Write-Optimized Skip Lists

Michael A. Bender[*]
Stony Brook University

Martín Farach-Colton[†]
Rutgers University

Rob Johnson[*]
Stony Brook University

Simon Mauras[‡]
ENS Lyon

Tyler Mayer[*]
Stony Brook University

Cynthia A. Phillips[§]
Sandia National Laboratories

Helen Xu[¶]
MIT

ABSTRACT

The skip list is an elegant dictionary data structure that is commonly deployed in RAM. A skip list with N elements supports searches, inserts, and deletes in $O(\log N)$ operations with high probability (w.h.p.) and range queries returning K elements in $O(\log N + K)$ operations w.h.p.

A seemingly natural way to generalize the skip list to external memory with block size B is to "promote" with probability $1/B$, rather than $1/2$. However, there are practical and theoretical obstacles to getting the skip list to retain its efficient performance, space bounds, and high-probability guarantees.

We give an external-memory skip list that achieves write-optimized bounds. That is, for $0 < \varepsilon < 1$, range queries take $O(\log_{B^\varepsilon} N + K/B)$ I/Os w.h.p. and insertions and deletions take $O((\log_{B^\varepsilon} N)/B^{1-\varepsilon})$ amortized I/Os w.h.p.

Our write-optimized skip list inherits the virtue of simplicity from RAM skip lists. Moreover, it matches or beats the asymptotic bounds of prior write-optimized data structures such as B^ε trees or LSM trees. These data structures are deployed in high-performance databases and file systems.

The main technical challenge in proving our bounds comes from the fact that there are so few levels in the skip list, an aspect of the data structure that is essential to getting strong external-memory bounds. We use extremal-graph coloring to show that it is possible to decompose paths in the skip list into uncorrelated groups, regardless of the insertion/deletion pattern. Thus, we achieve our bounds by averaging over these uncorrelated paths rather than by averaging over uncorrelated levels, as in the standard skip list.

1. INTRODUCTION

The skip list [39] is an elegant randomized dictionary data structure built from cascading linked lists of geometrically decreasing sizes.

A skip list with N elements supports searches, inserts, and deletes in $O(\log N)$ operations with high probability[1] (w.h.p.) and range queries returning K elements in $O(\log N + K)$ operations w.h.p. [18, 28, 36]. Skip lists have found broad application [3, 5, 6, 19, 21, 25, 26, 35, 43], and they are widely deployed in production [29, 37, 42].

In this paper, we propose a ***write-optimized skip list***. The write-optimized skip list is a randomized ***external-memory*** dictionary that offers asymptotically optimal point-query and insertion performance in the external-memory model while inheriting many of the practical and theoretical advantages of a traditional skip list.

By external-memory, we mean that our data structure resides on a large external storage device, such as a disk or SSD. The external storage device is accessed via I/Os that transfer blocks of size B to a smaller cache (e.g. RAM) of size M.

By write-optimized, we mean that the data structure has asymptotically better insertion performance than a B-tree [7] and query performance at or near that of a B-tree. In practice, the best write-optimized dictionaries match B-trees in terms of query speed while performing insertions and deletions one or two orders-of-magnitude faster. Over the past two decades, researchers have developed write-optimized dictionaries for databases and file systems [4, 9–11, 13–15, 22, 23, 27, 30, 34, 40, 41, 48], several of which have been shown to be asymptotically optimal [14, 48].

Skip list structure. A skip list consists of $h = O(\log N)$ lists $\{\mathcal{L}_0, \mathcal{L}_1, .., \mathcal{L}_h\}$, called ***levels***, where the base level $\mathcal{L}_0$ is a linked list of all items of the set, in sorted order. Each item in level $\mathcal{L}_i$ also appears in (i.e., ***is promoted to***) level $\mathcal{L}_{i+1}$ with probability 1/2. All elements that have been promoted to $\mathcal{L}_{i+1}$ are ***pivots*** with respect to $\mathcal{L}_i$ because they partition $\mathcal{L}_i$ into ranges for searches.

[*] Department of Computer Science, Stony Brook University, Stony Brook, NY 11794-2424 USA. Email: {bender, rob}@cs.stonybrook.edu.
Department of Applied Mathematics and Statistics, Stony Brook University, Stony Brook, NY 11794-3600 USA. Email: tyler.mayer@stonybrook.edu.

[†] Department of Computer Science, Rutgers University, Piscataway, NJ 08854 USA. Email: farach@cs.rutgers.edu.

[‡] École Normale Supérieure de Lyon, Lyon, France. Email: simon.mauras@ens-lyon.fr.

[§] MS 1326, PO Box 5800, Albuquerque, NM 87185 USA. Email: caphill@sandia.gov.

[¶] Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, 02139 USA. Email: hjxu@mit.edu.

PODS'17, May 14-19, 2017, Chicago, IL, USA
© 2017 ACM. ISBN 978-1-4503-4198-1/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3034786.3056117

[1] An event E_n on a problem of size n occurs ***with high probability*** if $\Pr[E_n] \geq 1 - 1/n^c$ for some constant c.

An element promoted to level $\mathcal{L}_{i+1}$ has a pointer to its successor in level $\mathcal{L}_{i+1}$ as well as a pointer to its own occurrence in level $\mathcal{L}_i$ (see Figure 1).

A query for element y begins at the first node on level $\mathcal{L}_h$ and ends on level $\mathcal{L}_0$ at the smallest element no greater than y. At level i, the search performs a sequential scan until it finds the last element, e, that is less than or equal to y in $\mathcal{L}_i$. At that point, the search follows the pointer to e in level $\mathcal{L}_{i-1}$ and resumes its sequential scan from that point.

An insertion of element e first performs coin tosses to compute the height h_e of e. The insertion then searches for e and inserts it into lists $\mathcal{L}_0, \ldots, \mathcal{L}_{h_e}$, with appropriate pointer adjustments.

Inheriting the desirable properties of skip lists. Skip lists have desirable algorithmic properties, which our write-optimized skip list inherits.

For example, it is an advantage to be built from a collection of linked lists. Practitioners generally like to make concurrent lock-free dictionaries as lock-free skip lists [19,24,38,46,47] because it is attractive to build on top of existing, production-quality lock-free linked lists [31,32,45].

Moreover, skip lists are elegant and have an easy-to-understand randomized balancing mechanism. Finally, skip-lists are weight balanced [33] in a probabilistic sense, which makes them useful as an algorithmic tool.

It is these desirable properties that makes us particularly excited to have another optimal write-optimized data structure at our disposal, even though (a few) other optimal structures already exist [9,13–15,48].

See Section 7 for our speculation how a write-optimized skip list may make it easier to implement concurrency and perhaps lock-freedom. Given that the community is only now exploring how to make full-featured, scalable, acid-compliant, write-optimized indexing structures, it is worth having many options in an implementer's arsenal.

Adapting to External Memory

We now articulate the subtleties in adapting skip lists to work in external memory. We review the external-memory model, which is used to analyze disk-resident indexing structures in databases and file systems.

External-memory model. The external-memory or disk-access model (DAM) [2] consists of two levels of memory: a fast memory (RAM) of size M and a slow arbitrarily large external memory, such as a disk. Block transfers, or I/Os, between disk and RAM occur in blocks of size B. Performance is measured in terms of the number of I/Os.

