Flushing Without Cascades

Michael A. Bender[1] Rathish Das[1] Martín Farach-Colton[2] Rob Johnson[3]
William Kuszmaul[4]

Abstract

Buffer-and-flush is a technique for transforming standard external-memory search trees into write-optimized search trees. In exchange for faster amortized insertions, buffer-and-flush can sometimes significantly increase the latency of operations by causing cascades of flushes. In this paper, we show that flushing cascades are not a fundamental consequence of the buffer-flushing technique, and can be removed entirely using randomization techniques.

The underlying implementation of buffer flushing relies on a buffer-eviction strategy at each node in the tree. The ability for the user to select the buffer eviction strategy based on the workload has been shown to be important for performance, both in theory and in practice.

In order to support arbitrary buffer-eviction strategies, we introduce the notion of a *universal flush*, which uses a universal eviction policy that can simulate any other eviction policy. This abstracts away the underlying eviction strategy, even allowing for workload-specific strategies that change dynamically.

Our deamortization preserves the amortized throughput of the underlying flushing strategy on all workloads. In particular, with our deamortization and a node cache of size poly-logarithmic in the number of insertions performed on the tree, the amortized insertion cost matches the lower bound of Brodal and Fagerberg. For typical parameters, the lower bound is less than 1 I/O per insertion. For such parameters, our worst-case insertion cost is $O(1)$ I/Os.

1 Introduction

Storage systems—including file systems [20, 29, 30, 39, 44, 47] as well as SQL and NoSQL databases [4, 15, 25, 28, 34, 40, 45, 46]—have been transformed over the last decade by new high-performance external-memory data structures. These ***write-optimized dictionaries (WODs)*** include log-structured merge trees (LSMs) [8, 38, 43], B$^\varepsilon$-trees [9, 14], write-optimized skip lists [10], COLAs [8], and xDicts [13].

The primary performance advantage of WODs over older data structures, such as B-trees [6,16], is their insertion throughput. Some WODs also support asymptotically optimal point queries, thus matching a B-tree's optimal point-query performance while far exceeding a B-tree's insert performance. WODs also support deletions via delete messages, which are special values indicating that the associated key has been deleted.

Despite the profusion of WODs [8–10, 13, 14, 23, 26, 27, 38, 43], most share the same overall organization. These structures are partitioned into sorted levels that grow exponentially in size. New key-value pairs are inserted into the top level. As new elements arrive, elements are ***flushed*** (moved down) from one level to the next. ***Upserts*** (inserts, deletes, and updates) are fast because each I/O simultaneously flushes many elements down one level, leading to a small amortized I/O cost per upsert. Indeed, in the common case when one I/O flushes at least a superlogarithmic number of elements, the amortized I/O cost per upsert is $o(1)$. A key search now needs to look in multiple locations but this need not affect its asymptotic I/O cost.

WODs vary primarily in two ways: their indexing structures and, more significantly, their flushing policies. The choice of indexing structure can have an effect on the query performance, but maintaining them is typically a lower-order term compared with the cost of upserts; see Section 6 for details. Flushing policies, on the other hand, can have a major impact on upsert performance.

Practitioners have explored a large number of flushing policies in order to optimize for different performance criteria. For example, the LSM community has developed a dizzying array of so-called "compaction strategies," which govern how the data structure chooses which runs of elements get merged together. Example compaction strategies include leveled [21, 23, 27], leveled-N [21], tiered [21], tiered+leveled [21], size-tiered [23, 26], FIFO [21], time-windowed [42], eager [12], adaptive [12], lazy [17], and fluid [17]. These policies make various trade-offs between the I/O and CPU complexity of upserts and queries, the

[1]Stony Brook University, Stony Brook, NY, USA. Email: {bender,radas}@cs.stonybrook.edu.

[2]Rutgers University, New Brunswick, NJ, USA. Email: martin@farach-colton.com.

[3]VMWare Research, Palo Alto, CA, USA. Email: robj@vmware.com.

[4]MIT, Cambridge, MA, USA. Email: kuszmaul@mit.edu.

This work was supported by a Fannie & John Hertz Foundation Fellowship, a NSF GRFP Fellowship, and NSF grants: CNS 1408695, CNS 1755615, CCF 1439084, CCF 1725543, CSR 1763680, CCF 1716252, CCF 1617618, CCF 1533644, CSR 1938180, CCF 1715777, CCF 1712716.

and by Sandia National Laboratories.

amount of space used on disk, the level of concurrency supported by the key-value store, the amount of cache, and many other concerns. Furthermore, all of these compaction strategies can be combined with partitioned compaction [36], which attempts to partially deamortize compaction (albeit without guarantees) and to make the compactions adaptive to the upsert workload. B$^\varepsilon$-trees can also use a variety of different flushing policies, including flush-all, greedy [7], round-robin, random-ball [7], and random.

Most WODs have no guarantees on insertion latency. A major drawback of WODs is that they generally lack latency guarantees. Considerable effort has been devoted to reducing the latency of WODs in practice through partitioning [36] and by performing flushing on background threads [21], but less so in theory [8]. Indeed most of the authors of this paper have substantial engineering experience in reducing WODs latencies in the field. For example, Tokutek pushed to reduce the latency of TokuDB [45] and to get rid of periods of lower throughput [31]. But TokuDB does not have provable latency guarantees.

Latency is a paramount concern but, as the examples above illustrate, latency reduction must contend with other design goals.[1] Consequently, it is not enough to deamortize a particular flushing rule that is being used in a particular version of a particular system.

This paper: flushing with strong latency guarantees. We present a randomized construction that enables a wide class of flushing rules to be deamortized. Our deamortization preserves the instance-specific I/O cost of the underlying buffer flushing strategy, i.e. if the original strategy incurs C I/Os on some workload W at a node x, then our deamortized version will incur $O(C)$ I/Os on the same workload. For concreteness, we describe our results in terms of B$^\varepsilon$-trees, and in Section 6 we explain how our results apply to other WODs.

1.1 Flushing in B$^\varepsilon$-trees

We review flushing in B$^\varepsilon$-trees and why they provide almost no guarantee on insertion latency.

B$^\varepsilon$-trees. The B$^\varepsilon$-tree [9, 14] is like a B-tree in that it has nodes of size B, where B is the I/O (i.e., block-transfer) size. Leaves all have the same depth and key-value pairs are stored in the leaves. Balance is maintained via splits and merges, similar to a B-tree. Like a B-tree, B$^\varepsilon$-trees support inserts, deletes, point queries, and successor/predecessor queries.

B$^\varepsilon$-trees have a fanout of $\Theta(B^\varepsilon)$, where constant ε ($0 < \varepsilon < 1$) is a tunable parameter. Thus, a tree with N key-value pairs will consist of $n = \Theta(N/B)$ nodes and have height $O(\frac{1}{\varepsilon}\log_B N)$, which is $O(\log_B N)$, since $\varepsilon = \Theta(1)$. Only $O(B^\varepsilon)$ space in an internal node is needed for pivots, so the rest of the space is used for a ***buffer***. Buffers batch up insertion and deletion messages as they work their way down the tree. Searches are as in a B-tree, except that we need to check buffers along the way for any pending insertion or deletion messages. Since each node, including its buffer, is stored in a single block, searches cost $O(\log_B N)$ I/Os. Successor and predecessor queries must search in all buffers along the current root-to-leaf path to find the next/previous key. Performing a sequence of k successor or predecessor queries costs $O(k/B + \log_B N)$ I/Os.

New messages are queued in the buffer at the root of the tree. In a standard (amortized) B$^\varepsilon$-tree, whenever a buffer ***overflows*** (contains B or more messages), we perform a ***buffer flush***, i.e., we move some number of messages from the node's buffer to the buffers of the node's children. When a delete message reaches a leaf, all older key-value pairs for that key are deleted from the leaf, and the delete message is discarded. When an insertion message reaches a leaf, it replaces any existing key-value pair for that key.

The main design question is: how many messages should be moved in a flush, and to which children? The simplest policy is to flush everything. With this policy, $\Omega(B)$ messages get moved one level down the tree in a single flush. At most B^ε children need to be accessed, and each access costs $O(1)$ I/Os. Thus, the amortized cost to move one message down one level is $O(1/B^{1-\varepsilon})$. Since the height of the tree is $O(\log_B N)$, the amortized message insertion cost is $O(\frac{\log_B N}{B^{1-\varepsilon}})$ I/Os, which is the best possible on a random insertion workload [14].

Many flushing strategies. There are several flushing strategies that can improve upon the performance of the simple strategy given above without sacrificing its good amortized performance guarantees. For non-random (e.g., skewed) insertion workloads, partial flushes can give substantial speedups.

The speedup of a B$^\varepsilon$-tree is directly proportional to the average number of messages moved per I/O in a flush. So the goal is to push as many messages down per I/O as possible.

For example, the ***greedy policy*** flushes messages *only* to the single child to which the most buffered messages are directed, and obtains the same bounds as above. Greedy is never worse than the flush-everything policy and, for a purely random workload, will improve throughput by a factor of 2 on average [7]. However, there are some workloads where greedy will do no better than the flush-everything policy, but a "random ball" [7] policy achieves an amortized insertion cost of $O(\frac{\log_B N}{B})$ I/Os, which is asymptotically faster.

High latency and flushing cascades. In the context of B$^\varepsilon$-trees, we can see why WODs generally have large latencies

[1] For example, the deamortized COLA [8] offers good (although suboptimal) worst-case latency, but the flushing policy sacrifices throughput for the sake of latency. Specifically, the performance of a sequential-insertion workload gets throttled back to that of a random-insertion workload. This is inconsistent with a system that optimizes for common-case workloads.

for upsert workloads—upserts can trigger flushing cascades. A ***flushing cascade*** occurs when a flush at one node triggers flushes in multiple children nodes; these flushes may, in turn, trigger flushes in the grandchildren, etc.

Structural changes to the B$^\varepsilon$-tree are another major source of flushing cascades. Regardless of the eviction strategy, when two nodes are merged into a new node, the resulting buffer can overflow, triggering new flushes, which again cause cascades (and more node merges). The result is that a single insert or delete could trigger modifications to $\Omega(n^{1-o(1)})$ nodes of the tree.

One could attempt to mitigate cascades by always flushing exactly $B^{1-\varepsilon}$ elements from parent to child, but this discards the potential performance gains of sophisticated flushing policies, such as greedy or random ball. Furthermore, as explained above, it doesn't even work when the workload contains deletions and the structure of the tree changes.

Informally, there appears to be a tension between throughput and latency. Whenever a flushing policy succeeds in moving a large number of elements to a child, that child becomes more likely to need to flush to multiple grandchildren, causing a cascade.

Results. We show that there is no inherent trade-off between flushing policy throughput and latency. We give a randomized algorithm that eliminates flushing cascades with high probability, for *any* buffer-eviction policy.

To understand why this is inherently an algorithmic issue, note that there is a small amount of slack in when we are forced to perform a buffer flush. In particular, we are free to defer some flushes, letting some nodes grow up to a constant factor larger than B and flushing others a little early, without asymptotically harming insertion or query performance. We could even allow for very small (only constant-sized) cascades.

We model the problem of deamortizing an arbitrary flushing policy as a game between a deamortizer and a buffer-eviction policy. In each round, the deamortizer picks which node to flush. Then the ***buffer-eviction policy*** chooses which and how many messages to flush from that node. By fully deamortizing an adversarial flushing strategy, we deamortize them all. This is important because a flushing strategy may be benevolent in terms of write performance but adversarial in terms of latency.

We obtain the following performance bounds. Let N_t be the size of the data structure, in terms of messages, after the t-th operation, and $N_{\max} = \max_t N_t$. Then the t-th insert/delete costs $O(\lceil (\log_B N_t)/B^{1-\varepsilon} \rceil)$ I/Os with high probability in $N_{\max}$, which is $O(1)$ for typical values of B and N. This improvement to worst-case insertion performance does not come at a cost in amortized insertion performance: The data structure deterministically (i.e. with probability 1) achieves the same query and amortized insertion performance as a B^ε-tree.

Our results support the use of an arbitrary message-eviction strategy in each node. In order to establish this, we introduce the notion of a ***universal flush***, a buffer-eviction strategy that we prove can simulate any other eviction strategy. Universal flushes provide a layer of abstraction between the data structure and the underlying eviction strategy.

However, with essentially any flushing strategy (greedy, flush all, universal), the constant amount of slack in each node is nowhere close to enough for deterministic bounds. For that matter, neither is randomization alone. In the rest of this paper, we show that by randomizing when flushes happen, caching a small number of nodes, randomizing the rebalances slightly, and performing an asymptotically negligible number of "cleanup" I/Os, we can obtain our optimality bounds.

Paper outline. In Section 2, we present an overview of the main technical ideas in the paper. In Section 3.1, we define universal flushes, and use them to simulate arbitrary buffer-eviction policies. In Sections 3 and 4, we present technical preliminaries and present a simplified version of the data structure that supports only ***static trees***, in which the only write operations are updates. Finally, in Section 5, we present the full version of the data structure, which supports arbitrary insert/delete/update operations.

2 Technical Overview

We now give an overview of some of the main technical ideas in the paper. Let N denote the size of the tree. (When being formal in later sections, we instead use $N_{\max}$, the *maximum* size of the tree over all time.) Assume throughout that the block-size B is at least $c \log N$ for a constant c of our choice, that the internal memory M is of size at least $\log^c N$ blocks, and that the internal memory satisfies the tall-cache assumption, meaning that M is at least cB^2.

