What Does Dynamic Optimality Mean in External Memory?[*]

Michael A. Bender[†]
Stony Brook University, Stony Brook, NY, USA
bender@cs.stonybrook.edu

Martín Farach-Colton[‡]
Rutgers University, New Brunswick, NJ, USA
martin@farach-colton.com

William Kuszmaul[§]
MIT, Cambridge, MA, USA
kuszmaul@mit.edu

April 22, 2022

Abstract

A data structure A is said to be dynamically optimal over a class of data structures $\mathcal{C}$ if A is constant-competitive with every data structure $C \in \mathcal{C}$. Much of the research on binary search trees in the past forty years has focused on studying dynamic optimality over the class of binary search trees that are modified via rotations (and indeed, the question of whether splay trees are dynamically optimal has gained notoriety as the so-called dynamic-optimality conjecture). Recently, researchers have extended this to consider dynamic optimality over certain classes of external-memory search trees. In particular, Demaine, Iacono, Koumoutsos, and Langerman propose a class of external-memory trees that support a notion of tree rotations, and then give an elegant data structure, called the Belga B-tree, that is within an $O(\log \log N)$-factor of being dynamically optimal over this class.

In this paper, we revisit the question of how dynamic optimality should be defined in external memory. A defining characteristic of external-memory data structures is that there is a stark asymmetry between queries and inserts/updates/deletes: by making the former slightly asymptotically slower, one can make the latter significantly asymptotically faster (even allowing for operations with sub-constant amortized I/Os). This asymmetry makes it so that rotation-based search trees are not optimal (or even close to optimal) in insert/update/delete-heavy external-memory workloads. To study dynamic optimality for such workloads, one must consider a different class of data structures.

The natural class of data structures to consider are what we call buffered-propagation trees. Such trees can adapt dynamically to the locality properties of an input sequence in order to optimize the interactions between different inserts/updates/deletes and queries. We also present a new form of beyond-worst-case analysis that allows for us to formally study a continuum between static and dynamic optimality. Finally, we give a novel data structure, called the Jεllo Tree, that is statically optimal and that achieves dynamic optimality for a large natural class of inputs defined by our beyond-worst-case analysis.

1   Introduction

Static and dynamic optimality in internal memory   Since the early 1960s, many balanced binary trees have been developed with worst-case $O(\log N)$ time per operation [1, 6, 22], where N is the number of elements in the tree. In such trees, the cost of any particular operation can be much smaller, even $O(1)$, if the element being queried is stored near the root of the tree. This means that a search tree can potentially achieve $o(\log N)$ time per operation on workloads that exhibit locality properties.

Since the 1980s, considerable effort has been devoted to designing ***distribution-sensitive binary search trees*** that perform workload-specific optimizations. Broadly speaking, there are two approaches to analyzing distribution-sensitive search trees. The first approach is to bound the performance based on some property of the input sequence [5, 13, 17–19, 25, 26, 33, 34], e.g., the sequential access bound [34], the working set bound [26, 33], the weighted dynamic finger bound [13, 18, 19], and the unified bound [5, 26]. The second approach is competitive analysis, where one must select a class of data structures $\mathcal{C}$, such as static binary trees or binary trees that are modified via rotations, and then design a single data structure A (not necessarily from $\mathcal{C}$) that is competitive with any data structure in $\mathcal{C}$. If the members of $\mathcal{C}$ are static, a $O(1)$-competitive algorithm[1] is said to be ***statically optimal against $\mathcal{C}$*** and if they are dynamic, a $O(1)$-competitive algorithm is said to be ***dynamically optimal against $\mathcal{C}$***.

Splay trees are famously statically optimal against the class of binary trees [33]. On the other hand, whether dynamic optimality can be achieved against the class of binary-trees-with-rotations remains one of most elusive problems in the field of data structures (see [27] for a survey). The Tango Tree [23] is known to be within a factor of $O(\log \log N)$ of dynamic optimality. The splay tree [32] is widely believed to achieve dynamic optimality, but it remains open whether the structure is even $o(\log N)$-competitive.

Research on dynamic optimality against internal-memory search trees has historically considered sequences of operations consisting exclusively of queries. We emphasize that this is *not* a limitation of past work—indeed, it turns out that queries and inserts/updates/deletes are sufficiently similar to one another that the queries-only assumption is without loss of generality. As we will see later, this equivalence does not hold in external memory.