External-memory skip lists. Given the success of the skip list in internal memory, it is natural to extend it to external memory. Indeed, such a data structure exists and is called a B-skip list [1,8, 12,16,17,20].

The straightforward way to extend the skip list to external memory is to promote elements with probability $1/B$ rather than $1/2$. At a given level, each promoted element is stored in a contiguous chunk along with the run of nonpromoted elements that follow it. These chunks define the ***nodes*** of the B-skip-list. Since disk blocks have size B, each node consumes at least B space, regardless of how many elements it contains (see Figure 1).

This B-skip list retains the simplicity of the original RAM skip list but unfortunately has optimal search performance only in expectation, not with high probability [8]. Each node has an expected $\Theta(B)$ elements, but w.h.p. there exist nodes with as many as $\Theta(B \log N)$ elements and nodes with as few as $\Theta(B/\log N)$ elements. Large nodes cause problems because we want searches to take $O(\log_B N)$ I/Os, but performing a linear scan of a node of size $\Theta(B \log N)$ requires $\Theta(\log N)$ I/Os.

We can obtain high-probability bounds on the cost of searches by changing the promotion probability to $1/\sqrt{B}$, rather than $1/B$ [8]. Even with this larger promotion probability, there are only $O(\log_{\sqrt{B}} N) = O(\log_B N)$ levels. Each node now has $\sqrt{B}$ elements in expectation, with the actual number of elements ranging from $\Theta(\sqrt{B}/\log N)$ to $\Theta(\sqrt{B}\log N)$ w.h.p. No matter how big B is relative to $\log N$, this version of the skip list has a search cost of $O(\log_B N)$.

However, now most nodes are mostly empty, so this version wastes space.

Write-Optimized Skip List

Our write-optimized skip list uses the random and variable amount of extra space in each node to store a buffer, similar to a B$^\varepsilon$-tree [10, 14]. By buffering elements within nodes, we can move (or "flush") inserted items down the skip list in batches. This speeds up insertions on average, similar to buffer use in other write-optimized data structures. However, unlike deterministic write-optimized structures, the number of pivots in a node can vary by a factor of as much as $O(\log^2 N)$, which changes the effectiveness of the buffer substantially, and threatens the attainability of optimal high-probability amortized insert bounds.

The main contribution of the paper lies in the analysis. We show that the write-optimized skip list has an asymptotically optimal search-insert tradeoff [14, 48], similar to the B$^\varepsilon$-tree [14, 15, 27], the COLA [9], or the xdict [13]. Our search-insert bounds hold both in expectation and with high probability.

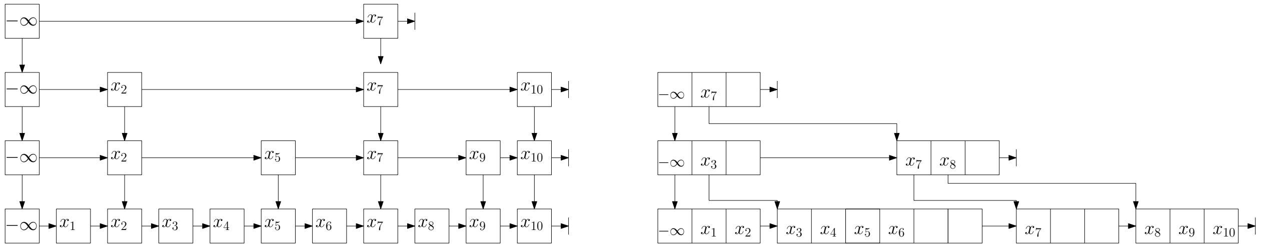
Our write-optimized skip list has an additional technical complication at the leaves to ensure good range-queries and space consumption. We promote with probability $1/B^{1-\varepsilon}$ at the leaf level and $1/B^\varepsilon$ at all other levels. We delay the promotion of elements from buffers at the leaf level as a simple mechanism to guarantee that leaf nodes remain $\Theta(B)$ full.

Challenges in attaining high-probability bounds. A particularly troublous aspect of this data structure is that the ratio of a node's buffer size to number of children can vary by a factor of $\Theta(\log^2 N)$. For example, the root itself might be one of these outlier nodes, an $O(\log N)$ factor larger than average. In that case, the large number of pivots (and low amortized per-child buffer size) would affect all insertions.

In data structures with depth $O(\log N)$ such local variation would even out, both on average and w.h.p. But, our data structure has only $O(\log_B N)$ depth, which is insufficient to overcome unlucky coin tosses. The surprising result is that this buffered skip list meets the desired I/O goals.

To prove high-probability bounds, we need to find, for any workload, sets of $\Omega(\log B^\varepsilon)$ insert paths whose I/O complexity is uncorrelated. This appears to be challenging for some workloads. For example, in a sequential-insert workload, any insert path lies on the rightmost spine of the data structure. Furthermore, since all insertions pass through the top level of the data structure, a large node at the top of the skip list can affect the I/O performance of a substantial fraction of insertions.

Fortunately, we are operating in external memory: we can assume that the top few levels of the data structure are cached. Traversing cached levels incurs no I/Os. We show that the remaining levels of the tree offer enough disjoint root-to-leaf paths so that



(a) RAM skip list.

(b) B-skip list.

Figure 1: An in-memory (RAM) skip list (a) and external-memory B-skip list (b). In the B-skip list, the node size varies by a factor of $O(\log^2 N)$. While the B-skip list achieves asymptotically better bounds than the RAM skip list in expectation, they both achieve the same high-probability bounds [8]. In contrast, the write-optimized skip list has better bounds than the B-skip list both in expectation and w.h.p.

we can prove the desired bounds for write optimization. Indeed, even if all insert paths seem to follow the same root-to-leaf path (e.g., the rightmost spine), the insert path changes structure sufficiently frequently that we can find disjoint root-to-leaf paths.

This proof assumes an optimal paging algorithm. However, our performance bounds still hold in systems that use LRU, since LRU with constant resource augmentation is constant competitive with the optimal paging algorithm.

For ease of presentation, we first give a proof of high-probability bounds that applies when there are insertions, but no deletions. Our proof relies on a coloring argument of the insert paths.

Deletions destroy this first proof: paths that are independent at some point can be moved together by deletions of intervening elements so that they become correlated. We show, via an extremal graph-coloring argument, that there is always a good partitioning of the paths into uncorrelated sets, no matter what the deletion pattern is. This allows us to prove high-probability bounds under any mix of insertions and deletions.

Results

We prove the following theorem establishing the performance of write-optimized skip lists.