As a convention, we say a node x is in the ***first level*** of a B$^\varepsilon$-tree T if x is a leaf, and otherwise we say that x is at level $\ell + 1$, where ℓ is the level of x's children. This convention of indexing levels from the bottom of the tree (rather than the top) is particularly natural for B$^\varepsilon$-trees, since structurally new levels form when the node at the *top* of the tree splits.

Simulating eviction strategies with universal flushing. In Section 3.1, we present the definition of a universal flush, and show how to use universal flushes in order to simulate arbitrary buffer-eviction strategies.

A ***universal flush*** on a node x can be performed whenever the node's buffer contains $3B$ or more messages. The universal flush is required to select *exactly* B messages to evict from that node. We show that, for any buffer eviction strategy $\mathcal{A}$, one can construct an implementation $\mathcal{B}$ of universal flushing that simulates $\mathcal{A}$ efficiently. In particular, the number of I/Os expended by $\mathcal{B}$ to manage a stream of mes-

sages S to a buffer of size $3B$, is, in the worst case, at most twice the number of I/Os that would expended by $\mathcal{A}$ to manage the same stream S of messages to a buffer of size B.

Since each universal flush at the root of the tree evicts B elements, we can always spread the I/Os for each universal flush across the next B insert/update/delete operations. The I/Os associated with a single universal flush at the root (and the subsequent flushes it induces lower in the tree) are referred to as a ***flushing round***. In general, our goal is to implement flushing rounds using at most $O(\log_B N)$ universal flushes. Since each universal flush requires at most $O(B^\varepsilon)$ I/Os, and since each universal flush is spread across B insert/update/delete operations, this results in a worst-case-operation-running-time of $O(\lceil B^\varepsilon \cdot \log_B N/B \rceil) = O(\lceil \log_B N/B^{1-\varepsilon} \rceil)$, as desired.

Eliminating flushing cascades in a static tree. We begin by focusing on the simpler problem of eliminating flushing cascades from a tree whose node-structure is static. One can think of such a tree as supporting updates only, and no insertions or deletions.

The first algorithm we consider is called the ***aggressive randomized flushing algorithm***, which works as follows. Whenever a new node x is created, a ***random threshold*** r_x is selected uniformly at random from $\{0, 1, \ldots, B-1\}$, and r_x ***dummy messages*** are placed in x's buffer. The algorithm then follows the simple rule of performing a randomized flush on each node x whenever the size of x's buffer is $3B$ or larger.

The purpose of the r_x dummy messages is to randomize the number of messages modulo B in each node x's buffer, also known as the ***B-signature*** of node x. Because universal flushes always remove *exactly* B messages from x's buffer, they do not affect the B-signature. In particular, at any given point in time, the B-signature is given by $r_x + u_x \bmod B$, where u_x is number of messages to have ever been flushed into node x. Since r_x and u_x are independent, and since r_x is uniformly random in $\{0, 1, \ldots, B-1\}$, it follows that, at any given point in time, the B-signature at each node x is uniformly random in $\{0, 1, \ldots, B-1\}$.

The randomness of the B-signatures of the nodes x at each level ℓ ensures that, whenever $j \leq B$ messages are flushed into node x, there is probability exactly $\frac{j}{B}$ that the size of x's buffer crosses a multiple of B. More generally, suppose that during a sequence of k flushing rounds, a total of j_ℓ messages are flushed into level ℓ from level $\ell+1$. We show that the total number of flushes performed at level ℓ will then be a sum of independent indicator random variables with total mean j_ℓ/B. If j_ℓ is sufficiently large in $\text{polylog}(N) \cdot B$, then with high probability in N, the number of messages $j_{\ell-1}$ flushed from level ℓ to level $\ell-1$ during the k flushing rounds will be at most $(1+\frac{1}{\log_B N})j_\ell$. By applying this fact to every level ℓ, we see that during any sequence of k flushing rounds with k sufficiently large in $\text{polylog}(N)$, the total number of flushes performed at ℓ levels below the root is at most $(1+\frac{1}{\log N})^\ell k \leq O(k)$ for all $\ell \in [O(\log N)]$.

The aggressive randomized flushing algorithm achieves low latency for batches of $B\text{polylog}(N)$ consecutive operations, but fails to give a worst-case bound for any individual operation. To achieve a worst-case bound, we take motivation from a recent algorithm for what is known as the ***single-processor cup game*** [5, 11, 18, 19, 35]. At the beginning of the single-processor cup game, n cups are initially empty. At each step of the cup game, a ***filler*** distributes 1 new unit of water among the N cups, and then an ***emptier*** selects one cup out of which to remove $1+\varepsilon$ units of water for some resource-augmentation parameter ε. The emptier's goal is to minimize the ***backlog*** of the system, which is defined to be the amount of water in the fullest cup. It has been known for decades that the greedy-emptying algorithm (of emptying from the fullest cup) achieves backlog $O(\log N)$. In recent work [11], we showed that if the filler is an oblivious adversary (rather than an adaptive one), then a randomized emptying strategy can do significantly better. If $\varepsilon \geq \frac{1}{\text{polylog}(N)}$, then the randomized algorithm achieves backlog $O(\log \log N)$ with high probability in N. The algorithm works by first inserting a random amount of water r_x between 0 and $1+\varepsilon$ into each cup at the beginning of the game, and then, after each step, removing $1+\varepsilon$ units of water from the fullest cup; if, however, the fullest cup contains less than $1+\varepsilon$ units of water, then no water is removed from the cup. (This is so that the amount of water modulo $1+\varepsilon$ in each cup x remains uniformly random as a function of r_x.)

To a first approximation, one can treat each level ℓ of a B^ε tree as its own cup game in which each buffer represents a cup. This suggests the following algorithm, which we call the ***lazy randomized flushing algorithm***: As before, whenever a node is created, we insert a random number $r_x \in \{0, 1, \ldots, B-1\}$ of dummy messages into that node's buffer. Then, during each flushing round, we perform *at most one flush* at each level ℓ, prioritizing the fullest buffer at the level (and flushing from the buffer only if it contains $3B$ or more messages).

Although the lazy randomized flushing algorithm attempts to treat each level ℓ in the tree as its own cup game, the cup game is missing a key ingredient: resource augmentation. In order to create the $(1+\varepsilon)$ resource augmentation needed, we modify the B^ε-tree so that each level ℓ uses a slightly larger block size B_ℓ than does its parent level $\ell+1$. In particular, we set $B_1 = \Theta(B)$, and $B_{\ell+1} = B_\ell - \lceil B/\ell^2 \rceil$. This simultaneously guarantees that $B_\ell = \Theta(B)$ for all $\ell \leq B$, while also guaranteeing that $B_\ell \geq B_{\ell+1} \cdot (1 + 1/\log^2 N)$ for all ℓ. The increase in block-size by a factor of $1 + 1/\text{polylog}(N)$ simulates resource augmentation between the corresponding cup games.

Concurrent work to ours presents a randomized algorithm for the single-processor cup game that achieves bounds

on backlog without the use of resource augmentation [33]. As we shall see, however, the resource augmentation used by the lazy flushing algorithm also grants the algorithm additional behavioral properties that will be useful (and, in fact, necessary) in designing our final flushing algorithm.

Whereas the aggressive randomized flushing algorithm deterministically bounds the size of each buffer to $O(B)$ (but may sometimes perform many universal flushes), the lazy algorithm deterministically bounds the number of universal flushes (but may sometimes allow buffers to grow large). Even allowing for buffers to have size as large as $B \log\log N$ can have disastrous performance consequences in the corresponding B^ε-tree level, however. In particular, lookups may in the worst-case require $\log\log N \cdot \log_B N$ I/Os, which if $B = \text{polylog}n$ would result in worst-case read performance of $\Omega(\log N)$. Thus even a buffer size of $\Theta(B \log\log N)$ is unacceptable.

To handle the large buffers caused by the lazy algorithm, we exploit a second property of the lazy randomized flushing algorithm. Although the algorithm allows for buffers to become overfilled (containing a maximum of $O(B \log\log N)$ messages), the total number of overfilled buffers at any given moment is at most $\text{polylog}(N)$ with high probability in N. In fact, a stronger property is also true with high probability: the sum of the sizes of the overfilled buffers is at most $\text{polylog}(N) \cdot B$. This means that we can maintain an in-memory cache of size $\text{polylog}(N) \cdot B$ storing all the contents of any overfilled buffers. By referencing the in-memory cache, operations that access an overfilled buffer need not pay more than $O(1)$ I/Os in order to perform the access.

Building upon the ideas outlined above, Section 4 designs a flushing strategy for the static B^ε-tree that completely eliminates flushing cascades. The key challenge becomes to adapt this strategy to dynamic B^ε trees. This requires us to not only handle flushing cascades due to structural changes, but also to modify the lazy randomized flushing strategy in order to handle difficulties from nodes being merged and split.

We remark that the use of cup-emptying-game techniques in this paper differs significantly from past applications of cup-emptying games to deamortization [2, 3, 18, 19, 22, 24, 32, 37]. Whereas past applications have relied only on backlog guarantees, our application instead relies primarily on what one might normally view as a secondary property of the random-thresholds-based emptying algorithm (specifically, the fact that the algorithm bounds the sum of the sizes of cups that contain more than 1 unit of water). Applying this property to other problems in external-memory data structures and write optimization is therefore an interesting possible direction for future work.

Eliminating cyclic dependencies in dynamic trees. In order for each level of the B^ε-tree to be treated as a separate randomized "cup game", it is necessary that the input flushes for a level ℓ (i.e., the flushes performed at level $\ell + 1$) are completely independent of the random bits used to determine the flushes at level ℓ. That is, the flushes performed at level $\ell + 1$ must be oblivious to the flushes performed at lower levels.

Node merges and splits violate this obliviousness property by introducing cyclic dependencies between levels. The random bits used at level ℓ partially determine which messages make it into lower levels by a given point in time, and thus which messages make it to the leaves by a given point in time. The arrival of messages to the leaves of the B^ε-tree determines when (and which) nodes at level 2 of the tree merge and split. The merging and splitting of nodes at level 2 in the tree then, in turn, influences when nodes at level 3 of the tree merge and split, etc. Thus the random bits at level ℓ indirectly influence the set of nodes at level $\ell + 1$, which in turn affects the flushing decisions made in level $\ell + 1$.

Eliminating the cyclic dependencies requires a node-rebalancing mechanism in which the nodes at a given level ℓ of the tree are a function *only* of randomness and flushing decisions made at levels ℓ or higher in the tree. Even when splitting a node x into two new nodes x_1 and x_2, the mechanism must not examine node x's children in determining the pivot at which the split should occur, since doing so would introduce a cyclic dependency between levels ℓ and $\ell - 1$.

We introduce a node-rebalancing mechanism in which each node x at level ℓ maintains a ***summary-sketch*** $\mathcal{S}_\ell$ of the contents of the tree T_x rooted at x. The summary sketch simultaneously maintains a size estimate for T_x, and a median estimate, taking the form of a key $u \in T_x$ that is guaranteed (with high probability) to have rank between $|T_x|/4$ and $3|T_x|/4$ in tree T_x. Node merges and splits are performed based on the size estimate, ensuring that each node x at level ℓ always has tree-size $|T_x|$ within a constant factor of $B^{1-\varepsilon} \cdot B^{\varepsilon\ell}$; when a node splits, the median estimate u is used as the pivot at which the split occurs. Although node-rebalancing decisions for node x are made independently of x's child set, the weight-based approach to node-rebalancing guarantees that every node has exactly $\Theta(B^\varepsilon)$ children.

In addition to performing accurate size and median estimation, the summary sketches $\mathcal{S}_\ell(x)$ at each level ℓ must each fit within a single disk block, and be easily composable (meaning, for example, that $\mathcal{S}_\ell(x \cup y)$ can be constructed from $\mathcal{S}_\ell(x)$ and $\mathcal{S}_\ell(y)$).

A natural approach to constructing each summary sketch $\mathcal{S}_\ell(x)$ would be to randomly sample the elements of T_x using a hash function, and then to perform size estimation based on the number of sampled elements (and median estimation using the median of the sampled elements). Such an approach can be easily made successful assuming access to a family of hash functions $\mathcal{H}$ with a high degree of independence (i.e., N-wise independence). The description bits for

such a family of hash functions would be far too large to fit in memory, however. To solve this, we instead use a collection of B low-independence hash functions, $h_1, \ldots, h_B$, each of which performs size and median estimation based on a large constant number of sample-elements from the tree T_x. Each hash function h_i can be shown to perform size and median estimation correctly with probability at least $2/3$; by a Chernoff bound, it follows that the median of the size estimates acts as a correct size estimate with high probability, and that the median of the median estimates also works as a correct median estimate with high probability.

When eliminating flushing cascades due to node-merges, one additional property of the node-rebalancing mechanism will be required:

- **The Breathing-Space Property:** Say a node x is ***touched*** whenever an universal flush occurs either in x or in one of x's sibling nodes in level ℓ. Whenever a new node x is created at level ℓ, either via a node merge or a node split, the new node x is touched at least $\Omega(B^{\varepsilon(\ell-1)})$ times before being again involved in a merge or split.