Static and dynamic optimality in external memory   Search trees are every bit as ubiquitous in external memory as they are in internal memory—e.g., they are used prominently in file systems [28, 37], databases [7, 20], and key-value stores [12, 21, 30]. The principle difference with internal memory is that disks are accessed in blocks of some (typically large) size B.[2] External-memory search trees are analyzed in the ***Disk-Access Model*** [2], where the goal is to minimize the number of block accesses (also known as ***I/Os***).

B-trees [7, 20] are balanced search trees optimized for external memory, which means they have fanout $\Theta(B)$, and hence a worst case of $O(\log_B N)$ I/Os per operation. Again, the cost of any particular operation can be much smaller if the element being queried is stored near the root. This raises a natural question: what can be said about static and dynamic optimality for external-memory search trees? We remark that most of the work on this question has focused on dynamic optimality and treated static optimality implicitly.

The notion of dynamic optimality in external-memory search trees is less well understood than in internal memory. Part of the reason for this is the difficulty of identifying the class of data structures over which dynamic optimality should be defined, and in particular identifying the mechanism by which elements can move vertically in the tree.

Early work focused on a skip-list-like mechanism where keys can move towards the root in the tree by becoming a pivot and splitting some node in two [14, 31]. We call this class of data structures ***merge-split trees***. Bose et al. [14] gave a data structure that achieves dynamic optimality in this model and showed that the performance of their data structure is determined by the working-set bound [26, 33].

Merge-split trees are limited in their ability to exploit the locality of a workload. For example, on the sequential workload in which $1, 2, \ldots, N$ are accessed round robin, merge-split trees incur amortized cost

---

[1]Here, we can see an example where it is especially important that A not have to be a member of $\mathcal{C}$. In particular, if we wish to construct an A that is $O(1)$-competitive with any (omnisciently constructed) static $\mathcal{C}$, then we must allow for A to adapt dynamically over time.

[2]As a convention, B is measured in terms of the number of machine words that fit in a block.

$\Omega(\log_B N)$ per operation. In contrast, even in-memory binary trees implemented using rotations can achieve the **_sequential access bound_** [34] on the same workload, that is, an amortized $O(1)$ I/Os. The main difference between merge-split trees and binary trees with rotations is that merge-split trees move individual elements up and down the tree, but they do not move entire subtrees together (as is the case for rotations).

Recently, Demaine et al. [24] introduced a different class of dynamic trees that support "rotations" similar to those in in-memory binary search trees. The authors study dynamic optimality over this class of data structures and introduce the Belga B-tree, which they prove is $O(\log \log N)$-competitive against any rotation-based search tree.

The work on external-memory dynamic optimality so far has focused on exploiting the underlying locality properties of the workload in order to optimize queries. The tradeoff being explored is the decision of which keys are stored near the root of the tree and which keys are stored further down.

This paper: optimizing the asymmetry of external-memory operations   One of the remarkable (and perhaps unexpected) differences between search trees in internal and external memory, however, is that in external memory, inserts/updates/deletes can be implemented to have amortized performance asymptotically faster than that of queries. While the worst-case cost of queries is logarithmic, inserts/updates/deletes can take an amortized subconstant number of I/Os [8–10, 15, 16, 29].

An important consequence of this asymmetry is that there are many input sequences for which the positioning of different keys in the tree is _not_ the dominant factor controlling performance. To study dynamic optimality for such sequences, we must consider a class of algorithms that can optimize the cost of queries vs insertion/deletions/updates.

The key technique for such optimizations is **_buffered propagation_** in which one propagates insert/update/delete operations down the tree in buffered batches. This allows for a single I/O to make progress on many insert/update/delete operations simultaneously, so that the amortized cost of such operations is small. We emphasize that, on insert/update/delete-heavy workloads, even standard trees that _non-adaptively_ use buffered propagation, such as the B^ε-trees [3, 8, 10, 15], can be asymptotically faster than the best possible adaptive rotation-based search trees.