THEOREM 1. *Consider an N-element write-optimized skip list running in external memory. Let memory-hierarchy parameters B and M obey the "tall-cache" assumption that $M = \Omega(B^2 \log^4 B)$. Let the block size B be large enough that $\min\{B^\varepsilon, B^{1-\varepsilon}\} \geq \log N$.*

A write-optimized skip list that performs insertions, deletions, and queries achieves the following I/O bounds for tunable performance parameter $0 < \varepsilon < 1$:

- *Insertions and deletions take $O((\log_{B^\varepsilon} N)/(B^{1-\varepsilon}))$ amortized I/Os in expectation and w.h.p.*
- *Range queries returning K elements take $O(\log_{B^\varepsilon} N + K/B)$ I/Os in expectation and w.h.p. (Point queries are range queries with $K = 1$.)*
- *The structure takes $O(N)$ space, in expectation and w.h.p.*

The write-optimized skip list's guarantees (like those of a regular skip list) are based on an oblivious adversary. The oblivious adversary can issue arbitrary insert and delete operations, but does not have access to the heights of the elements in the structure (i.e., the random tosses).

Organization. In Section 2, we explain how to build and use the write-optimized skip list. In Section 3 we prove several structural properties of the write-optimized skip list. In Section 4 we prove performance bounds for point queries and range queries whp. We also prove bounds on insertion and deletion in expectation. In Section 5 we prove amortized insertions bounds w.h.p. and in Section 6 we adjust the argument to include deletions w.h.p. We conclude with some extensions and implementation issues in Section 7.

2. STRUCTURE AND OPERATIONS OF A WRITE-OPTIMIZED SKIP LIST

In this section we explain how to build the write-optimized skip list. This section also sets up notation that will be used throughout the rest of the paper.

Overall Structure. The write-optimized skip list has pointer structure similar to that of the B-skip list [8, 20]. It is composed of a sequence of hierarchical ***levels*** $\mathcal{L}_0, \mathcal{L}_1, \ldots, \mathcal{L}_h$, where h is the height of the data structure. We will show $h = O(\log_{B^\varepsilon} N)$ w.h.p.

Each level consists of a linked list of ***nodes*** (which will have size $\Theta(B)$ w.h.p.), where each node is partially filled with ***pivots***. Nodes at level 0 are ***leaves***. Each pivot element e on level $\mathcal{L}_{i\geq 1}$ has a pointer to the ***child*** node containing its occurrence on level $\mathcal{L}_{i-1}$. (We will see that sometimes there may temporarily be no node that contains e on level $\mathcal{L}_{i-1}$; in this case, the pointer points to the node that would contain e based on the sort order.) The smallest pivot in a node is called its ***leader***. Each node at level i contains a pointer to the next node at that level (see Figure 2).

Write-optimized skip list nodes are similar to nodes in a B$^\varepsilon$-tree [10, 14] in that each node also contains a ***buffer***. All the elements in a node's buffer are greater than or equal to the node's leader and smaller than the leader of the next node on that level. Inserted items are stored in nodes' buffers and are ***flushed*** in batches from parents to children. Thus, all elements move towards $\mathcal{L}_0$, where they remain (until they are deleted).

Randomized balancing. Each element e in the data structure has an integer ***height*** h_e determined by a sequence of biased coin flips. Coin flips are implemented by hashing e, meaning that even if an element is inserted, deleted, and reinserted, h_e does not change. To determine h_e, flip a biased coin until the first tail and set h_e to the length of the run of heads. For the first flip, the probability of heads is $1/B^{1-\varepsilon}$, and on subsequent flips the probability of heads is $1/B^\varepsilon$. We say that an element e has been ***promoted*** to level $i > 0$ if $h_e \geq i$.

The promotion probabilities are set such that each node on levels $\mathcal{L}_{i\geq 1}$ has $\Theta(B^\varepsilon)$ pivot elements in expectation and each node on level $\mathcal{L}_0$ has $\Theta(B^{1-\varepsilon})$ elements in expectation. The variable (random) amount of extra space in each node serves as the buffer space in our insertion algorithm and enables us to achieve amortized high-probability write-optimized update bounds, as discussed in Section 5 and Section 6.

As with a regular skip list, to ensure that there is a root for the entire structure, there is a special element $-\infty$ that is defined to have the largest height of any element.

Insertions and deletions. When a new element is inserted, store it qin the root's buffer. When an element e is deleted, store a ***tombstone*** $\bar{e}$ in the root's buffer.

Buffer-flushing mechanism. When the buffer in node D at level $\mathcal{L}_{i\geq 1}$ becomes full (i.e., it ***overflows***), perform a ***flush*** operation. During a flush, distribute the elements in D's buffer among the buffers of D's children. This may require an I/O per child to bring the children nodes into main memory.

Whenever any one child has $B^{1-\varepsilon}$ delete messages destined for it, flush those delete messages to the appropriate child immediately. (This extra rule for flushing deletes helps us achieve the desired range-query bounds; see Theorem 9).

Pivots and leaders. When an element e gets flushed out of the buffer of a node D of height $i \leq h_e$, e becomes a pivot of D in addition to being flushed to the buffer of one of D's children. This new pivot will point to the node to which e is being flushed. This means that D now has two (or more) pivots that point to the same child.

If $i < h_e$, then split D into two nodes D' and D'', making the current leader of D the leader of D' and e the leader of D''. Since D may have multiple pivots pointing to the same child, splitting D may result in some of D's children having more than one parent.

Whenever a node D that has multiple parents is split, update all of D's parents to point to the newly created nodes. Splitting a node D will not change the size of any of D's parents so that, unlike a B-tree, splitting can proceed in a purely top-to-bottom fashion. This is because, whenever a node D is split to create a new node D'' with leader e, element e must already be a pivot in D's parents.

When a delete message $\bar{e}$ is flushed from a node, delete e as a pivot of that node, if it happens to be one. If e is also the leader of that node, then merge that node with its predecessor on that level. Thus, merges are the reverse of splits.

Leaves require special handling. Whenever there is a flush from a parent D at level 1 to all of its leaves, rebalance all the leaves as follows: greedily choose the breaks between leaves so that each leaf approximately fills a block and each leaf begins with a pivot of D (but not every pivot of D begins a leaf).

Queries. To search for element e, traverse the root-to-leaf path to e, and retain all these nodes in memory until the query is done. Our assumptions on the size of memory imply that $M > B \log_{B^\varepsilon} N$, so that a complete root-to-leaf path fits in memory.

The leaf may or may not contain e itself. Insertions and deletions of e may reside in buffers on the root-to-leaf path. Find the messages in the highest buffer that affects e: if it is an insert, then e is present. If it is a delete, then e is absent. The I/O complexity is $O(\log_{B^\varepsilon} N)$ w.h.p.

Each buffer could be checked on the way down, until the first message that affects e is found. But the above method generalizes to richer queries. Consider finding the successor of e. First, find the successor of e in every root-to-leaf buffer and return the min-value of these that is currently in the dictionary. A trivial way to do this is to sort all the messages in all the buffers under consideration by (f, i, t), where f is the key, i is the height, and t is the type (insertion or deletion), then to remove all but the first occurrence of each key. This yields the current state of each key. Finally, search for e's successor by finding the smallest $f > e$ and then scanning to the first insertion.

There is one missing detail. If e is the largest element in its leaf and is larger than everything in the root-to-leaf buffers, then the successor of e will reside in the root-to-leaf path of the successor leaf. This does not increase the I/O complexity of successor, which is $O(\log_{B^\varepsilon} N)$.

A range query is implemented by repeated successor queries. Once the beginning of the range is found, successive leaves contain $\Theta(B)$ values in the range, and the I/Os for fetching internal nodes is dominated by that of fetching leaves. Thus, a K-element range query takes $O(K/B + \log_{B^\varepsilon} N)$ I/Os.