The Breathing-Space Property can be enforced by the following modification to the summary-sketch-based node-rebalancing mechanism described above: Whenever a node x is created (due to a node-split or node-merge), the node x is ***frozen*** until the node x has been touched at least $B^{\varepsilon(\ell-1)}/c$ times, for some large constant c. The node x is only permitted to take part in merges and splits after the node stops being frozen (*even* if the summary sketch $\mathcal{S}_\ell(x)$ wishes to performs splits or merges earlier). A key observation is that, whenever the summary sketch $\mathcal{S}_\ell(x)$ for a node x first requires a split or merge, the time that node x must wait for x to become unfrozen (as well as for one of x's siblings to become unfrozen in the event of a merge) is small enough that the size of node x's tree $|T_x|$ will change by at most a constant fraction during the wait. Thus the Breathing-Space Property can be enforced without compromising the weight-balanced guarantees of the B$^\varepsilon$-tree.

Eliminating flushing cascades from node merges. The final problem is to eliminate flushing cascades caused by buffer overflows due to node merges.

One approach would be to eliminate node merges through the use of ***tombstone records***. When a record r is deleted, it is replaced with a tombstone indicating the deletion. Non-tombstone records are gradually copied to a second copy of the tree, which replaces the original tree every $O(n)$ operations.

The global-rebalancing approach is dissatisfying in that it requires the maintenance of two tree structures at a time. In Section 5.2 we show that, by exploiting the techniques already presented above, one can directly support deletions in the tree T without using tombstones.

We begin by describing the approach taken for nodes at levels $\ell > \frac{1}{\varepsilon} + 1$.

Whenever two nodes x_1 and x_2 are merged into a new node x at some level $\ell > \frac{1}{\varepsilon} + 1$, we consider the new node x to initially have an empty buffer, and for the old nodes x_1 and x_2 to be ***retired*** (but to still have potentially non-empty buffers). The new node x is deemed the ***caretaker*** for the two retired nodes.

Whenever node x or one of its siblings y are flushed out of, an additional ***trickle flush*** is performed in each of x_1 and x_2. A trickle flush removes a *single message* from a node's buffer and passes it to the next level of the B$^\varepsilon$-tree.

The Breathing-Space Property ensures that trickle flushes will entirely eliminate both x_1's and x_2's buffers before node x is next involved in either a split or merge. (Note that, in addition to trickle flushes, "regular" flushes may also be performed in x_1's and x_2's buffers whenever the "cup game" at level ℓ sees fit; this is necessary so that the buffers from retired nodes do not end up clogging the in-memory cache.)

The fact that trickle flushes do not cause additional flushing cascades is a consequence of each level of the tree being already modeled as a "cup game". In particular, the cup game ensures that even if the content that is flushed into a level originates from several nodes at the preceding level (instead of just a single node), this doesn't increase the risk of flushing cascades.

The strategy outlined above handles flushing cascades at levels $\ell > \frac{1}{\varepsilon} + 1$. At lower levels, a similar strategy is used, except with larger trickle flushes consisting of multiple messages (rather than just 1). The large trickle flushes are needed to compensate for the fact that, when ℓ is small, fewer trickle flushes occur between consecutive node-rebalances. These larger trickle flushes can cause a level ℓ to flush significantly more than B_ℓ messages to level $\ell - 1$; we show that this issue can be handled by slightly increasing the resource augmentation between the lower levels of the tree.

3 Preliminaries

The Disk Access Model. Due to the long latencies of accessing data stored on disk drives, algorithms that interact with data stored in external memory are often evaluated in the ***Disk-Access Model (DAM)*** [1]. The DAM model allows for an algorithm to interact with external memory by reading and writing in chunks of B machine words (the quantity B is known as the ***block size***), and defines the running time of an algorithm to be the number of such reads and writes performed, also known as I/Os. Algorithms are typically also allowed to access a small ***internal memory*** of some size M for free.

External-memory search trees. In order to exploit the fact

that the block size B is typically quite large, the data on disk is often laid out as a ***B-tree***. A B-tree is balanced tree (i.e., all leaves have the same depth) in which every node (except possibly for the root) stores a list of between B and $3B$ keys. For internal nodes of the tree, these keys represent the children of the node, and for leaf nodes, these keys are simply $\Theta(B)$ consecutive-valued keys in the data structure. Since the $\Theta(B)$ keys in each node fit in $O(1)$ blocks, lookups in B-trees can be performed in time proportional to the tree's depth, $\Theta(\log_B N)$. When B is large, this offers significant speedup over traditional binary trees.

B-trees can be made to also support insertions in time $O(\log_B N)$ by following the rule that whenever the number of keys stored in a node x increases past $3B$, the node splits into two nodes, each with $1.5B$ keys.[2] Note that this split increases the number of children for x's parent by one, and thus may also recursively trigger a split in x's parent. The only way that a B-tree can increase its depth is when the root node performs a split. This split creates a new root node one level higher.

Similarly, deletions can be supported in time $O(\log_B N)$ by allowing nodes x to merge with one of their neighbors whenever the number of keys stored in x decreases below B. The exception to this rule is the root node r, which never performs a merge and is instead simply removed once its number of children drops to one.

Rather than storing only keys, B-trees are typically used to store key/value pairs, allowing for one to store a value associated with each key.

As a convention, we say a node x is in the ***first level*** of a tree T if x is a leaf, and otherwise we say that x is at level $\ell + 1$, where ℓ is the level of x's children. This convention of indexing levels from the bottom of the tree (rather than the top) is particularly natural for B$^\varepsilon$-trees, since structurally new levels form when the node at the *top* of the tree splits.

Making writes faster with buffer-flushing techniques. Buffer flushing is a technique that can be used to transform a standard B-tree into a write-optimized data structure. Writes are made faster by buffering together large collections of insert/update/delete operations high in the tree, and then passing down collections of writes together. The first step in doing this is to reduce the number of children at each internal node in the tree to $O(B^\varepsilon)$ for some constant $0 < \varepsilon < 1$. Note that this only affects the depth of the tree by a factor of $\frac{1}{\varepsilon} = \Theta(1)$, and thus does not change the asymptotic running time of queries.

Each internal node x stores not only the $O(B^\varepsilon)$ children for x, but also a buffer of size $O(B)$ consisting of ***messages*** that are meant to be sent to x's children. Insertion/deletion/update operations are performed lazily by placing a message in the buffer of the root node indicating the operation that must occur. Messages percolate down the tree until they reach the leaf node containing the record to which they apply (or the leaf that would contain the record if it were present in the tree), at which point the corresponding insertion/deletion/update is performed. As described in the introduction, buffer flushing can be performed with an arbitrary buffer-eviction scheme; this is formalized in Section 3.1.

Note that, in order for insert/update/delete operations to be performed lazily, update/insertion/deletion operations cannot be permitted to perform a lookup for the key k that they apply to (e.g., the operation cannot return the previous status of key k). Thus update and insertion operations become essentially the same operation: an upsert on key k is performed with value v operates by inserting the key k with value v if k is not currently present, or updating key k's value to be v if k is currently present.

Since messages higher in the tree may modify a record at a leaf, lookups must examine not only a leaf node but also the root-to-leaf path that may contain any messages relevant to the lookup. In general, we use the term ***lookup*** to refer either to membership queries or to predecessor/successor queries. Both can be performed in $O(\log_B N)$ I/Os.

Tree conventions. Throughout the paper, when we refer to the *size* of a tree, we actually mean the *logical* number of records in a tree. If, for example, a record in the leaf of the tree has a deletion message sitting in a buffer higher in the tree, then that record is not counted in the size. As long as the height of the tree is at least some large constant, the contents of the leaves of the tree will always dominate the mass of the tree (i.e., there are many more records in leaves than messages in buffers), and thus the tree will always have height at most $O(\log_B N)$, where N is the size of the tree.

More generally, when discussing a node x, we say the ***size*** $|\boldsymbol{x}|$ of x, or equivalently the size of the subtree rooted at x, is the logical number of records in the subtree rooted at x, including the messages in x's buffer.

When two write messages for the same key are present in the same buffer, they can potentially be combined into a single message. For convenience, we assume that these combinations are not performed until the messages arrive at a leaf node. When we refer to the ***size*** $|\boldsymbol{b}|$ of a node x's buffer b, we count the total space required by that buffer, including messages with duplicate keys.

We define the ***siblings*** of a node x at level ℓ in a tree T to be the (up to two) nodes at level ℓ that are adjacent to x in the value ranges that they store. Note that the siblings of a node x need not have the same parent as x (and in Section 5 the notion of parent will stop being well defined, anyway, since the value range considered by a node will potentially cross between two parents).

As a convention, we use the term ***message*** to refer to a

[2] Here we describe the simplest node-merging and node-splitting strategies. Often more complex splitting strategies are actually used in order to achieve better constants.

buffered insert/update/delete operation and ***record*** to refer to a key-value pair that is either stored in a leaf, or is implicitly in the tree due to a message that has not yet made it do the leaves. (That is, the set of records in a tree is the same as the tree's logical size.)

Finally, for convenience, we overload the $\log_B N$ operator to return $\max(\log_B N, 1)$ (so that we do not need to worry about $\log_B N$ being sub-constant).

Allowing buffer-sizes to differ by constant factors. Throughout the paper, we fill find it useful to allow for the values of B used to size buffers at each level of the B^ε-tree to vary slightly. Given a ***buffer-size sequence***, $B_1, B_2, B_3, \ldots \in \mathbb{N}$ such that $B_i \in \Theta(B)$ for all i, a $(B_1, B_2, B_3, \ldots)$-sized B^ε-tree in one in which the buffer-sizes in nodes at level ℓ are determined by B_ℓ (rather than B). In particular, this means that an oblivious flush on nodes at level ℓ is performed on a buffer of size at least $3B_\ell$, and flushes exactly B_ℓ elements.

We will always assume that B is at least a sufficiently large constant multiple of $\log_B N$. It follows that, when we define a buffers-size sequence $B_1, B_2, \ldots$, we need only define the first B terms, $B_1, B_2, \ldots, B_B$. Given a sequence $p_1, p_2, \ldots, p_B \in [0, 1]$ satisfying $\sum_\ell p_\ell \leq O(1)$, we define the ***$(p_1, \ldots, p_B)$-induced buffer-size sequence*** by $B_B = B$, and $B_\ell = B_{\ell+1} + \lceil p_\ell B \rceil$ for $\ell \in \{B-1, \ldots, 1\}$. Note that this guarantees $B_\ell \in \Theta(B)$ and $B_\ell = (1+\Theta(p_\ell)) \cdot B_{\ell+1}$ for all $\ell \in [B-1]$.

Placing a small number of small buffers in memory. A key ingredient in our algorithms is the use of a small in-memory cache to store over-sized buffers. The in-memory cache is permitted to consist of polylogarithmically many blocks (i.e., we assume that $M \geq \mathrm{polylog}(N) \cdot B$). We call the buffer of a node x a ***cached buffer*** (and x a ***cached node***) if the buffer is stored in our in-memory cache. Whereas buffers not stored in memory must be flushed before their size exceeds $O(B)$, cached buffers are permitted to grow past $O(B)$ without being flushed. The combined size of all cached buffers, however, must not exceed $\mathrm{polylog}(N) \cdot B$.

3.1 Universal Flushes

In this section, we present the definition of a universal flush, and show how to use universal flushes in order to simulate arbitrary buffer-eviction strategies. This allows for us to use universal flushes as the primitive through which our data structure interacts with nodes, while abstracting away the underlying buffer-eviction strategy.

A ***universal flush*** on a node x can be performed whenever the node's buffer contains $3B$ or more records. The universal flush is then required to select *exactly* B records to evict from that node.

In this section we prove that, given an arbitrary buffer-eviction strategy $\mathcal{A}$, one can implement universal flushing to simulate the behavior of $\mathcal{A}$ on a buffer of size B.

Formally, a ***buffer-eviction strategy*** $\mathcal{A}$ is any algorithm for mapping a B-element multiset M consisting of at most $O(B^\varepsilon)$ distinct elements to a subset $S \subseteq M$. The set M represents the set of records contained in a node's buffer (with each record represented by the child to which it is destined), and the set S represents a selection of which records to evict from the node.

Given a buffer-eviction strategy $\mathcal{A}$, and sequence of numbers $K = (k_1, k_2, \ldots, k_m)$ with each $k_i \in \{1, 2, \ldots, B^\varepsilon\}$, the ***I/O-complexity of strategy $\mathcal{A}$ on sequence K*** can be computed as follows. Fill a B-element buffer to contain $k_1, \ldots, k_B$, and then use $\mathcal{A}$ to select a subset S_1 of the elements in the buffer to evict; record the number d_1 of distinct elements in S_1. Then fill the buffer back to size B using elements $k_{B+1}, k_{B+2}, \ldots$, and again use $\mathcal{A}$ to select a subset S_2 of the elements in the buffer to evict; record the number d_2 of distinct elements in S_2. Repeat this until every element $k_i \in K$ has at some point been placed into the buffer, and the buffer size is less than B. The sum $\sum_i d_i$ is the I/O complexity of $\mathcal{A}$ on sequence K, and represents the number of distinct times that eviction strategy $\mathcal{A}$ touches a child while handling the input stream K.

The ***I/O-complexity of universal flushing on sequence K*** is computed in the same way, except that the buffer is refilled to size $3B$ between successive flushes, instead of only to size B.