Buffered propagation comes with a tradeoff curve: we can make inserts/updates/deletes faster (up to a factor of $O(B)$) at the cost of making queries slower (up to a factor of $O(\log B)$). This means that there is an opportunity to adapt dynamically to a sequence of operations, both by adjusting the amount of buffered propagation over time, and by using _different amounts of buffered propagation in different parts of the tree_. There is also an opportunity to adapt the choice of pivots used by each internal node in the tree in order to strategically split collections of operations in a way that sends queries in one direction and inserts/updates/deletes in another.

All of these tradeoffs can be formally captured with a class of data structures that we call **_buffered-propagation trees_**—the problem of optimizing the tradeoffs between queries and non-queries in an external-memory search tree corresponds to the problem of achieving dynamic optimality against buffered-propagation trees.

We remark that even _static optimality_ against buffered propagation trees is an interesting question. That is, given a workload of query operations and update operations (which change values associated with keys, so the set of keys does not change over time), can one construct a data structure that is competitive with the optimal statically-structured buffered propagation tree? Even though the buffered propagation tree has a static structure, it can still strategically select the pivots and the amount of buffered propagation at each internal node in order to optimize for spatial locality between operations. Even if we are given the operations up front (in an offline manner), it is not immediately clear how one should go about constructing the optimal static buffered propagation tree.

A continuum between static and dynamic optimality   Achieving full dynamic optimality against any sophisticated class of search trees (whether it be internal-memory rotation-based trees or external-memory buffered-propagation trees) is a difficult problem to get traction on: even small changes to a tree can have significant impact on asymptotic performance, so an omniscient adversary can potentially perform rapid

modifications to the data structure in order to adapt to the workload at a very fine-grained level. In addition to considering the question of how to model dynamic optimality in external memory, a second contribution of this paper is to revisit the question of how we should perform beyond-worst-case analysis within that model, in order to characterize how "close" a given data structure is to achieving dynamic optimality.

One insight is that, in practice, it is natural to expect that the properties of an input sequence may evolve slowly over time, meaning that the (offline) optimal dynamic buffered propagation tree will also evolve slowly. We capture this property formally by declaring a sequence of operations to be **K-smooth** if there exists an optimal dynamic buffered propagation tree T for the sequence such that only a $1/K$-fraction of T's I/Os are spent restructuring the tree.

We propose a natural form of beyond-worst-case analysis: rather than trying to achieve full dynamic optimality, can we achieve dynamic optimality for the class of K-smooth inputs (and for some reasonably small K)? Instead of thinking of this as a restriction on input sequences, one can also think of it as a type of resource augmentation. Can we design a data structure that is $O(1)$-competitive with any **K-speed-limited buffered propagation tree**, that is any buffered propagation tree that is limited to spend at most a $1/K$-fraction of its I/Os on modifying the tree? Note that any data structure that is $O(1)$-competitive against K-speed-limited buffered propagation trees is guaranteed to be $O(1)$-competitive on all K-smooth input sequences—therefore the problem of achieving competitive guarantees against K-speed-limited adversaries subsumes the problem of achieving optimality on K-smooth inputs.

The study of K-speed-limited adversaries offers an intriguing continuum between static and dynamic optimality. Optimality against ∞-speed-limited adversaries is equivalent to static optimality, and optimality against 1-speed-limited adversaries is equivalent to full dynamic optimality. The smaller a K that we can achieve optimality against, the closer we are to achieving true dynamic optimality. We remark that this same continuum would also be interesting to study for internal-memory search trees, and we leave this direction of work as an open problem.

Achieving dynamic optimality against a speed-limited adversary   The third contribution of the paper is a new data structure that we call the Jεllo Tree. The Jεllo Tree is statically optimal, meaning that on any workload of updates and queries, the tree is constant competitive with any static buffered propagation tree. The Jεllo Tree is also dynamically optimal against any sufficiently speed-limited buffered propagation tree.

Our main theorem is that, for any $\delta \in (0, 1) \cap \Omega(\log \log N / \log B)$, we can build a J$\varepsilon$llo Tree that is $O(1/\delta)$-competitive with any $B^{3\delta}$-speed-limited buffered propagation tree. The construction and analysis of the Jεllo Tree is the main technical result of the paper.