Top-down splits and merges: another advantage of write-optimized skip lists. Splits, merges, and promotions are performed in a *top-to-bottom* fashion. As we describe briefly in Section 7, this artifact of using a randomized rebalancing scheme may, in fact, turn out to be another hidden advantage of the write-optimized skip list over other data structures.

In particular, it may make it easier to implement concurrent write-optimized skip lists. There may be advantages both for lock-based implementations as well as well as lock-free versions. See Section 7 for details.

3. STRUCTURAL BOUNDS

In this section we establish structural properties of the write-optimized skip list, establishing both expected and high-probability bounds.

We assume throughout that $\min\{B^\varepsilon, B^{1-\varepsilon}\} \geq \log N$.

3.1 Local Structure

LEMMA 2 (PIVOTS IN AN INTERNAL NODE). *An internal node has B^ε pivots in expectation. and $O(B^\varepsilon \log N) = O(B)$ pivots w.h.p.*

PROOF. By construction, we begin a new internal node when we see a promotion to the next level. Therefore, the number of pivots in each internal node can be modeled as the number X of tails before the first heads in a sequence of independent coin flips with a head probability of $B^{-\varepsilon}$. The expectation of X is B^ε. The high probability bounds follow from the Chernoff bounds. □

The following lemma implies that accessing any node requires $O(1)$ I/Os w.h.p.

LEMMA 3 (NODE SIZE). *For $0 < \varepsilon < 1$, a node is comprised of $O(1)$ blocks w.h.p.*

PROOF. For levels greater than 0, nodes contain pivots and $\Theta(B)$ buffer space. By Lemma 2, nodes have $O(B)$ pivots w.h.p., so the total size of an internal node is $O(1)$ disk blocks.

By the same argument, even though the promotion probability at the leaves is $1/B^{1-\varepsilon}$, every run of $\Theta(B)$ elements at the leaf level has a promoted element w.h.p. Thus, when packing elements at the leaf level into blocks, we can create a new leaf every $\Theta(B)$ blocks w.h.p. Hence, every node at level 0 consumes $O(1)$ blocks w.h.p. □

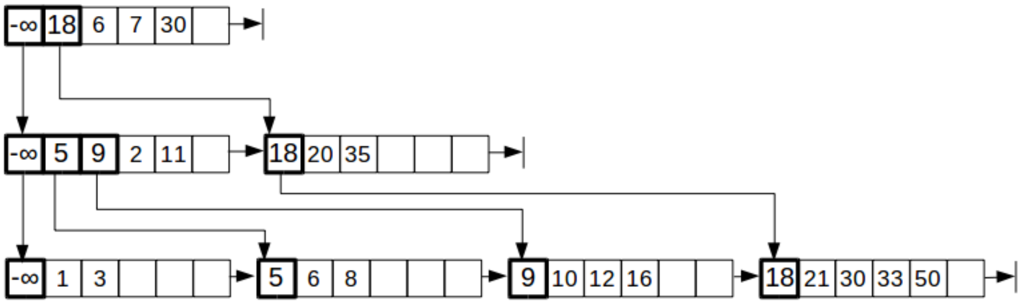


Figure 2: Structure of a write-optimized B-skip list with block size $B = 6$. We illustrate the pointer structure of the skip list as well as the pivot and buffer structure of nodes. Each node has size $O(B)$ w.h.p. Any extra space in the nodes is used as buffer space. The number of children at any (internal nonroot) node varies by an $O(\log^2 N)$ factor, meaning that the contribution to the amortized I/O cost for insertions and deletions from that node also varies by an $O(\log^2 N)$ factor. This large variation is an obstacle for designing external-memory skip lists with high-probability performance bounds.

The following lemma bounds the number of neighbors—parents, children, successors and predecessors—of a node. This will help us bound the cost of performing flushes, since flushes may have to access all of a node's neighbors.

LEMMA 4 (NEIGHBOR BOUNDS). *Let D be a node at height at least 1. The number of parents of D is $O(1)$ w.h.p. The expected number of children of D is $O(B^\varepsilon)$. If the height of D is exactly 1, then D has $O(\log N) = O(B^\varepsilon)$ children w.h.p.*

PROOF. The bound on children breaks into two cases:

- If D is at level $i > 1$ then, by Lemma 2, its expected number of pivots is $O(B^\varepsilon)$, and therefore so is the expected number of children.
- Nodes at level 1 are split whenever an element is promoted to level 2. Each element in level 0 has a $1/B$ chance of being promoted to level 2. By Chernoff bounds, any run of $\Omega(B \log N)$ elements at level 0 has at least 1 element promoted to level 2 w.h.p. Thus, w.h.p. no node at level 1 has more than $O(B \log N)$ elements in its children. Since each child has $\Theta(B)$ elements, nodes at level 1 have $O(\log N) = O(B^\varepsilon)$ children w.h.p.

The number of parents of D is at most the number of messages in D's buffer that have height at least 2 larger than the height of D. Since D has height at least 1, the probability that any particular item in D's buffer has height 2 greater than the height of D is at most $1/B^{1+\varepsilon}$. Since D's buffer contains $O(B)$ items, the expected number of such elements in D's buffer is $O(1/B^\varepsilon)$. Thus, by the Chernoff bounds, the number of such elements is $O((\log N)/B^\varepsilon)$ w.h.p. Since $\log N < B^\varepsilon$, the number of such elements, and hence the number of parents of D, is $O(1)$. □

3.2 Global Structure

THEOREM 5 (LINEAR SPACE). *A write-optimized skip list on N elements uses $O(N/B)$ blocks in expectation and w.h.p.*

PROOF. Each leaf holds $\Theta(B)$ items by construction and from Lemma 3 consumes $O(1)$ blocks w.h.p. Thus, the total space consumed by leaves is $O(N/B)$ w.h.p.

The number of blocks at $\mathcal{L}_1$ is also $O(N/B)$ since it is not more than the number of leaves.

For levels 2 and above, the space consumption follows the same analysis as the B-skip list. □

The following two lemmas help us bound the I/O costs of queries and inserts.

LEMMA 6 (HEIGHT UPPER BOUND). *For constant $0 < \varepsilon < 1$, the height of the write-optimized skip list is $O(\log_{B^\varepsilon} N)$ both in expectation and w.h.p.*

PROOF. The probability that any given element has height at least $h \geq 1$ is $1/B^{1-\varepsilon+(h-1)\varepsilon}$.

Let $c \geq 2$ be a constant. The probability that a given element has height at least $h = 1 + c\log_{B^\varepsilon} N$ is at most

$$\frac{1}{B^{1+(h-2)\varepsilon}} \leq \frac{1}{B^{\varepsilon(h-1)}} \leq \frac{1}{B^{\varepsilon c \log_{B^\varepsilon} N}}.$$

The probability that any given element has height at least $1 + c\log_{B^\varepsilon} N$ is at most $1/N^c$. By the union bound, the probability that any of the N elements has height at least $1 + c\log_{B^\varepsilon} N$, is at most $1/N^{c-1}$. □

LEMMA 7 (PIVOTS ON A SEARCH PATH). *The total number of pivots at level 2 or higher touched by any root-to-leaf search path in the data structure is $O\left(B^\varepsilon \log N\right)$ w.h.p.*

PROOF. Consider the search path "backwards." That is, start from the element x_i in the leaf level, and consider the unique trajectory from x_i back to the root following pointers backwards. The search path is comprised of some number of horizontal pointers (point to pivots on the same level) and $O(\log_{B^\varepsilon} N)$ vertical pointers (from Lemma 6).