The next proposition shows how to use universal buffer flushing in order to simulate any buffer-eviction strategy $\mathcal{A}$.

PROPOSITION 3.1. *Given any buffer-eviction strategy $\mathcal{A}$, there is an implementation of universal flushing $\mathcal{U}$ such that on any sequence of numbers $K = (k_1, k_2, \ldots, k_m)$ with each $k_i \in \{1, 2, \ldots, B^\varepsilon\}$, the I/O-complexity of $\mathcal{U}$ on K is at most twice the I/O complexity of $\mathcal{A}$ on K.*

Proof. Given a buffer-eviction strategy $\mathcal{A}$, we can implement universal flushing $\mathcal{U}$ as follows: Partition our buffer u into a ***future buffer*** u_1, a ***current buffer*** u_2, and a ***past buffer*** u_3. Whenever new elements are placed into the buffer u, they are inserted directly into the future buffer u_1. In order to perform a universal flush on the buffer, we first move records from u_1 to u_2 until u_2 is of size B; next we use the buffer-eviction strategy $\mathcal{A}$ to select a subset of records in u_2 to evict and we move those records to u_3; then, we repeat these two steps until u_3 has size at least B. Finally, we evict exactly B records from u_3; any remaining records in u_3 will be evicted during the next universal flush.[3]

[3] Several of our algorithms will allow for up to B ***dummy records*** to be present in buffer u_1. We may feel free to treat these dummy records as always being present in the future buffer u_1 and never being moved to buffer u_2. Since universal flushes are only performed when $u = u_1 \cup u_2 \cup u_3$ is size at least $3B$, and since u_3 never exceeds size B at the beginning of a universal flush, one can always fill u_2 to size B using entries from u_1

Now consider the I/O complexity of $\mathcal{A}$ on sequence K. Let S_i denote the set of elements evicted in the i-th eviction by $\mathcal{A}$ on the sequence K, and let d_i denote the number of distinct elements in S_i. In particular, the I/O complexity of $\mathcal{A}$ on K is given by $\sum_i d_i$.

When the universal flushing strategy $\mathcal{U}$ is performed on sequence K, the i-th eviction from the current-buffer u_2 evicts exactly the set S_i from the u_2 to u_3. Whenever a set S_i is evicted from u_2 it is guaranteed to be evicted from u_3 by the end of the next universal flush. It follows that each set S_i incurs at most $2d_i$ flushes for $\mathcal{U}$. Thus the I/O complexity of $\mathcal{U}$ on K is at most $2\sum_i d_i$, as desired. □

The preceding proof shows how to implement universal flushes directly in terms of a buffer-eviction strategy $\mathcal{A}$. More generally, *any* implementation of universal flushing that selects a set of exactly B records to evict will be compatible with our final data structure. It would be allowable, for example, for the eviction strategy to make eviction decisions with full knowledge of u_1 and u_3. Moreover, even if an eviction were to try to adversarially *minimize* write-optimization, the fact that universal flushes send B records to $O(B^\varepsilon)$ children ensures the bare-minimum amount of write-optimization required to achieve B$^\varepsilon$-tree-like guarantees.

Note that, from the perspective of flushing cascades, universal flushes essentially represent the worst possible flushing strategy. If the B elements flushed by the universal flush go to $\Theta(B^\varepsilon)$ different children, then each of those children's buffers could easily overflow; those buffers could then also evict to $\Theta(B^\varepsilon)$ different children each, and so on. Nonetheless, we will present a suite of randomization techniques that allow for flushing cascades to be avoided in this setting.

4 The Static Cup-Based B$^\varepsilon$-Tree

We begin by designing flushing strategies for static B^ε-trees, in which the only operations to the tree are update-operations (for records that are already present). In particular, we are interested in designing what we call a global flushing algorithm:

DEFINITION 1. *A **global flushing algorithm** with buffer-size sequence $(B_1, B_2, B_3, \ldots)$ is any algorithm for deciding which nodes to perform oblivious flushes on during each operation of a B^ε-tree. The oblivious flushes are then performed using buffer-size sequence $(B_1, B_2, B_3, \ldots)$. Additionally, a global flushing algorithm may mark certain nodes as **cached**.*

A global flushing algorithm must provide two guarantees: (1) If the tree size is N, then the sum of the sizes of the buffers of all cached nodes is at most polylog$(N) \cdot B$, the size of internal memory; and (2) Every uncached node deterministically has buffer-size at most $O(B)$.

The purpose of the section is to prove the following theorem:

THEOREM 4.1. *Let $0 < \varepsilon < 1$ be a constant. Suppose that $M \geq (\log N)^c B$ for a sufficiently large constant c, and that B is at least a sufficiently large constant multiple of $\log_B N$.*

Let T be a B$^\varepsilon$-tree on N records, and suppose that T's buffers are initially empty. Consider a workload consisting only of read operations and update operations (to records already in the tree).

Then there exists a global flushing algorithm for T that achieves worst-case update-operation time

$$O\left(\lceil \log_B N / B^{1-\varepsilon} \rceil\right).$$

Analyzing Flushing Rounds Instead of Operations.

DEFINITION 2. *A **flushing round** occurs whenever a flush is performed in the root node of a tree. The flushing round consists of any further flushes that are performed as a consequence.*

To prove Theorem 4.1, we design a global flushing algorithm with two properties. First, successive flushing rounds are always separated by $\Omega(B)$ operations. And second, each flushing round requires at most $O(\log_B N)$ universal flushes with high probability in N.

Since each universal flush requires at most $O(B^\varepsilon)$ I/Os (in order to touch each of the node's children), the I/O-complexity of a flushing round is $O(B^\varepsilon \log_B N)$. By spreading the work for each flushing round across $\Omega(B)$ operations, we can then obtain worst-case update-time $O\left(\lceil \log_B N / B^{1-\varepsilon} \rceil\right)$, as desired by Theorem 4.1.

Randomizing node overflows with dummy messages. A key step in our algorithm is to initially place a random number $r_x \in \{0, 1, 2, \ldots, B_\ell - 1\}$ of ***dummy messages*** into the buffer of each node x in each level ℓ. The quantity r_x is known as the ***random initial offset*** at node x. The dummy messages are for bookkeeping purposes only, and are never flushed from the buffer in which they reside. (If universal flushes are implemented in terms of a ball-recycling strategy, as described in Section 3, then one can follow the rule that the dummy messages never leave the future buffer u_1.)

We say that a node x in level ℓ ***overflows*** whenever the fill of the node (i.e., the number of messages, including dummy messages, that reside in node x's buffer) increases from a value less than $3B_\ell$ to a value at least $3B_\ell$. More generally, we say that a node ***crosses a threshold*** whenever the fill of a node increases from a value less than kB_ℓ to a

without using any dummy records.

value at least kB_ℓ for an integer $k \geq 3$. (Note that a fill of $3B_\ell$ including dummy messages may correspond with a fill of as little as $2B_\ell + 1$ excluding dummy messages.)

The presence of r_x dummy messages in each node x randomizes the threshold crossings so that the number of crossings in nodes at a given level ℓ during a sequence of flushing rounds $t_0, \ldots, t_1$ is a sum of well-behaved random variables. Specifically, if j messages are flushed from level $\ell + 1$ to level ℓ during a time interval, then the number of crossings in nodes at level ℓ is a sum of independent indicator random variables with total mean j/B_ℓ.

LEMMA 4.1. *Suppose that a global flushing algorithm is used in which flushes at levels $\ell+1, \ell+2, \ldots$ are independent of the random thresholds r_x used for nodes x at level ℓ. At each level ℓ, define an **input round** to be the time interval between two consecutive universal flushes at level $\ell + 1$ (containing the first of the two).*

Let j be the number of messages flushed from level $\ell + 1$ to level ℓ during input rounds $t_0, \ldots, t_1$ for some $t_0, t_1 \in \mathbb{Z}$. Then the number of threshold crossings in level ℓ during input rounds $t_0, \ldots, t_1$ is bounded above by a sum of independent 0-1 random variables with mean j/B_ℓ.

Proof. For each node x at level ℓ, let g_x be the number of messages placed into node x's buffer during input rounds $1, \ldots, t_0$, including the initial r_x dummy messages used for bookkeeping. Since the random value of r_x is independent of the number of messages $g_x - r_x$ to have arrived from level $\ell - 1$ during those input rounds, the quantity g_x is the sum of two independent random variables, one of which is uniformly random in $\{0, \ldots, B_\ell - 1\}$. It follows that $g_x \pmod{B_\ell}$ is uniformly random in $\{0, \ldots, B_\ell - 1\}$.

Whenever an universal flush is performed in cup x, exactly B_ℓ messages are removed from the buffer. This ensures that, at the beginning of input round t_0, the number of messages a_x presently in node x's buffer satisfies $a_x \equiv g_x \mod B_\ell$. Thus $a_x \pmod{B_\ell}$ is uniformly random in $\{0, \ldots, B_{\ell-1}\}$.

Let j_x denote the number of messages that arrive into cup x's buffer during input rounds $t_0, \ldots, t_1$. Note that every time a threshold crossing occurs in node x, the total number of messages to have been placed in node x's buffer, over all time, must cross a multiple of B_ℓ. (Here, again, we are exploiting the fact that universal flushes remove exactly B_ℓ messages, thereby not changing the number of messages modulo B_ℓ in the buffer.) It follows that the first $j_x - (j_x \pmod{B_\ell})$ messages to arrive into x's buffer during input round $t_0, \ldots, t_1$ can induce at most $\lfloor j_x/B_\ell \rfloor$ threshold crossings in cup x. The final $j_x \pmod{B_\ell}$ can induce an additional threshold crossing only if $(a_x \pmod{B_\ell}) \geq B_\ell - (j_x \pmod{B_\ell})$. Since $a_x \pmod{B_\ell}$ is uniformly random, it follows that the extra threshold crossing occurs with probability at most $(j_x \pmod{B_\ell})/B_\ell$.

The total number of threshold crossings in cup x during input rounds $t_0, \ldots, t_1$ is therefore bounded above by

$$\left(\sum_{i=1}^{\lfloor j_x/B_\ell \rfloor} 1 \right) + Y_x,$$

where Y_x is 0-1 random variable that takes value 1 with probability $(j_x \pmod{B_\ell})/B_\ell$. Summing over nodes x at level ℓ, the total number of threshold crossings at level ℓ during input rounds $t_0, \ldots, t_1$ is at most

$$\sum_{x \in \text{ level } \ell} \left(Y_x + \sum_{i=1}^{\lfloor j_x/B_\ell \rfloor} 1 \right).$$

The above sum is a sum of independent 0-1 random variables, since the outcome of each Y_x is randomized by r_x, and since the random variable 1 is trivially independent of all other random variables. Moreover, the sum has mean

$$\sum_{x \in \text{ level } \ell} j_x/B_\ell = j/B_\ell,$$

as desired. ∎

Two randomized flushing algorithms. The first algorithm we consider is the ***aggressive randomized flushing algorithm***, which performs an universal flush on each node x whenever that node overflows. One advantage of the aggressive randomized flushing algorithm is that it forgoes the use of caching; the guarantee that we give for the algorithm is weaker than Theorem 4.1, however, bounding the number of flushes across collections of $\text{polylog}(N)$ consecutive flushing rounds, instead of in a single flushing round. We begin by analyzing the aggressive randomized flushing algorithm, and then extend the analysis to our final algorithm, which we call the lazy randomized flushing algorithm.

PROPOSITION 4.1. *Suppose that B is at least a sufficiently large constant multiple of $\log_B N$. Let T be a B^ε-tree on N records, and suppose that T's buffers are initially empty. Consider a workload consisting only of read operations and update operations (to records already in the tree). Suppose that the operations are performed using the aggressive randomized flushing algorithm, and using the $(\frac{1}{1^2}, \frac{1}{2^2}, \frac{1}{3^2}, \ldots)$-induced buffer-size sequence.*

Define $k = (\log N)^c$. Then for any sequence of k flushing rounds, the total number of universal flushes in the k rounds is at most $O(k \cdot \log_B N)$, with high probability in N.

Proof. Consider a sequence of k flushing rounds $t_0, \ldots, t_1$. For each level ℓ, let j_ℓ be the number of universal flushes performed by the aggressive randomized flushing algorithm at level ℓ during rounds $t_0, \ldots, t_1$.

