κ) *and* DELETE(κ) *are accelerated to $O(1+\log_B N+\log^{2+\varepsilon}\log N\,\|\kappa\|/B)$ amortized memory transfers w.h.p., for any $\varepsilon>0$. If range queries only need to return pointers to keys, then the updates become $O(1+\log_B N+\|\kappa\|/B)$ amortized memory transfers w.h.p.. All results hold in the cache-oblivious model with the tall-cache assumption.*

5. REFERENCES

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

[2] S. Alstrup, M. A. Bender, E. D. Demaine, M. Farach-Colton, T. Rauhe, and M. Thorup. Efficient tree layout in a multilevel memory hierarchy. arXiv:cs.DS/0211010, 2004. `http://www.arXiv.org/pdf/cs.DS/0211010`.

[3] A. Amir, M. Farach, and Y. Matias. Efficient randomized dictionary matching algorithms (extended abstract). In *Proc. 3rd Symp. on Combinatorial Pattern Matching (CPM)*, pp. 262–275, Tucson, Arizona, Apr. 1992.

[4] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Inf.*, 1(3):173–189, Feb. 1972.

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

[6] M. A. Bender, R. Cole, E. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proc. 10th European Symp. on Algorithms (ESA)*, pp. 152–164, Rome, Italy, Sept. 2002.

[7] M. A. Bender, R. Cole, E. D. Demaine, and M. Farach-Colton. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In *Proc. 10th Annual European Symp. on Algorithms (ESA)*, pp. 139–151, Rome, Italy, Sept. 2002.

[8] M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. 41st Annual Symp. on Foundations of Computer Science (FOCS)*, pp. 399–409, Redondo Beach, California, 2000.

[9] M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM J. Comput.*, 35(2):341–358, 2005.

[10] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. 13th Annual Symp. on Discrete Mathematics (SODA)*, pp. 29–38, San Francisco, California, Jan. 2002.

[11] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. *J. Algorithms*, 3(2):115–136, 2004.

[12] G. S. Brodal and R. Fagerberg. Cache-oblivious string dictionaries. In *Proc. 17th Annual Symposium on Discrete Algorithm (SODA)*, pp. 581–590, Miami, Florida, Jan. 2006.

[13] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. 13th Annual Symposium on Discrete Algorithms (SODA)*, pp. 39–48, San Francisco, California, Jan. 2002.

[14] H. K. Chang. Compressed indexing method. *IBM Technical Disclosure Bulletin*, 11(11):1400–1401, 1969.

[15] W. A. Clark IV, K. A. Salmond, and T. A. Stafford. Method and means for generating compressed keys. US Patent 3,593,309, 3 Jan. 1969.

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

[17] E. D. Demaine, J. Iacono, and S. Langerman. Worst-case optimal tree layout in a memory hierarchy. Manuscript. arXiv:DS/0410048, 2004. `http://john2.poly.edu/papers/manu04/paper.pdf`.

[18] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th Annual Symposium on Theory of Computing (STOC)*, pp. 365–372, 1987.

[19] P. Ferragina and R. Grossi. The String B-Tree: A new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, Mar. 1999.

[20] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Science*, 47:424–436, 1994.

[21] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual Symp. on Foundations of Computer Science (FOCS)*, pp. 285–297, New York, New York, Oct. 1999.

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

[23] A. Itai, A. G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *Proc. 8th Internationl Colloquium on Automata, Languages, and Programming (ICALP)*, vol. 115, pp. 417–431, Acre (Akko), Israel, July 1981.

[24] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, Mar. 1987.

[25] I. Katriel. Implicit data structures based on local reorganizations. Master's thesis, Technion, Israel Inst. of Tech., Haifa, May 2002.

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

[27] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, pp. 198–199. Springer-Verlag, Berlin, 1984.

[28] M. Naor. String matching with preprocessing of text and pattern. In *Proc. 18th International Colloquium on Automata, Languages and Programming (ICALP)*, pp. 739–750, 1991.

[29] H. Prokop. Cache-oblivious algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Inst. of Tech., June 1999.

[30] Sleepycat Software. The Berkeley Database. `http://www.sleepycat.com`, 2005.

[31] K. A. Smith and M. I. Seltzer. File system aging - increasing the relevance of file system benchmarks. In *Proc. 1997 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pp. 203–213, Seattle, Washington, 1997.

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