We can model the length of this search path mathematically as the number of coin flips until $O(\log_{B^\varepsilon} N)$ heads have been seen with high probability. At levels 1 and above, the probability of a head is $1/B^\varepsilon$. Using Chernoff bounds, one can prove that $O(B^\varepsilon \log N)$ coin flips are enough to go back from level 1 to the root w.h.p. □

4. SIMPLE RUNTIME BOUNDS

This section gives high-probability bounds on the query performance. It also gives expected bounds on the amortized cost of insertion and deletion.

The amortization in the insertion bound is similar to the analysis of flushes in a B$^\varepsilon$-tree [14]. One interesting difference is that, with B$^\varepsilon$-trees, one must analyze the cost of splitting separately from the cost of flushes, since splitting is a non-local operation. In the write-optimized skip list, on the other hand, splitting and merging is performed locally as part of flushing, so we can bound its cost as part of the analysis of the flushing cost.

4.1 Query Performance

Next, we show bounds for point queries with constant tunable performance parameter ε.

THEOREM 8 (POINT QUERIES). *A point query has a worst case I/O complexity $O(\log_{B^\varepsilon} N)$ w.h.p.*

PROOF. From Lemma 7, each search path contains $O(B^\varepsilon \log N)$ elements w.h.p. Furthermore, the height of the tree is $O(\log_{B^\varepsilon} N)$ w.h.p. (from Lemma 6). For any search path, we must pay at most a single random I/O each time we descend a level. However, elements of the same level are stored contiguously in blocks (nodes), therefore we can make a linear scan over a level reading $O(B)$ elements per I/O.

Thus, the cost to read all elements in a particular search path is $O(\log_{B^\varepsilon} N + (B^\varepsilon \log N)/B) = O(\log_{B^\varepsilon} N)$ w.h.p. □

THEOREM 9 (RANGE QUERIES). *The I/O complexity of range queries is $O(\log_{B^\varepsilon} N + K/B)$ w.h.p. where K is the number of elements in the requested interval.*

PROOF. The cost for range queries can be analyzed using the search paths of the left and right ends of the requested interval. The complexity of a range query is bounded by the number of leaf nodes holding the elements in the range plus $O(\log_{B^\varepsilon} N)$ (the cost of a point query w.h.p.). Between the two search paths is a small write-optimized skip list of the K items returned by the range query. By Theorem 5, the total space consumed by the nodes in this mini skip list is $O(K/B)$ w.h.p. □

4.2 Insert and Delete Bounds in Expectation

THEOREM 10 (WRITE-OPTIMIZED UPDATES). *For $0 < \varepsilon < 1$, the amortized cost of inserting or deleting an element in the data structure is $O((\log_{B^\varepsilon} N)/B^{1-\varepsilon})$ in expectation.*

PROOF. We first analyze the expected cost of a flush. A flush of a node D must access all the children and parents of D, in addition to writing any new nodes that result from splitting or merging D. By Lemma 4, there are $O(B^\varepsilon)$ parents and children in expectation. If we do a merge, we may have to access D's predecessor and its parents, but this is $O(1)$ additional nodes in expectation. Thus, the total number of nodes accessed during a flush is $O(B^\varepsilon)$ in expectation. From Lemma 3, each node fits in $O(1)$ blocks w.h.p., so the total number of I/Os required by a flush is $O(B^\varepsilon)$ in expectation.

We now analyze the expected amortized insertion/deletion cost. By Lemma 6, Each element (or tombstone) must be flushed $O(\log_B N)$ times w.h.p. Thus, the total number of element flushes we must perform during any sequence of N insertions and deletions is $O(N \log_B N)$ with high probability. Each flush performs $\Theta(B)$ element-flushes with high probability. Thus, the total number of flushes performed is $O((N \log_B N)/B)$ with high probability. Since each flush costs $O(B^\varepsilon)$ I/Os in expectation, the amortized insertion cost is $O((\log_B N)/B^{1-\varepsilon})$ I/Os in expectation. □

5. HIGH PROBABILITY INSERTION-ONLY BOUNDS

This section establishes expected and high-probability bounds on the amortized insertion cost for a write optimized skip list that only handles insertions. We prove these bounds for a skip list that also handles deletes in Section 6.

Unfortunately, Theorem 10 does not obviously generalize to give matching high probability bounds on the amortized insertion cost. This is because, although there are many node flushes, many are not independent, preventing us from applying Chernoff bounds. We may flush a node many times before it gets split or merged with one of its siblings.

To overcome this problem, we partition the elements inserted into the skip list into ***color classes***, where all the elements of the same color follow (mostly) disjoint flushing paths. As a result, all the flushes (and flushing costs) involving these elements are independent. As long as the number of elements in a color class is large enough, we can use Chernoff bounds on the total cost of all the flushes of all the elements in that class.

The main challenge is that we are not guaranteed enough disjoint paths near the root of the skip list.

We use caching to address this problem. Flushes between nodes in cache incur no I/O, and hence can be ignored. As long as enough levels at the top of the skip list are cached, we can find large classes of elements that all follow disjoint paths through the uncached portion of the skip list.

5.1 Caching Assumptions and Structural Bounds

Caching Assumptions. Our high-probability bounds assume that the top $\Omega(1)$ levels of this data structure (those closest to the root) are permanently pinned in cache. An optimal cache-replacement policy will outperform these results, but an optimal policy requires prescience, rendering it unimplementable. However, the LRU (least-recently used) cache-replacement strategy is a 2-approximation to optimal, given a cache of twice the size [44], implying that our bounds still hold asymptotically with LRU. We account for the doubled memory in the asymptotics of our tall cache assumption.

For the analysis of the skip list assuming only insertions, we need the cache size to be $\Omega(B^2 \log^2 B)$; for the analysis with insertions and deletions, $\Omega(B^2 \log^4 B)$. Therefore, we generalize the analysis to a cache of size $M = \Omega(B^2 X)$.

Structural Properties. We now establish preliminary lemmas about the ***cached region*** (i.e., levels stored in cache).

We first give a lower bound on the number of levels that can be cached and the size of the largest cached level.