Let h be the height of the tree T. Then j_h is exactly the number of flushing rounds k. For $\ell < h$, the number of messages flushed from level $\ell + 1$ to level ℓ during rounds $t_0, \ldots, t_1$ is exactly

$$j_{\ell+1} \cdot B_{\ell+1}.$$

By Lemma 4.1, the number of threshold crossings at level ℓ, which is in turn exactly j_ℓ, is a sum of independent 0-1 random variables with mean

$$j_{\ell+1} \cdot \frac{B_{\ell+1}}{B_\ell}.$$

The use of the $(\frac{1}{1^2}, \frac{1}{2^2}, \frac{1}{3^2}, \ldots)$-induced buffer-size sequence ensures that $\frac{B_{\ell+1}}{B_\ell} \leq 1 - \Omega(1/\ell^2)$. It follows that for each $\ell < h$, j_ℓ is a sum of independent 0-1 random variables with mean at most

$$j_{\ell+1}(1 - \Omega(1/\ell^2)) \leq j_{\ell+1}(1 - \Omega(1/\log^2 N)).$$

By a Chernoff bound, assuming that $k \geq (\log N)^c$ for a large enough constant c, the probability that $j_\ell > k$ given that $j_{\ell+1} \leq k$ is $\frac{1}{\text{poly}(N)}$. Thus, with high probability in N, no $j_\ell \leq k$ for all levels ℓ. ∎

We remark that the use of the $(\frac{1}{1^2}, \frac{1}{2^2}, \frac{1}{3^2}, \ldots)$-induced buffer-size sequence is not necessary for Proposition 4.1. In particular, as noted in Section 2, even if every level uses the same block-size, one can argue that over the course of a large number j of flushes from level $\ell+1$ to level ℓ, only at most $(1 + \frac{1}{\log_B N_{\max}})j$ flushes are caused at level ℓ. The use of the buffer-size sequence allows for the stronger property that at most j flushes are performed at level ℓ, which slightly simplifies the proof. The induced buffer-size sequence will also play a similar (and more critical) role in the analysis of the aggressive randomized flushing algorithm.

Although Proposition 4.1 shows that the aggressive randomized flushing algorithm behaves well across batches of polylogarithmically many flushing rounds, any individual flushing round could potentially involve an unacceptably large number of universal flushes. The ***lazy randomized flushing algorithm*** resolves this problem by caching the buffers for a small number of nodes in internal memory. During each flushing round, and for each level ℓ (beginning with the root level $\ell = h$), the algorithm selects the node x with the fullest buffer, and if x's buffer is of size at least $3B_\ell$, then the algorithm performs an universal flush on x. Any other buffer of size $3B_\ell$ or more is cached.

Finally, at the end of each flushing round, there is a cleanup step that occurs with low probability. If, at the end of any round, the total combined size of the cached buffers at a level ℓ reaches $M/(1 + \log_B N) - B_1$, then the algorithm selects the smallest such level ℓ, and performs an additional universal flush in each of level $\ell, \ell-1, \ell-2, \ldots, 2$. If there remains a level ℓ with a cache of size at least $M/(1 + \log_B N) - B_1$, then the algorithm again selects the lowest such ℓ, and repeats the process until no such ℓ exists. Note that by selecting the lowest such ℓ each time, the algorithm ensures that a level ℓ' can only be flushed *into* if level ℓ' has a cache of size smaller than $M/(1 + \log_B N)$. The combined sizes of the caches across all levels therefore does not exceed M.

Note that, even with the cleanup step, the lazy randomized flushing algorithm continues to satisfy the requirement by Lemma 4.1 that flushes at levels $\ell + 1, \ell + 2, \ldots$ are independent of the random thresholds r_x used for nodes x at level ℓ. For this property to remain true, it is important that cleanup is performed based on the cache-size at each level, rather than based on the total cache size of the tree.

With the exception of when the cached buffers cause cleanup steps, the lazy algorithm performs at most $O(\log_B N)$ universal flushes during each flushing round. In order to complete Theorem 4.1 it suffices to bound the sum of the sizes of the cached buffers, demonstrating that cleanup steps are rare. This is accomplished by Lemma 4.2.

LEMMA 4.2. *Suppose that B is at least a sufficiently large constant multiple of $\log_B N$. Let T be a B^ε-tree on n records, and suppose that T's buffers are initially empty. Consider a workload consisting only of read operations and update operations (to records already in the tree). Suppose that the operations are performed using the lazy randomized flushing algorithm, and using the $(\frac{1}{1^2}, \frac{1}{2^2}, \frac{1}{3^2}, \ldots)$-induced buffer-size sequence.*

Then after any flushing round, the combined sizes of the cached buffers is at most $B \cdot \text{polylog}(N)$, with high probability in N.

Proof. As in Lemma 4.1, define an input round at each level ℓ to be the time period beginning just before an universal flush at level $\ell + 1$ and ending just before the next universal flush at level $\ell+1$. Note that level ℓ may experience multiple input rounds during a single flushing round due to clean-up steps within the flushing round.

Let h be the level of the root node. For a level $\ell < h$ and a input round t, define the potential function $Q_\ell(t)$ to be,

$$\sum_{x \in A_t} \lceil (|x| - 2B_\ell)/B_\ell \rceil,$$

where A_t is the set of cached buffers in level ℓ after input round t.

We prove that for each ℓ and t, $Q_\ell(t) \leq \text{polylog}(N)$ with high probability in N. This completes the lemma, since $3Q_\ell(t)$ is at least as large as the combined sizes of the cached buffers at level ℓ.

Note that for each ℓ and $t \geq 1$, the value $Q_\ell(t)$ is either zero, or satisfies

$$Q_\ell(t) \leq Q_\ell(t-1) + u_t - 1,$$

where u_t is the number of threshold crossings at level ℓ during input round t. In particular, each threshold crossing increases Q_ℓ by one, and then the universal flush performed by the lazy flushing algorithm reduces Q_ℓ by 1 (unless Q_ℓ is already 0).

Thus, in order for $Q_\ell(t)$ to be large, say $(\log N)^c$ for a large constant c, there must be some $m > 0$ such that

$$\sum_{i=t-m+1}^{t} (u_i - 1) \geq (\log N)^c.$$

Rearranging gives

$$\sum_{i=t-m+1}^{t} u_i \geq (\log N)^c + m. \tag{4.1}$$

Lemma 4.1 establishes that $\sum_{i=t-m+1}^{t} u_i$ is bounded above by a sum of independent random variables with mean at most

$$m \cdot \frac{B_{\ell+1}}{B_\ell} = m \cdot (1 - \Omega(1/\log^2 N)),$$

since at most $B_{\ell+1}$ messages are flushed from level $\ell+1$ to ℓ during each input round.

By a Chernoff bound, when $m \leq (\log N)^c$, the probability that Eq. 4.1 occurs is polynomially small in N (i.e., at most $\frac{1}{\text{poly}(N)}$). Thus with high probability in N, Eq. 4.1 does not hold for any $m \leq (\log N)^c$.

On the other hand, when $m \geq (\log N)^c$, we get by a Chernoff bound that

$$\Pr\left[\sum_{i=t-m+1}^{t} u_i \geq m\right] \leq \exp\left(-\Omega\left(\frac{1}{\log^4 N} m\right)\right).$$

By a union bound over all $m \geq (\log N)^c$, it follows that with high probability in N, Eq. 4.1 does not hold for any $m \geq (\log N)^c$.

Since Eq. 4.1 fails for all m with high probability in N, we have that $Q_\ell(t) \leq (\log N)^c$ with high probability in N, as desired. ∎

An additional property of the Lazy Algorithm. We conclude the section by noting one additional useful property of the lazy randomized flushing algorithm. Specifically, we show that although the sum of the sizes of the cached buffers may in the worst case be M, the size of the largest cached buffer never exceeds $O(B \log M)$.

LEMMA 4.3. *Let T be a B$^\varepsilon$-tree on N records, and suppose that T's buffers are initially empty. Consider a workload consisting only of read operations and update operations (to records already in the tree). Suppose that the flushing operations are performed using the lazy randomized flushing algorithm.*

Then after any flushing round, the maximum size of any cached buffer in a level ℓ is $O(B \cdot \log m)$, where m is the number of cached nodes in level ℓ. Note that if the cached buffers fit into memory, then this implies that the maximum size of any cached buffer is also at most $O(B \cdot \log M)$.

Proof. We prove the lemma by modeling the cached buffers as cups in a cup-emptying game. Each cached buffer x at level ℓ can be thought of as a cup containing w_x units of water, where $(w_x+3)B_\ell$ is the number of messages in buffer x. Using this analogy, in each input round, at most 1 unit of water is placed into cups, and then 1 unit of water is removed from the fullest cup (unless that cup contains less than 1 unit of water, in which case the cup is emptied). This is precisely the ***dynamic cup game*** analyzed by [11]. It follows that the amount of water in the fullest cup is at most $O(\log m)$, where m is the number of non-empty cups. Thus the fullest cached buffer at level ℓ has size at most $O(B \log m)$, where m is the number of cached buffers. ∎

5 The Dynamic Cup-Based B$^\varepsilon$-Tree

In this section, we introduce the Dynamic Cup-Based B$^\varepsilon$-tree, which extends the techniques in Section 4 to support arbitrary workloads of queries and upserts. Throughout the section, we use s_ℓ as shorthand for $B^{1-\varepsilon} \cdot B^{\varepsilon\ell}$, which one should think of as approximately representing the size of a node at level ℓ.

THEOREM 5.1. *Let $0 < \varepsilon < 1$ be a constant. Suppose that $M \geq cB^2 + (\log N_{\max})^c B$ for a sufficiently large constant c, and suppose that B is at least a sufficiently large constant multiple of $\log N_{\max}$.*

Consider an arbitrarily long sequence of insert/update/delete operations on an initially empty tree T, and let $N_{\max}$ denote the maximum size of T during those operations. If T is implemented as a dynamic cup-based B$^\varepsilon$-tree, then T will deterministically never use memory more than M. The tree T will satisfy the following performance guarantees:

- ***Worst-Case Write Costs:*** *For each insert/update/delete operation operation i, with high probability in $N_{\max}$, the operation i has I/O cost at most*

  $$O(\lceil (\log_B N_i)/B^{1-\varepsilon} \rceil),$$

  where N_i is the size of the tree after operation i.

- ***Domination by Universal Flushes:*** *For a given flushing round i, the number of I/Os incurred by the flushing round is, with high probability in $N_{\max}$, at most $O(B^\varepsilon + q)$, where q is the number of I/Os incurred by universal flushes during the flushing round.*

- ***Amortized Operation Costs:*** *For any $n > 0$, with high probability in $N_{\max}$, the total I/O cost of the the first n insert/update/delete operations is*

$$O\left(\sum_{i=1}^{n} \frac{\log_B N_i}{B^{1-\varepsilon}}\right).$$

- ***Worst-Case Read Costs:*** *For any insert/update/delete operation i, with high probability in $N_{\max}$, any read performed after operation i incurs $O(\log_B N_i)$ I/Os.*

We assume throughout the section that $M \leq \text{poly}(B)$, which is w.l.o.g. since the requirement $M \geq cB^2 + (\log N_{\max})^c B$ is satisfied by $M \in \text{poly}(B)$. Importantly, the assumption affects the cleanup phase in lazy randomized flushing algorithm, and implies by Lemma 4.3 that the maximum size of a stashed buffer never exceeds $O(B \log B)$.

Supporting structural changes to the static cup-based B^ε-tree introduced in Section 4 leads to two technical issues. The first is the handling of flushing cascades caused by node merges. The second, more subtle issue, is the introduction of cyclic dependencies between levels.

Cyclic dependencies between levels. In order for the randomized flushing strategy at each level ℓ of the tree to be effective (and, specifically, for Lemma 4.1 to hold), it is necessary that the input flushes from level $\ell + 1$ to level ℓ be independent of the random initial offsets used in level ℓ. That is, for all $i \geq 1$, the set of messages flushed by the i-th universal flush in level $\ell + 1$ must be the same regardless of the random initial offsets used in levels $\ell, \ell - 1, \ldots, 1$.

Node merges and splits violate this property by introducing cyclic dependencies between levels. The random bits used at level ℓ partially determine which messages make it into lower levels by a given point in time, and thus which messages make it to the leaves by a given point in time. The arrival of messages to the leaves of the B^ε-tree determines when (and which) nodes at level 2 of the tree merge and split. The merging and splitting of nodes at level 2 in the tree then, in turn, influences when nodes at level 3 of the tree merge and split, etc. Thus the random bits at level ℓ indirectly influence the set of nodes at level $\ell + 1$, which in turn affects the flushing decisions made in level $\ell + 1$.

In designing the Dynamic Cup-Based B^ε-tree, we begin in Section 5.1 by adapting the lazy randomized flushing algorithm to the dynamic-structure setting in order to eliminate flushing cascades due to node merges. The new flushing algorithm assumes that the underlying node-rebalancing mechanism has a number of useful properties and avoids cyclic level dependencies. In order to design such a node-rebalancing mechanism, Section 5.2 then introduces a randomized weight-balancing approach in which each node maintains a summary sketch of the subtree that it roots, and uses the summary sketch in order to determine when to split or merge with neighbors.

5.1 Eliminating Flushing Cascades In this section, we adapt the lazy randomized flushing algorithm to handle structural changes to the tree T resulting from node merges and splits. The new flushing algorithm assumes certain important properties from the mechanism that is used for rebalancing nodes (i.e., the algorithm for determining when to perform node merges and node splits). Designing a rebalancing mechanism that guarantees these properties will then be the primary purpose of Section 5.2.

The first property required of the rebalancing mechanism is the Independent Levels Property.

- **The Independent Levels Property:** The key ranges of the nodes in level ℓ of the tree are determined only by (1) the set of messages that have ever been flushed from level $\ell + 1$ to level ℓ; and (2) random bits associated with level ℓ of the tree.

The Independent Levels Property eliminates cyclic dependencies between levels. The property ensures that, as long as the flushing algorithm at each level ℓ makes decisions based only on the contents of the buffers at level ℓ, then the contents of the i-th flush from level $\ell + 1$ to level ℓ will be independent of the random bits used by the flushing algorithm (and the rebalancing mechanism) in level ℓ. This, in turn, allows for Lemma 4.1 to be used at each level ℓ.

Note that, in order for the Independent Levels Property to be satisfied, it is necessary for the key ranges at each level ℓ to not necessarily be strict subsets of the key ranges at the higher level $\ell+1$. This means that our B^ε-tree has a structure similar to that of a skip list, in that adjacent nodes at level ℓ may have a child in common at level $\ell - 1$.

The second property required from the rebalancing mechanism is The Breathing-Space Property, which will prove useful in adapting the flushing algorithm to handle flushing cascades due to buffer merges.

- **The Breathing-Space Property:** Say a node x is ***touched*** whenever an universal flush occurs either in x or in one of x's sibling nodes in level ℓ. (Note that the sibling nodes of x in level ℓ need not share a parent with x.) Whenever a new node x is created at level ℓ, either via a node merge or a node split, the new node x is touched at least $\Omega(s_\ell/B)$ times before being again involved in a merge or split.

Finally (and perhaps most obviously), the rebalancing mechanism should guarantee a B^ε-tree-like structure with high probability in $N_{\max}$. Specifically, at any point in time, we require that the Balanced-Tree Property hold with high probability in $N_{\max}$:

- **The Balanced-Tree Property:** For each level ℓ such satisfying $s_\ell < 20|T|$, and for each node x at level ℓ, the size of the subtree T_x rooted at x must be $\Theta(s_\ell)$. Moreover, for any level $\ell \leq B$ satisfying $s_\ell > 20|T|$, the number of nodes at level ℓ must be at most one and must be stored in memory. Additionally, the metadata used by the rebalancing mechanism must consist of, with high probability in $N_{\max}$, no more than $O(B)$ machine words per node in the tree, and no more than $O(B^2)$ additional blocks for non-node-specific metadata.

Note, in particular, that the Balanced-Tree Property bounds the number of children that each node x can have to $O(B^\varepsilon)$, which is important both for keeping the size of the node small (so that it fits in a constant number of blocks) and for achieving write-optimized performance for insert/update/delete operations.

We call the flushing algorithm introduced in this section the ***dynamic lazy flushing algorithm***.

PROPOSITION 5.1. *Let $0 < \varepsilon < 1$ be a constant. Suppose that $cB^2 + (\log n)^c B \leq M \leq poly(B)$ for a sufficiently large constant c, and that B is at least a sufficiently large constant multiple of $\log_B n$. Assume that the rebalancing mechanism satisfies The Independent Levels Property, the Breathing-Space Property, and the Balanced-Tree Property (with high probability in $N_{\max}$ at any point in time). Then the dynamic lazy flushing algorithm satisfies the properties desired by Theorem 5.1.*

The dynamic lazy flushing algorithm performs universal flushes using the same approach as the (static) lazy flushing algorithm. During each flushing round, and for each level ℓ (beginning with the root level), the algorithm selects the node x with the fullest buffer, and if x's buffer is of size at least $3B_\ell$, then the algorithm performs an universal flush on x. Any other buffer of size $3B_\ell$ or more is stashed.

At the end of a flushing round, a cleanup step (that occurs with low probability) is also performed in the same manner as for the (static) lazy flushing algorithm. Specifically, if h is the largest level to contain multiple nodes, then cleanup is performed as necessary to ensure that no level ℓ contains a stash of size greater than $(M - O(B^2))/h - B$, where the $O(B^2)$ term is the sum of the maximum space in memory that may be required from levels at which there is only a single node, along with the space required by the rebalancing mechanism for non-node-specific metadata.

In addition to performing universal flushes, the dynamic lazy flushing algorithm performs a second type of flush called a ***trickle flush***, described below.

Retiring buffers on merges and splits. When two nodes x_1 and x_2 at a level ℓ are combined to form a new node y, we create a new buffer b for node y (along with a new random initial offset), and we ***retire*** the buffers a_1, a_2 for nodes x_1 and x_2. The new buffer b for node y is assigned as the ***caretaker*** for retired buffers a_1 and a_2. Similarly, when a buffer y is split into two nodes x_1 and x_2, y's buffer b is retired, and two new buffers a_1 and a_2 are created for x_1 and x_2; in this case, both a_1 and a_2 are said to be b's caretakers.

Once a buffer u is retired, it is guaranteed that no further messages will be placed into that buffer (since such messages are now sent to u's caretakers). The lazy randomized flushing algorithm may still perform universal flushes on a retired buffer, however, if that buffer is the fullest of any buffer at level ℓ. (And, importantly, if a stashed buffer is retired, that buffer remains stashed.)

Trickle flushes from retired buffers. Whenever an universal flush is performed on a node x at a level $\ell > \frac{1}{\varepsilon} + 1$, *one message* is flushed from each of the (up to six) retired buffers for which x's siblings or x are caretakers. This is called a ***trickle flush*** (since it flushes only a very small amount). By the Breathing-Space Property, whenever a node x at level $\ell > \frac{1}{\varepsilon} + 1$ is rebalanced, any nodes y for which x is a caretaker will have had at least $\Omega(s_\ell/B) \geq \Omega(B^{1+\varepsilon})$ trickle flushes. Since B is at least a sufficiently large constant, B^ε is at least a sufficiently large constant multiple of $\log M \leq O(\log B)$, and it follows that as long as no retired buffer ever has buffer-size greater than $O(B \log B)$, then any nodes for which x is a caretaker will have an empty buffer by the time x is rebalanced.

When an universal flush is performed on a node x in a level $\ell \leq \frac{1}{\varepsilon} + 1$, $\lceil \varepsilon B \rceil$ messages are flushed from each retired node for which x's siblings or x are caretakers. This is again called a trickle flush (although it is much larger than the trickle flushes at levels $\ell > \frac{1}{\varepsilon} + 1$). Note that the εB messages are not flushed using an universal flush, but are instead selected arbitrarily from the retired buffers. Since at least B^ε-trickle flushes are performed between rebalances of a node x, by the time the node x is next rebalanced its caretakers will each have had at least $\Omega(\varepsilon B^{1+\varepsilon}) \gg B \log B$ messages removed from them via trickle flushes; thus all of the nodes for which x is a caretaker have empty buffers when x is next rebalanced (assuming they initially had buffer-size no greater than $O(B \log B)$).

We remark that extra care must be taken to ensure that when a node x is a caretaker for a node y, the messages that share a given key k are flushed out of x's and y's buffers in the same order that they arrived (since the order of arrival corresponds with the order in which they should be applied to the key k). That is, if a message r is flushed from x, but y contains a message r' with the same key as r, then we actually replace the message r' with r in y's buffer and flush the message r instead of the message r'.[4]

[4]Note that this maintains the invariant that for each key k, the messages with key k in y's buffer are all older than the messages with key k in x's

Analyzing dynamic lazy flushing. Trickle flushes make it so that up to $B_{\ell+1}+1$ messages may be flushed from level $\ell+1$ to level ℓ on a given step, when $\ell \geq \frac{1}{\varepsilon}+1$, and up to $B_{\ell+1}+\lceil \varepsilon B \rceil$ messages may be flushed from level $\ell+1$ to level ℓ for $\ell < \frac{1}{\varepsilon}+1$. To absorb this into the resource augmentation between levels, we modify the buffer-size sequence used to be the $(p_1, \ldots, p_B)$-induced buffer-size sequence, where

$$p_i = \begin{cases} \frac{1}{i^2} + \varepsilon & \text{if } i \leq \frac{1}{\varepsilon}+1 \\ \frac{1}{i^2} & \text{if } i > \frac{1}{\varepsilon}+1 \end{cases}.$$

Since $\sum_{i=1}^{B} p_i = O(1)$, the buffer-size sequence satisfies $B_\ell \in \Theta(B)$ for $\ell = 1, \ldots, B$. Moreover, the buffer-size sequence enforces the property that the maximum number of messages r_ℓ that may be flushed from level $\ell+1$ to level ℓ during a flushing round satisfies

$$\frac{r_\ell}{B_\ell} \leq \left(1 - \frac{1}{\Omega(\ell^2)}\right) \leq \left(1 - \Omega\left(\frac{1}{\log^2 N_{\max}}\right)\right),$$

which is precisely the property needed from the buffer-size sequence in the proof of Lemma 4.2.

Trickle flushes also violate the property used in Lemma 4.1 that flushes from a node x at level ℓ always flush exactly B_ℓ elements. This property was used to ensure that the size of each buffer modulo B_ℓ is independent of the random threshold at that buffer, and thus that threshold crossings in each buffer occur at random points in time. However, the property is unnecessary in retired buffers, since new messages are never added to retired buffers, and thus retired buffers never incur threshold crossings.

Besides the issues discussed above, trickle flushes do not interfere with the proofs of Lemma 4.1, Lemma 4.2, or Lemma 4.3. Moreover, the Independent Levels Property, ensures that the prerequisite requirements for Lemma 4.1 are met by the dynamic lazy flushing algorithm.

Lemma 4.3 also bounds the size of buffers (both retired and non-retired) in levels containing only one node by $O(B)$; this ensures that the buffers containing only one node can be stored in memory using space $O(B^2)$. This means that, when analyzing the I/O complexity of the Dynamic Cup-Based B$^\varepsilon$-tree, we need only consider levels containing multiple nodes; by the Balanced-Tree Property, there are only $O(\log_B |T|)$ such levels with high probability in $N_{\max}$.

Since Lemma 4.3 bounds the size of each buffer by $O(B \log B)$, it follows that trickle flushes successfully eliminate retired buffers prior to their caretaker(s) being rebalanced. Thus every node is the caretaker for at most two retired nodes. This is especially important when the retired buffers are not stashed, since read operations accessing a node x must also examine all of the retired nodes for which x is a caretaker. Since each node x is a caretaker for at most two retired nodes, the Balanced-Tree Property ensures that the number of I/Os required by a read operation is at most $\log_B |T|$, with high probability in $N_{\max}$.

Lemma 4.2, bounds the probability of any cleanup occurring during a flushing round to $\frac{1}{\text{poly}(N_{\max})}$. Thus, with high probability in $N_{\max}$, there is only one universal flush in each level ℓ of the tree T. Since the trickle flushes corresponding with an universal flush incur only $O(1)$ additional I/Os at each level $\ell > \frac{1}{\varepsilon}+1$, and only $O(B^\varepsilon)$ additional I/Os at each level $\ell \leq \frac{1}{\varepsilon}+1$, it follows that with high probability in $N_{\max}$, the total number of I/Os required by a given flushing round is at most $O(\frac{B^\varepsilon}{\varepsilon}+q) = O(B^\varepsilon+q)$, where q is the number of I/Os incurred by universal flushes during the flushing round. Successive flushing rounds are separated by B insert/update/delete operations, meaning that if the work for a flushing round is spread out over the following B insert/update/delete operations, then the worst-case insert/update/delete costs becomes

$$O(\lceil (B^\varepsilon + q)/B^{1-\varepsilon} \rceil).$$

Since q is, with high probability in $N_{\max}$, at most $O(B^\varepsilon \cdot \log_B |T|)$, the i-th operation has, with high probability in $N_{\max}$, I/O cost at most

$$O(\lceil (\log_B N_i)/B^{1-\varepsilon} \rceil),$$

where N_i is the size of the tree after operation i.

To prove Proposition 5.1, it remains to analyze the amortized performance of insert/update/delete operations. So far we have shown that, with high probability in $N_{\max}$, each flushing round incurs at most $O(B^\varepsilon \log_B N_i)$ I/Os, where i is the index of any insert/update/delete performed during the flushing round. The total I/O cost of the the first n insert/update/delete operations is therefore at most

$$O\left(\sum_{i=1}^{n}\left(X_i + \frac{\log_B N_i}{B^{1-\varepsilon}}\right)\right),$$

where X_i is 0 for each operation i contained in a flushing round that incurs at most $O(B^\varepsilon \log_B N_i)$ I/Os, and X_i is $N_{\max}$ otherwise. Since each X_i is zero with high probability in $N_{\max}$, the sum $\sum_{i=1}^{n} X_i$ has expected value $n/\text{poly}(N_{\max})$. By Markov's inequality, the sum $\sum_{i=1}^{n} X_i$ is at most n with high probability in $N_{\max}$. The total I/O cost of the first n insert/update/delete operations is therefore at most

$$O\left(\sum_{i=1}^{n}\frac{\log_B N_i}{B^{1-\varepsilon}}\right),$$

with high probability in $N_{\max}$.

buffer.

5.2 Eliminating Inter-Level Dependencies In this section we introduce a node-rebalancing scheme satisfying the Independent Levels Property, the Breathing-Space Property, and (with high probability in $N_{\max}$) the Balanced-Tree Property. Combining this with Proposition 5.1 completes the proof of Theorem 5.1. We begin by designing a rebalancing scheme satisfying the Independent Levels Property and the Balanced-Tree Property, and then describe how to further satisfy the Breathing-Space Property at the end of the section.