LEMMA 11 (HEIGHT OF CACHED REGION). *Suppose that internal memory has size $M = \Omega(XB^2)$ and let h' be the height of the lowest level with $O(XB^\varepsilon \log N)$ nodes. Then every node in level at least h' fits in memory w.h.p.*

PROOF. The number of nodes at level h' is $O(XB^\varepsilon \log N)$. Each node requires $\Theta(B)$ space w.h.p. The amount of space for all nodes at height h' is order the following:

$$BXB^\varepsilon \log N \leq XB^{1+\varepsilon} B^{1-\varepsilon} = XB^2.$$

Thus, the size needed to store nodes of level h' is $O(XB^2)$ w.h.p. By Chernoff bounds, the number of nodes in higher levels decreases exponentially. Once the expected number of elements at a level is at most $\log N$, w.h.p, that level consumes at most one block. So the total number of nodes in all levels at or above h' is $O(XB^\varepsilon \log N + \log_{B^\varepsilon} N)$. Given the tall cache assumption that $M = \Omega(XB^2)$, there is sufficient space to store all levels with height at least h'. □

LEMMA 12 (PIVOTS IN LAST LEVEL CACHED). *With a cache of size $\Omega(B^2 X)$, the lowest level that fits in cache has $\omega(X \log N)$ pivots with separate children w.h.p.*

PROOF. Let h be the height of the highest level that does not fit into internal memory. (Since $M < N$, h exists.) Then, from Lemma 11, the number of nodes at that level is $\omega(XB^{\varepsilon} \log N)$. This means that the number of pivots at level $h+1$ that have separate children is $\omega(XB^{\varepsilon} \log N)$. □

The following theorem will help us argue the existence of independent paths through the disk-resident region.

THEOREM 13 (CACHED ELEMENT FREQUENCY). *If $M = \Omega(XB^2)$, then, in any set of $\Omega(N/X)$ contiguous distinct elements at least one element is promoted into a cached level w.h.p.*

PROOF. Let p be the probability that an element has been promoted to the cached region. Using Lemma 12, $p \geq (cX \log N)/N$ with high probability for all constants $c > 0$.

Now, let q be the probability that no element is promoted to the cached region in a group of $\Omega(N/X)$ elements.

$$\begin{aligned} q = (1-p)^{\Omega(N/X)} &= \exp\left(\Omega\left(\frac{N}{X}\right)\log(1-p)\right) \\ &\leq \exp\left(-\Omega\left(\frac{pN}{X}\right)\right) \\ &\leq \exp\left(-\Omega\left(\frac{cNX\log N}{NX}\right)\right) \\ &= \frac{1}{\Omega(N^c)}. \end{aligned}$$

Thus, in any such such group of $\Omega(N/X)$ elements, w.h.p. at least one element is promoted to a cached level. □

5.2 Element Coloring Algorithm and Analysis

We use the aforementioned bounds on the size of the cached region and the frequency of cached elements to present an element coloring algorithm for insertion analysis.

We describe why normal Chernoff-bound analysis is insufficient and then use a coloring argument on disjoint root-to-leaf paths in a skip list with a large enough cache to establish the amortized cost of insert operations w.h.p.

The obstacle to using Chernoff bounds as above is that insertions that pass through the same node of the skip list will have correlated flushing costs. However, flushes between nodes in cache require no I/Os. Thus, if enough levels at the top of the tree are cached, then many insertions will follow independent paths through the uncached levels of the skip list enabling us to use Chernoff bounds to bound their overall cost.

The ***rank*** of an element is its position in the sorted list of all elements in the data structure regardless of whether or not it has reached the leaves. That is, the ith smallest element in the data structure has rank i, even if it is still making its way through the internal nodes due to the buffer-flushing scheme described earlier.

The insertion paths of two elements a, b are ***independent*** if they are node-disjoint in the part of the skip list that is not cached in memory. The following lemma proves the existence of a coloring of elements into such independence classes.

LEMMA 14. *There exists a coloring of elements inserted in the data structure such that elements in the same color class experience disjoint root-to-leaf paths w.h.p.*

PROOF. The following algorithm colors elements such that after every operation, the difference in rank between any two elements of the same color is at least $N/2X$.

Coloring Algorithm:

- Insert the first N/X elements with distinct colors, establishing the set of colors C.
- When a new element e is inserted at rank k, let C_I be the set of colors of the elements at ranks $[k - N/2X, k + N/2X]$. Assign e the color in $C \setminus C_I$ that currently has the fewest elements.

If the difference in rank between two elements a and b (where $a < b$) is $\Omega(N/X)$, then there exists at least one element (greater than a and at most b) between them promoted into the cached region w.h.p. (from Theorem 13). This element will be a leader at every level not cached in internal memory. The presence of such an element is enough to isolate the insertion paths of a and b.

Later insertions in the data structure do not affect this property, because the difference in rank between two elements can only increase over time. □

Now we prove, using the above coloring scheme, that the expected amortized cost of insertions holds w.h.p. We use the union bound on the amortized insertion costs for each color class as described in Theorem 15.

THEOREM 15. *The amortized insertion cost for $\Omega(\log B)$ elements with independent insertion paths in the data structure is $O((\log_{B^{\varepsilon}} N)/B^{1-\varepsilon})$ w.h.p.*

PROOF. We show that the amortized cost of flushes is $O((\log_{B^{\varepsilon}} N)/B^{1-\varepsilon})$ w.h.p. Recall from Lemma 4 that the amortized cost of flushes from level 1 to level 0 is $O(1/B^{1-\varepsilon})$ I/Os w.h.p. Thus, we need to bound the amortized cost of flushing elements only to levels 1 and above.

As an element moves down the $\log_{B^{\varepsilon}} N$ levels, there is a flush at each level. Each flush moves $\Theta(B)$ elements down one level. The total number of I/Os for all these flushes is the total number of pivots on the path, since each pivot has a child at the next lower level which could receive elements in a flush. Lemma 7 shows that the total number of pivots on levels 1 and above along any root-to-leaf search path of length $\log_{B^{\varepsilon}} N$ is $O(B^{\varepsilon} \log N)$ w.h.p., not matching the expected bounds.

For the amortized analysis to hold w.h.p., we need $\Theta(B^{\varepsilon} \log_{B^{\varepsilon}} N)$ pivots at levels two and above per path when averaged over all paths. At a high level, when we only consider one path there are not enough trials for us to avoid paying an additional asymptotic cost.

By identifying $\Omega(\log B)$ disjoint paths through the above coloring scheme, we can "concatenate" them. We model the total number of pivots at level 2 and above along this grouped search path mathematically as the number of coin flips needed until $\Omega(\log N)$ heads have been seen with high probability.

By Chernoff bounds, we need $O(B^{\varepsilon} \log N)$ coin flips in total for $\Omega(\log B)$ heads. Therefore, the amortized number of pivots at level 2 or above per path is $O((B^{\varepsilon} \log N)/\log B) = O(B^{\varepsilon} \log_{B^{\varepsilon}} N)$ w.h.p.

Since we transfer B elements with each I/O, the amortized cost of inserting an element along each of these disjoint paths is $O((B^{\varepsilon} \log_{B^{\varepsilon}} N)/B) = O((\log_{B^{\varepsilon}} N)/B^{1-\varepsilon})$. □

Recall our "tall cache" assumption that the size of memory $M = \Omega(XB^2)$ for some memory parameter X. We now show that $X = \Omega(\log^2 B)$ is sufficient to achieve the desired write-optimized bounds in an insert-only data structure.