Each node x_i maintains a simple summary sketch of the tree T_i rooted at x_i. If T is a tree of size in the range $[s_\ell/c^{0.2}, s_\ell \cdot c^{0.2}]$ (recall c is a large constant), the ***level-ℓ summary sketch*** $\mathcal{S}_\ell(T)$ satisfies the following properties:

- **Unit-Size.** With high probability in $N_{\max}$, $\mathcal{S}_\ell(T)$ consists of at most $O(B)$ machine words. (Moreover, this continues to be true for any T satisfying $|T| < s_\ell \cdot c^{0.2}$.)
- **Size Estimation.** $\mathcal{S}_\ell(T)$ provides an estimate s for $|T|$ (i.e., the logical size of the key set represented by T) satisfying $0.9|T| \leq s \leq 1.1|T|$ with high probability in $N_{\max}$. (Moreover, if $|T| > s_\ell/c^{0.2}$ or $T < s_\ell \cdot c^{0.2}$, then the size estimate s satisfies $s \geq 0.9 s_\ell/c^{0.2}$ and $s \leq 1.1 s_\ell \cdot c^{0.2}$, respectively.)
- **Median Estimation.** $\mathcal{S}_\ell(T)$ provides a message $r \in T$ such that, with high probability in $N_{\max}$, the rank of r in T is between $\frac{1}{4}|T|$ and $\frac{3}{4}|T|$.
- **Composability.** For two trees T_1, T_2, the sketch $\mathcal{S}_\ell(T_1 \cup T_2)$ can be computed directly from $\mathcal{S}_\ell(T_1)$ and $\mathcal{S}_\ell(T_2)$; moreover, given $\mathcal{S}_\ell(T_1 \cup T_2)$, and given the key boundaries of T_1 and T_2, one can directly compute $\mathcal{S}_\ell(T_1)$ and $\mathcal{S}_\ell(T_2)$.

For convenience, we will sometimes denote $\mathcal{S}_\ell(T)$ by $\mathcal{S}_\ell(x)$, where x is the root node of the tree T.

Before describing how to construct summary sketches, we explain and analyze their usage in the data structure. Node joins and splits at each level ℓ are performed based on the estimated size of nodes given by the sketches $\mathcal{S}_\ell(T_i)$. Specifically, a node is merged with one of its siblings whenever its estimated size is smaller than s_ℓ, and is split whenever its estimated size is larger than $10 s_\ell$.

When a node x_i is split, the median estimate r given by $\mathcal{S}_\ell(T_i)$ is used as the boundary on which the split occurs. Note that r *might not* be used as a boundary at level $\ell - 1$. This means that our B$^\varepsilon$-tree has a structure similar to that of a skip list, in that adjacent nodes at level ℓ may have a child in common at level $\ell - 1$.

Creating and eliminating new root nodes introduces a subtle issue. A natural approach would be to create a new root node whenever the old root splits, and eliminate a root node whenever its number of children becomes one. However, the creation or destruction of a root node at a level h would then be a function of random bits used in sketches $\mathcal{S}_{h-1}$, which would violate the Independent Levels Property. To resolve this, we simply make each level $\ell \in 1, \ldots, B$ *always* contain at least one node. All levels that contain only a single node fit in memory (with high probability $N_{\max}$). Nodes are merged and split according to summary-sketch-based weight balancing, except that if a level contains only one node then that node can never be merged (since it has no siblings).

Establishing the independent levels and balanced-tree properties. The Independent Levels Property follows from the fact that the sketches S_ℓ use different random bits at each level ℓ.

To establish the Balanced-Tree Property (with high probability in $N_{\max}$), we must be careful about the fact that the random bits determining the sketches $\mathcal{S}_\ell$ also implicitly determine what interval each subtree T_i consists of in level ℓ. That is, the random bits used to determine $\mathcal{S}_\ell$ help determine the partition $P = (T_1, T_2, \ldots, T_k)$ of the records in levels $1, 2, \ldots, \ell$ (where each T_i corresponds with an interval of keys for which a single node at level ℓ is responsible). This means that the partition P could potentially be adversarial against the sketch $\mathcal{S}_\ell$. However, since the set of records $S = T_1 \cup T_2 \cup \cdots \cup T_k$ after a given universal flush at level $\ell + 1$ is determined entirely by randomness in levels $\ell+1, \ell+2, \ldots$ of the tree (and by the operations on the tree), and since for a given set S there are only $O(|S|^2) \leq N_{\max}^2$ options for each T_i, it follows that with high probability in $N_{\max}$, the sketches $\mathcal{S}_\ell(T_i)$ perform correct size and median estimation even for a worst-case choice of the partition P, and satisfy the Unit-Size Requirement.

The Unit-Size Requirement ensures that the sketch of each subtree T_i can be stored in the root node x_i of the subtree using a constant number of blocks (with high probability in $N_{\max}$). The composability of sketches makes it so that when a node split or join occurs, the corresponding sketches can also be split or joined. By the accuracy of size- and median- estimation, we get that with high probability in $N_{\max}$, the Balanced-Tree Property holds after each operation.

Constructing summary sketches. A natural approach to constructing $\mathcal{S}_\ell(T)$ is to randomly sample $B \geq \Omega(\log N_{\max})$ keys from T. Suppose we have access to a fully independent hash function h that maps each key r to 0 with probability p and to a non-zero value with probability $1 - p$, where p is selected so that $p \cdot s_\ell = B \geq c \log N_{\max}$. (Note, however, that our final construction cannot use such a hash function, since the description bits for h would not fit in memory.)

Then $\mathcal{S}_\ell(T)$ could store the set of keys $r \in T$ for which $h(r) = 0$. Whenever an insert for a key r is placed into the tree T, the sketch $\mathcal{S}_\ell(T)$ is updated to include r if $h(r) = 0$; whenever a delete for a key r is placed into the tree T,

the sketch $\mathcal{S}_\ell(T)$ is updated to exclude r. Assuming c is a sufficiently large constant, and that $|T| \in \Theta(s)$, then with high probability in $N_{\max}$,

$$0.9 \cdot |T| \leq \frac{1}{p}|\mathcal{S}_\ell(T)| \leq 1.1 \cdot |T|,$$

allowing $\mathcal{S}_\ell(T)$ to be accurately used for size estimation. Additionally, the median of $\mathcal{S}_\ell(T)$ is, with high probability in $N_{\max}$, of rank between $|T|/4$ and $3|T|/4$ in T. This is because, with high probability in $N_{\max}$, no more than $1.1 \cdot \frac{p}{4}|T|$ of the keys in S are among the bottom quartile in T (in terms of ranking), and similarly, no more than $1.1 \cdot \frac{p}{4}|T|$ of the keys in S are among the top quartile in T.

In order to remove the requirement that h be fully independent, we replace h with a collection of low-independence hash functions as follows. Let $h_1, \ldots, h_B$ be B-wise independent hash functions, with each h_k mapping keys r to 0 with probability q and to non-zero with probability $1 - q$, where $q \cdot s_\ell = c$.

To compute $\mathcal{S}_\ell(T)$ for a tree T, define $S_1, \ldots, S_B$ so that $S_i = \{r \in T \mid h_i(r) = 0\}$, and define the sketch $\mathcal{S}_\ell(T) = (S_1, \ldots, S_B)$. Composability is immediate from the definition of the sketch. The following lemma demonstrates how to use $\mathcal{S}_\ell(T)$ for size estimation.

LEMMA 5.1. *Fix a tree T, and define u_T to be the median of values $|S_1|/q, \ldots, |S_k|/q$. Then with high probability in $N_{\max}$, if $|T| \geq c^{-0.2}s_\ell$ then $0.9|T| \leq u_T \leq 1.1|T|$; and if $|T| \leq c^{-0.2}s_\ell$ then then $u_T \leq 1.1 \cdot c^{-0.2}s_\ell$.*

Proof. Suppose that $|T| \geq c^{-0.2}s_\ell$, and thus $|T| \cdot q \geq c^{0.8}$. Since $|S_i|$ is the sum of $|T|$ B-wise (and thus pairwise) independent 0-1 random variables, each with mean $q \leq 1/2$, the variance of $|S_i|$ is given by $|T| \cdot q \cdot (1 - q) \leq |T| \cdot q$. The variance of $|S_i|/q$ is therefore at most $|T|/q$. By Chebyshev's inequality,

$$\Pr\left[\left||S_i|/q - |T|\right| > \delta|T|\right] \leq \frac{|T|/q}{\delta^2|T|^2} = \frac{1}{\delta^2|T| \cdot q} \leq \frac{1}{\delta^2 c^{0.8}},$$

for $\delta > 0$. Plugging in $\delta = 0.1$, and using that c is a large constant, yields

$$\Pr\left[\left||S_i|/q - |T|\right| > 0.1|T|\right] \leq \frac{1}{16}. \tag{5.2}$$

By a Chernoff bound, with high probability in $N_{\max}$, at least $2/3$ of the $B \geq N_{\max}$ values $|S_i|/q$ satisfy $0.9|T| \leq |S_i|/q \leq 1.1|T|$, and thus the median u_T also satisfies $0.9|T| \leq u_T \leq 1.1|T|$.

Finally, we consider the case of $|T| \leq c^{-0.2}s_\ell$. If T' is any tree of size $c^{-0.2}s_\ell$ satisfying $T \subseteq T'$ (i.e., T''s record set includes T's record set), then the $u_T \leq u_{T'}$. By the analysis above, $u_{T'} \leq 1.1 \cdot c^{-0.2}s_\ell$ with high probability in $N_{\max}$, completing the proof. ∎

The next lemma demonstrates how to use $\mathcal{S}_\ell(T)$ in order to achieve median estimation.

LEMMA 5.2. *Fix a tree T satisfying $|T| \geq c^{-0.2}s_\ell$, and define v_T to be the median of $m(S_1), \ldots, m(S_k)$, where $m(S_i)$ is the median of the elements of S_i. If S_i is empty, then $m(S_i)$ is omitted from the set over which the median v_T is taken.*

Then with high probability in $N_{\max}$, the key v_T has rank between $|T|/4$ and $3|T|/4$ in T.

Proof. By the proof of Lemma 5.1, each S_i satisfies $0.9|T| \leq |S_i|/q \leq 1.1|T|$ with probability at least $15/16$.

Let X_i denote the set of keys x in S_i that have rank $|T|/4$ or smaller in T, and let Y_i denote the set of keys $y \in S_i$ that have rank greater than $4|T|/4$ in T. Then the variance of each of $|X_i|/q$ and $|Y_i|/q$ is at most $O(|T|/q)$, by the same analysis as in Lemma 5.1. Using that c is a large constant, it follows by Chebyshev's inequality that,

$$\Pr\left[\left||X_i|/q - \frac{1}{4} \cdot |T|\right| > 0.1|T|\right] \leq \frac{1}{16},$$
$$\Pr\left[\left||Y_i|/q - \frac{1}{4} \cdot |T|\right| > 0.1|T|\right] \leq \frac{1}{16}.$$

Recalling that S_i satisfies $q \cdot 0.9|T| \leq |S_i| \leq q \cdot 1.1|T|$ with probability at least $15/16$, it follows that with probability at least $13/16$ the median element $m(S_i)$ of S_i is in neither X_i nor Y_i.

By a Chernoff bound, with high probability in $N_{\max}$, at least $2/3$ of the $m(S_i)$'s have rank between $|T|/4$ and $3|T|/4$ in T. It follows that the median v_T of the medians $m(S_i)$ also has rank between $|T|/4$ and $3|T|/4$ in T, as desired. ∎

Finally, we complete the analysis of the summary sketch $S_\ell(T)$, by proving the unit-size property.

LEMMA 5.3. *Fix a tree T satisfying $|T| \leq c^{0.2}s_\ell$. Then with high probability in $N_{\max}$, the size of the sketch $S_\ell(T)$ is at most $O(B)$ machine words.*

Proof. Here we finally use the B-independence of $h_1, \ldots, h_B$. (Indeed, for Chebyshev's inequality we only needed 2-independence.) By Theorem 2 of [41], for c' a sufficiently large constant, each S_i satisfies

$$\Pr[|S_i| \geq c'm] \leq 2^{-m},$$

for $m \geq 1$. Since the sizes of the $|S_i|$'s are independent, it follows that $\sum_i |S_i|$ is a sum of independent geometrically-bounded random variables. In particular, the probability that $\sum_i |S_i|$ exceeds its mean by $c' \cdot m$ for some $m \geq 1$ is at most the probability that $m - 1$ fair coin flips together yield less than B total heads. Since $B = \Omega(\log N_{\max})$, it follows that with high probability in $N_{\max}$, $\sum_i |S_i| \leq O(\log N_{\max}) \leq O(B)$, as desired. ∎

Bounding the metadata size. For each level $\ell \in [B]$, we must store the random bits for B hash functions. Each of the hash functions h_k at level ℓ must be B-wise independent and map keys to 0 with probability c/s_ℓ.

We may assume w.l.o.g. that c and s_ℓ are powers of two, meaning that there exists a finite field of size s_ℓ/c. Thus if d is the maximum number of bits of a record, then one can select h_k as a random degree-$(B-1)$ polynomial over the finite field $\mathcal{F}_{\max(2^d, s_\ell/c)}$. The number of random bits needed to store h_k is therefore at most $\max(d, s_\ell)B$.

When $s_\ell \leq O(d)$, the random bits for each h_k fit in $O(1)$ blocks on disk, since each block can store $\Theta(B)$ d-bit records. On the other hand, if $\log s_\ell \geq 10d$, then the maximum size of the tree $N_{\max}$ must satisfy $N_{\max} \ll s_\ell$, and thus we need not actually construct the sketch $\mathcal{S}_\ell$. (Instead we can simply force level ℓ to consist of a single node.)