THEOREM 16 (WRITE-OPTIMIZED INSERTIONS). *If the size of memory $M = \Omega(XB^2) = \Omega(B^2 \log^2 B)$ in an insert-only skip list, the insertion cost per element is $O((\log_{B^\varepsilon} N)/B^{1-\varepsilon})$ w.h.p.*

PROOF. After an insertion sequence, the elements have been divided into color classes. Consider the following two cases for the color classes:

Case 1: A color class has at least $\log B$ elements in it. We can apply the Chernoff bound analysis for concatenated paths and obtain $O((\log_{B^\varepsilon} N)/B^{1-\varepsilon})$ amortized insert cost per element w.h.p.

Case 2: A color class has fewer than $\log B$ elements in it. If $X = \Omega(\log^2 B)$, there are $O(N/X) = O(N/\log^2 B)$ color classes by construction. Therefore, there are at most $O(N/\log^2 B)$ "bad" color classes for which we cannot apply Chernoff bounds to acquire the same asymptotic bound w.h.p. as in Theorem 15. Furthermore, there are strictly fewer than $\log B$ elements in each of these "bad" classes. Therefore, there are at most $O(N/\log B)$ "bad" elements. For a "bad" element we may have to pay the naïve (single path analysis) cost of $(\log N)/B^{1-\varepsilon}$ (w.h.p.). However, the total cost amortized for these elements is at most $O((N \log_{B^\varepsilon} N)/B^{1-\varepsilon})$ w.h.p.

Finally, we calculate the amortized insertion cost per element over N inserts. That is,

$$\left(O(N)\frac{\log_{B^\varepsilon} N}{B^{1-\varepsilon}} + O\left(\frac{N}{\log B}\right)\frac{\log N}{B^{1-\varepsilon}}\right)/N$$
$$= O\left(\frac{\log_{B^\varepsilon} N}{B^{1-\varepsilon}}\right).$$

□

6. HIGH PROBABILITY BOUNDS WITH INSERTIONS AND DELETIONS

The previously described coloring scheme allows us to build groups of $\Omega(\log B)$ independent paths. However, we cannot perform delete operations, because the proofs do not allow the difference in rank between elements to decrease.

As the coloring algorithm is only a theoretical tool, we can assume that the adversary has no knowledge of the chosen colors. Equivalently, we can assume that we know all the requests from the beginning creating an "offline" coloring problem. We show that this allows us to extend the w.h.p. update bounds to include deletes.

We begin by describing a modified coloring scheme based on building a conflict graph of keys less than X apart in rank. Next, we show that this analysis technique allows us to prove the desired write-optimized bounds for both insertions and deletions.

Specifically, we describe a scheme to color $\Theta(N)$ updates on a skip list of size N. We introduce an undirected graph $G = (V, E)$, with the set of vertices V being the keys in the skip list. An edge $\{u, v\}$ is added to E if and only if at some point the difference in rank between keys u and v is smaller than N/X.

- When inserting an element at rank k, we add at most $2N/X$ edges to the graph, binding all the keys for which the difference in rank with k is smaller than N/X not to be of the same color.

- When removing an element at rank k, we add at most N/X edges between the key of rank $k + i - N/X - 1$ and key of rank $k + i$ for $0 \le i \le N/X$.

The total number of edges is therefore $O(N^2/X)$. We will use this information with Lemma 17.

Recall from Theorem 13 that if $M = \Omega(B^2X)$, then at least one element is promoted into the cached region in a block of N/X elements w.h.p.

Therefore, two elements with the same color will have at least one "splitting element" between them that causes them to have disjoint paths outside of the cached region.

At a high level, assume that we have a write-optimized skiplist with some N active elements. In the very beginning (startup stage), allow the data structure to fill to some chosen constant size $C_{\min}$—all operations in this stage therefore have constant cost. We set this as our first N in the following analysis.

We build a conflict graph for some sequence of N' operations such that even if all N' operations are deletes, we still have $\Theta(N)$ active elements in the data structure for our amortized analysis. Thus, N' is some constant fraction of N, e.g., $N/4$. We also stop the sequence if the size of the data structure falls below $C_{\min}$.

Coloring the conflict graph. The analysis of the update cost based on the conflict graph is done in stages, or **epochs**, based on the length of the sequence of operations. At the beginning of a sequence we have N active elements. After N' operations, we have some N_0 active elements such that $N_0 = \Theta(N)$. At the end of this epoch, we set N_0 as the new N and repeat.

LEMMA 17. *Given an undirected graph $G = (V, E)$, let $\chi(G)$ be the smallest number of colors needed to color the vertices (chromatic number). Then $\chi(G) = O(\sqrt{|E|})$.*

PROOF. Assume that we have fewer than $\binom{\chi(G)}{2}$ edges. There must be two colors that can be merged together, contradicting the fact that $\chi(G)$ is optimal. Therefore, $\chi(G) = O(\sqrt{|E|})$. □

We can therefore color the keys using $O(N/\sqrt{X})$ colors.

Analysis using color classes. The analysis is similar to the proof of Theorem 16. We use the grouping technique again with the coloring of the conflict graph and bound the number of color classes with fewer than $\log B$ elements.

If a color class has at least $\log B$ elements, we can do the analysis using Chernoff bounds within this color.

We can bound the number of elements for which the "grouped" analysis is not feasible by $O((N \log B)/\sqrt{X})$. For those elements, we can apply Chernoff bounds naïvely for a total cost of $O\left(\frac{N \log B}{\sqrt{X}} \cdot \frac{\log N}{B^{1-\varepsilon}}\right)$.

Since $X = \Omega(\log^4 B)$, the amortized complexity for the "bad" elements is $O((\log_{B^\varepsilon} N)/B^{1-\varepsilon})$.

At the end of each epoch, start a new conflict graph from scratch. That is, start with a vertex for each of the N'' keys currently in the data structure and add edges between each pair of keys whose ranks differ by at most N/X. Now repeat the analysis process with $N = N''$. We thus obtain the following:

THEOREM 18 (WRITE-OPTIMIZED INSERTS/DELETES). *If the size of memory $M = \Omega(B^2 \log^4 B)$ in a skip list with inserts and deletes, then with high probability, the amortized insertion and deletion cost per element is $O((\log_{B^\varepsilon} N)/B^{1-\varepsilon})$ w.h.p.*

7. CONCLUSION

The write-optimized skip list achieves the asymptotically optimal I/O bounds of the best write-optimized data structure while retaining the elegance and simplicity of skip lists. The high probability bounds are established via extremal graph-coloring arguments based on the elements' root-to-leaf paths through the data structure.

We are hopeful that the skip list's randomized rebalancing will have practical (as well as theoretical) impact in the burgeoning area of write-optimization (e.g., as a basis for full-featured production data structures). We briefly try to articulate why we are so hopeful.

Regular skip lists have formed the basis for concurrent and lock-free production dictionaries. Part of the reason why is that they have simpler implementations because they are built out of regular linked lists.

We believe that write-optimized skip lists will benefit from these advantages as well. Consider, for example, the node-splitting mechanisms in write-optimized skip lists versus a B-tree or B$^\varepsilon$-tree. In a write-optimized skip list, node splits and merges are triggered "on the way down," i.e., as insert and delete messages make their way deeper into the structure.

In contrast, in B-trees and B$^\varepsilon$-trees, splits are triggered "on the way up," once inserts and deletes have reached the leaves. In a concurrent data structure based on locks, it is important to grab locks according to a prespecified partial order in order to avoid deadlock. Naïve "hand-over-hand" locking (grab and release locks as you walk down the tree to avoid throttling concurrency) is insufficient to design concurrent B-trees or B$^\varepsilon$-trees. If an insert reaches a leaf and triggers some splits higher up in the tree, the data structure no longer has the necessary locks higher up in the tree. Industrial B-trees and B$^\varepsilon$-trees generally deal with this concurrency issue by implementing delayed splitting mechanisms.

For example, when we built TokuDB, we built a mechanism for delaying splits, letting node sizes grow, and letting future inserts or deletes take care of the splits. Clearly, this locking issue is solvable, but the coding seems simpler in a write-optimized skip list.

Similarly, skip lists have been the data structure of choice for theoretically good, in-production lock-free dictionaries. This is not only because they are built out of separate linked lists, but also because of other structural properties (such as level pointers along the entire level). Perhaps write-optimized skip lists will become the easiest-to-implement lock-free write-optimized data structures.

In future research, we will perform an implementation study to explore whether the theoretical advantages revealed in this paper can lead to benefits for implementers and users.

Acknowledgments

This work is supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

This research was also supported by NSF grants CCF 1617618, IIS 1247726, IIS 1251137, CNS 1408695, and CCF 1439084, CNS 1408782, IIS 1247750, by NIH grant CA198952-01, and by EMC, Inc and NetApp, Inc.

We thank Jon Berry of Sandia National Laboratories for helpful comments.

8. REFERENCES

[1] I. Abraham, J. Aspnes, and J. Yuan. Skip B-trees. In *Proc. 9th Annual International Conference on Principles of Distributed Systems (OPODIS)*, page 366, 2006.

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

[3] A. Anagnostopoulos, M. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. *Information Security*, pages 379–393, 2001.

[4] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[5] L. Arge, D. Eppstein, and M. T. Goodrich. Skip-webs: efficient distributed data structures for multi-dimensional data sets. In *Proc. 24th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 69–76, 2005.

[6] J. Aspnes and G. Shah. Skip graphs. *ACM Transactions on Algorithms*, 3(4):37, 2007.

[7] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[8] M. A. Bender, J. W. Berry, R. Johnson, T. M. Kroeger, S. McCauley, C. A. Phillips, B. Simon, S. Singh, and D. Zage. Anti-persistence on persistent storage: History-independent sparse tables and dictionaries. In *Proc. 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)*, pages 289–302, 2016.

[9] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *Proc. 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 81–92, 2007.

[10] M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan, and Y. Zhan. An introduction to B$^\varepsilon$-trees and write-optimization. *:login; magazine*, 40(5):22–28, October 2015.

[11] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. *Proc. of the VLDB Endowment (PVLDB)*, 5(11):1627–1637, 2012.

[12] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proc. 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 228–237, 2005.

[13] G. S. Brodal, E. D. Demaine, J. T. Fineman, J. Iacono, S. Langerman, and J. I. Munro. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1448–1456, 2010.

[14] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 546–554, 2003.

[15] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 859–860, 2000.

[16] P. Callahan, M. T. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Proc. 4th International Workshop on Algorithms and Data Structures (WADS)*, pages 381–392, 1995.

[17] V. Ciriani, P. Ferragina, F. Luccio, and S. Muthukrishnan. Static optimality theorem for external memory string access. In *Proc. 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 219–227, 2002.

[18] L. Devroye. A limit theory for random skip lists. *The Annals of Applied Probability*, pages 597–609, 1992.

[19] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proc. 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 50–59, 2004.

[20] D. Golovin. The B-skip-list: A simpler uniquely represented alternative to B-trees. *arXiv preprint arXiv:1005.0662*, 2010.

[21] M. T. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. *US Patent App*, 10(416,015), 2000.

[22] G. Graefe. Write-optimized B-trees. In *Proc. 30th International Conference on Very Large Data Bases (VLDB)*, pages 672–683, 2004.

[23] G. Graefe. B-tree indexes for high update rates. *SIGMOD Rec.*, 35(1):39–44, Mar. 2006.

[24] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A provably correct scalable concurrent skip list. In *Proc. Conference On Principles of Distributed Systems (OPODIS)*, 2006.

[25] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. *Proc. 14th Annual Colloquium on Structural Information and Communication Complexity (SIROCCO)*, page 124, 2007.

[26] R. Jacob, A. Richa, C. Scheideler, S. Schmid, and H. Täubig. A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In *Proc. 28th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 131–140, 2009.

[27] C. Jermaine, A. Datta, and E. Omiecinski. A novel index supporting high volume data warehouse insertion. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *Proc. 25th International Conference on Very Large Data Bases (VLDB)*, pages 235–246, 1999.

[28] P. Kirschenhofer and H. Prodinger. The path length of random skip lists. *Acta Informatica*, 31(8):775–792, 1994.

[29] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2010.

[30] Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *Proc. 2009 IEEE International Conference on Data Engineering (ICDE)*, pages 1303–1306, 2009.

[31] P. A. Martin. Method and apparatus for implementing a lock-free skip list that supports concurrent accesses, Dec. 11 2007. US Patent 7,308,448.

[32] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proc. 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 73–82, 2002.

[33] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973.

[34] P. O'Neil, E. Cheng, D. Gawlic, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[35] R. Oshman and N. Shavit. The SkipTrie: low-depth concurrent search without rebalancing. In *Proc. 32nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 23–32, 2013.

[36] T. Papadakis, J. I. Munro, and P. V. Poblete. Analysis of the expected search cost in skip lists. In *Proc. 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 160–172, 1990.

[37] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. ConcurrentSkipList. In *Java concurrency in practice*. Pearson Education, 2006.

[38] W. Pugh. Concurrent maintenance of lists. Technical report, Dept. of Computer Science, University of Maryland, College Park, 1990.

[39] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[40] R. Sears, M. Callaghan, and E. A. Brewer. Rose: compressed, log-structured replication. *Proc. of the VLDB Endowment*, 1(1):526–537, 2008.

[41] R. Sears and R. Ramakrishnan. bLSM: a general purpose log structured merge tree. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 217–228, 2012.

[42] N. Shamgunov. The MemSQL in-memory database system. In *Proc. 2nd International Workshop on In Memory Data Management and Analytics (IMDM)*, 2014.

[43] N. Shavit and I. Lotan. Skiplist-based concurrent priority queues. In *Proc. 14th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 263–268, 2000.

[44] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, Feb. 1985.

[45] G. L. Steele Jr, A. T. Garthwaite, P. A. Martin, N. N. Shavit, M. S. Moir, and D. L. Detlefs. Lock-free implementation of concurrent shared object with dynamic node allocation and distinguishing pointer value, Nov. 30 2004. US Patent 6,826,757.

[46] H. Sundell and P. Tsigas. Scalable and lock-free concurrent dictionaries. In *Proc. 2004 ACM Symposium on Applied Computing (SAC)*, pages 1438–1445, 2004.

[47] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 214–222, 1995.

[48] K. Yi. Dynamic indexability and the optimality of b-trees. *Journal of the ACM*, 59(4):21:1–21:19, Aug. 2012.