Thus each of the B hash functions h_k at each of the B levels requires only $O(1)$ blocks of random bits. The total non-node-specific metadata required by the rebalancing mechanism is $O(B^2)$ blocks, as required by the Balanced-Tree Property.

Adding the Breathing Space Property. Finally, we modify our rebalancing mechanism in order to fulfill the Breathing-Space Property, which we restate below.

- **The Breathing-Space Property:** Say a node x is ***touched*** whenever an universal flush occurs either in x or in one of x's sibling nodes in level ℓ. (Note that the sibling nodes of x in level ℓ need not share a parent with x.) Whenever a new node x is created at level ℓ, either via a node merge or a node split, the new node x is touched at least $\Omega(s_\ell/B)$ times.

We can enforce the breathing-space property with a small modification to our weight-balancing scheme. Call a node x ***rested*** if, since the creation of x the node x has received at least $\frac{1}{10} \cdot s_\ell$ new elements in its buffer. By Lemma 4.3, the buffer-size of a node x is at most $O(B \log M) \leq O(B \log B)$. It follows that each rested node x has been flushed out of at least

$$\frac{1}{10} \cdot s_\ell/B_\ell - O(\log B)$$

times, which by the assumption that B^ε is at least a large constant multiple of $\log B$ (i.e., that B is at least a sufficiently large constant), is at least $\frac{1}{20} \cdot s_\ell/B_\ell$.

Call a node x ***tangentially rested*** if, since the creation of node x, at least one of node x's current siblings (i.e., we do not count siblings that have since been rebalanced) has received at least $\frac{1}{10} \cdot s_\ell$ new elements in its buffer. Note that a tangentially rested node x can potentially become no-longer tangentially rested when one of its siblings is rebalanced. Again applying Lemma 4.3, the number of flushes applied to x's neighbors since x's creation must be at least $\frac{1}{20} \cdot s_\ell/B_\ell$ for any tangentially rested node x.

We modify our weight-balancing scheme as follows. Until a node x becomes either rested or tangentially rested, the node x is deemed ***frozen***, meaning that regardless of value of x's summary sketch, the node x is not permitted to be merged or split. Whenever a non-frozen node y wishes to perform a node-merge, the node waits for at least one of its neighbors to become non-frozen, and then performs a merge with that neighbor (although node y could potentially *stop* waiting on its neighbors if y's sketch $\mathcal{S}_\ell(y)$ stops indicating the need for a node-merge). If a node x becomes non-frozen, and at least one of node x's neighbors are waiting on x for a merge, then node x is merged with any neighbors waiting on x, and then if necessary node x is then immediately split. Whenever a node is non-frozen, wishes to perform a split, and has no neighbors waiting on it for a merge, the node can x performs the split without further delay.

The fact that frozen nodes are never rebalanced ensures the Breathing-Space Property. Moreover, the modifications to the algorithm easily preserve the Independent Levels Property. To complete the analysis of the rebalancing mechanism, we must show that the Balanced-Tree Property continues the hold. Specifically, we must prove that, at any time, each node x in each level ℓ is of size $\Theta(s_\ell)$ with high probability in $N_{\max}$, unless there is only one node at that level.

LEMMA 5.4. *Using the rebalancing mechanism described above, the following property holds: at any point in time, each node x in each level ℓ is of size $\Theta(s_\ell)$ with high probability in $N_{\max}$, unless there is only one node at that level.*

Proof. We consider three cases. The first case is that a node x has seeked rebalancing (i.e., $\mathcal{S}_\ell(x)$ estimates x's size to be either less than $s_\ell/2$ or estimates x's size to be at least $10s_\ell$) after the arrivals of each of the most recent $\frac{1}{5} \cdot s_\ell/B_\ell$ elements to have arrived in x's buffer; and furthermore, $\mathcal{S}_\ell(x)$ has provided correct size estimates after each of those arrivals. Then node x must have been unfrozen during all of the most recent $\frac{1}{10} \cdot s_\ell/B_\ell$ arrivals in x's buffer. Moreover, during those most recent $\frac{1}{10} \cdot s_\ell/B_\ell$ arrivals, at least one of node x's siblings must have become unfrozen, meaning that node x will have successfully been rebalanced, a contradiction.

As a second case, suppose $\mathcal{S}_\ell(x)$ has provided correct size estimates after all of the most recent $\frac{1}{5} \cdot s_\ell/B_\ell$ arrivals into node x's buffer, and that node x has *not* required rebalancing after at least one of those arrivals. Then node x must currently be of size between $s_\ell/(4B_\ell)$ and $11s_\ell/4B_\ell$, ensuring that node x is size $\Theta(s_\ell)$.

Finally, the third case is that $\mathcal{S}_\ell(x)$ has not provided correct size estimates after all of the most recent $\frac{1}{5} \cdot s_\ell/B_\ell$ arrivals into node x's buffer. We show that with high probability in $N_{\max}$, this case does not occur for any node x

in the tree T. For a given level ℓ, the arrivals of messages into level ℓ are determined independently of the random bits used to compute sketches $\mathcal{S}_\ell$. In the current state of tree T, there are $O(n^2)$ options for the interval $[a, b]$ that a node x can consist of (i.e., the node x has minimum key a and maximum key b). Consider the arrivals into level ℓ of messages in the interval $[a, b]$, and let $W_i(a, b)$ denote the state of tree T after the i-th most recent such arrival. Let $P_i(a, b)$ denote the set of records at level ℓ (or below) in the interval $[a, b]$ in state $W_i(a, b)$. Since $P_i(a, b)$ is independent of the random bits used to compute $\mathcal{S}_\ell$, the sketch $\mathcal{S}_\ell(P_i(a, b))$ has high probability in $N_{\max}$ of correctly estimating the size of $P_i(a, b)$. Unioning over $i \in [0, \ldots, \frac{1}{5} \cdot s_\ell/B_\ell]$, and over all $N_{\max}^2$ options for a and b, it follows that, with high probability in $N_{\max}$, $\mathcal{S}_\ell$ performs correct size estimation on all such $P_i(a, b)$. This, in turn, establishes that the third case occurs with low probability in $N_{\max}$. □

6 Related Work

COLAs [8], write-optimized skip-lists [10], and B$^\varepsilon$-trees [9, 14] are all on the optimal search-insert trade-off curve [14] and are easily seen to have roughly equivalent structures.

Write-optimized skip lists are just B$^\varepsilon$-trees with successor pointers between nodes on the same level and a randomized rebalancing scheme. The successor pointers have no impact on flushing. In fact, it's entirely plausible and useful to add sibling pointers to B$^\varepsilon$-trees, so the only difference is the rebalancing scheme.

COLAs can be transformed into B$^\varepsilon$-trees by simply storing each level as a sequence of chunks, where block boundaries are placed every $\Theta(B^\varepsilon)$ ghost elements. Breaking each level into chunks is useful for deamortizing flushing, since it makes it easy to perform flushing locally, i.e. the flushing policy can choose to flush from one chunk to another, rather than flushing an entire level. Thus the ghost pointers become B$^\varepsilon$-tree pivots and the chunks are B$^\varepsilon$-tree nodes.

Although LSMs are typically described quite differently from B$^\varepsilon$-trees, they are often much closer in practice. For example, LSMs were originally described with each level consisting of a single sorted array but, for obvious practical reasons, many implementations break each level into multiple sub-arrays, which correspond roughly to nodes in a B$^\varepsilon$-tree. This already opens the door to localized flushing, rather than level-by-level flushing. The other main difference is that many LSMs do not perform fractional cascading or maintain any indexing information from one level to the next, so that queries must start from scratch at each level. LSM implementers typically assume all this indexing information can fit in RAM, so that fractional cascading is not necessary. Regardless, this design choice doesn't interact with flushing.

One important difference is that many LSMs do not preserve chunk boundaries as they move down the tree. In a B$^\varepsilon$-tree, if a key k is the boundary between two nodes at depth i, then no node below level i will span k. Many LSMs don't maintain this invariant. As a result, a single chunk at a higher level may span the ranges covered by an unbounded number of chunks on the next level down. This can stymie any attempt to deamortize flushing, since flushing a single chunk at one level may require touching the entire level below it. However, it is easy to add this invariant to an LSM implementation. Once this is done, each chunk can include the indexing information of its subordinate chunks on the next level down, and now we have essentially the structure of a B$^\varepsilon$-tree.

References

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

[2] A. Amir, M. Farach, R. M. Idury, J. A. L. Poutré, and A. A. Schäffer. Improved dynamic dictionary matching. *Inf. Comput.*, 119(2):258–282, 1995.

[3] A. Amir, G. Franceschini, R. Grossi, T. Kopelowitz, M. Lewenstein, and N. Lewenstein. Managing unbounded-length keys in comparison-driven data structures with applications to online indexing. *SIAM Journal on Computing*, 43(4):1396–1416, 2014.

[4] Apache. Accumulo. `http://accumulo.apache.org`, 2015.

[5] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

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

[7] M. A. Bender, A. Conway, M. Farach-Colton, W. Jannen, Y. Jiao, R. Johnson, E. Knorr, S. McAllister, N. Mukherjee, P. Pandey, D. E. Porter, J. Yuan, and Y. Zhan. Small refinements to the DAM can have big consequences for data-structure design. In *Proc. 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 265–274, June 2019.

[8] 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 Parallel Algorithms and Architectures (SPAA)*, pages 81–92, 2007.

[9] 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.

[10] M. A. Bender, M. Farach-Colton, R. Johnson, S. Mauras, T. Mayer, C. Phillips, and H. Xu. Write-optimized skip lists. In *Proc. 36th ACM Symposium on Principles of Database Systems (PODS)*, pages 69–78, May 2017.

[11] M. A. Bender, M. Farach-Colton, and W. Kuszmaul. Achieving optimal backlog in multi-processor cup games. In *Proc. 51st Annual ACM Symposium on the Theory of Computing (STOC)*, pages 1148–1157, June 2019.

[12] E. Bortnikov, A. Braginsky, E. Hillel, I. Keidar, and G. Sheffi. Accordion: Better memory organization for LSM key-value stores. *VLDB*, 2018.

[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] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

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

[17] N. Dayan and S. Idreos. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proc. 2018 International Conference on Management of Data (SIGMOD)*, pages 505–520, 2018.

[18] P. Dietz and R. Raman. Persistence, amortization and randomization. 1991.

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

[20] J. Esmet, M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. The TokuFS streaming file system. In *Proc. 4th USENIX Workshop on Hot Topics in Storage (HotStorage)*, MA, USA, June 2012.

[21] I. FaceBook. RocksDB Compaction.

[22] J. Fischer and P. Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 160–171, 2015.

[23] T. A. S. Foundation. Cassandra compaction.

[24] M. T. Goodrich and P. Pszona. Streamed graph drawing and the file maintenance problem. In *International Symposium on Graph Drawing (GD)*, pages 256–267. Springer, 2013.

[25] Google, Inc. LevelDB: A fast and lightweight key/value database library by Google. `https://github.com/google/leveldb`.

[26] N. Har'El. Scylla's compaction strategies series: Space amplification in size-tiered compaction. `https://www.scylladb.com/2018/01/17/compaction-series-space-amplification/`.

[27] N. Har'El. Scylla's compaction strategies series: Write amplification in leveled compaction. `https://www.scylladb.com/2018/01/31/compaction-series-leveled-compaction/`.

[28] Apache HBase. `https://hbase.apache.org/`.

[29] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. Betrfs: A right-optimized write-optimized file system. In *Proc. 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 301–315, 2015.

[30] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. Betrfs: Write-optimization in a kernel file system. *Transactions on Storage—Special Issue on USENIX FAST 2015*, 11(4):18:1–18:29, October 2015.

[31] Z. Kasheff and B. Kuszmaul. Personal communication. 2015.

[32] T. Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *Proc. 53rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 283–292, 2012.

[33] W. Kuszmaul. Achieving optimal backlog in the vanilla multi-processor cup game. In *Proc. 31-st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2020.

[34] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, Apr. 2010.

[35] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[36] C. Luo and M. J. Carey. LSM-based storage techniques: A survey. *CoRR*, abs/1812.07527, 2018.

[37] C. W. Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 618–627, 2003.

[38] 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.

[39] K. Ren and G. A. Gibson. TABLEFS: Enhancing metadata efficiency in the local file system. In *USENIX Annual Technical Conference*, pages 145–156, 2013.

[40] RocksDB. `https://rocksdb.org/`.

[41] J. P. Schmidt, A. Siegel, and A. Srinivasan. Chernoff–hoeffding bounds for applications with limited independence. *SIAM Journal on Discrete Mathematics*, 8(2):223–250, 1995.

[42] ScyllaDB. Compaction. `https://docs.scylladb.com/kb/compaction/`.

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

[44] Tokutek Inc. ft-index. `https://github.com/Tokutek/ft-index`. Accessed: 2018-01-28.

[45] Tokutek, Inc. TokuDB: MySQL Performance, MariaDB Performance . `http://www.tokutek.com/products/tokudb-for-mysql/`.

[46] Tokutek, Inc. TokuMX—MongoDB Performance Engine. `http://www.tokutek.com/products/tokumx-for-mongodb/`.

[47] J. Yuan, Y. Zhan, W. Jannen, P. Pandey, A. Akshintala, K. Chandnani, P. Deo, Z. Kasheff, L. Walsh, M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. Optimizing every operation in a write-optimized file system. In *Proc. 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, February 2016.