In addition to requiring that the adversary is speed limited, our competitive analysis assumes a $B^{3\delta}$-factor of resource augmentation on cache size, meaning that we compete an adversary whose cache is a $B^{3\delta}$-factor smaller than ours. We also present a version of the analysis that incurs a small additive overhead in exchange for eliminating the cache-size resource augmentation.

Paper outline   In Section 2, we formally define buffered propagation trees and speed-limited adversaries, and we state our main results. Then in Section 3 we give a sketch of the Jεllo Tree's design and analysis. The full design and analysis appear in the extended version of the paper [11].

2   Defining the Class of Speed-Limited Buffered Propagation Trees

In this section, we formally define buffered propagation trees (and speed-limited buffered propagation trees). We then define the adversary Speed-Limited OPT (or OPT for short) against which we will analyze the Jεllo Tree. And finally we use these definitions to formally state the main result of the paper.

Both buffered propagation trees and the Jεllo Tree live in the Disk-Access Model [2]. In particular, the computer has a cache of size M machine words, and has access to an (unbounded-size) external memory consisting of blocks of some size B machine words. An algorithm can read/write a block from external

memory to cache at the cost of one **block access** (or **I/O**), and time is measured as the total number of I/Os incurred by the algorithm.

2.1   An introduction to buffered propagation

In external-memory data structures, there is an asymmetry that allows for inserts/updates/deletes to be implemented asymptotically faster than query operations. The fundamental technique for achieving these speedups is **buffered propagation**, in which one propagates insert/update/delete operations down the tree in buffered batches. For each node x in the tree, if x has f children $c_1, \ldots, c_f$, then x maintains a buffer of size B/f for each of those children. Each buffer collects insert/update/delete messages destined for that child c_i. Messages are flushed down from x to the children $c_1, \ldots, c_f$ in collections of size B/f (i.e., whenever a buffer for one of the children overflows). The $O(1)$ I/Os that are used to perform a buffer flush are shared across $\Theta(B/f)$ insert/update/delete operations. By giving a node x a smaller fanout, one can decrease the amortized cost of a buffer flush, making inserts/updates/deletes faster. On the other hand, smaller fanouts also make the height of the tree larger, which makes queries slower.

A classic example of buffered propagation is the B^ε-tree [8, 10, 15, 16], which has found applications in databases [10, 35, 36] and file systems [28, 37–41]. In a B^ε-tree, all nodes have the same fixed fanout f (typically, one sets $f = B^\varepsilon$ for some constant ε). Queries cost $O(\log_f n)$ and insert/update/delete operations have amortized cost $O(\frac{f}{B} \log_f N)$, so that, e.g., an insert/update/delete-heavy workload can be performed asymptotically faster than in standard B-trees if f is selected to be small. An interesting feature of this tradeoff curve is that, if $B \gg f \log_f N$, then insert/update/delete operations can even take *sub-constant* amortized time—the same guarantee is not possible for queries.

The fanout f used within a B^ε-tree can be tuned to the sequence of operations. One must be careful not to select the wrong fanout for the workload, however. For example, the $B^{1/2}$-tree, where $f = \sqrt{B}$, achieves an insert/update/delete performance $O(\frac{\log_B n}{\sqrt{B}})$ while achieving an optimal query performance of $O(\log_B n)$. But, if a workload consists exclusively of inserts/updates/deletes then the $B^{1/2}$-tree will perform a factor of $\sqrt{B}$ away from optimal.

Although we typically think of B^ε-trees as having fanout f that is uniform across all nodes (and unchanging), the B^ε-tree generalizes to a class of data structures where different nodes can have different fanouts. In this paper we define a broad class of data structures that we call **buffered propagation trees**, and which can be viewed as weight-balanced B^ε-trees with non-uniform fanout.

Non-uniform fanouts are especially natural if some parts of the key space are insert/update/delete heavy and other parts of the key space are query heavy. A buffered propagation tree can pick a large f for nodes that see mostly queries and a small f for nodes that see mostly inserts/updates/deletes. This means that pivots can be strategically selected in order to split collections of operations in a way that sends queries in one direction and inserts/updates/deletes in another.[3] By contrast, since f is uniform in a B^ε-tree, the performance of the tree is fairly insensitive to pivot choice. By choosing the right pivots and local f, one can potentially exploit the underlying spatial locality of the workload to achieve asymptotic improvements over any uniform-fanout B^ε-tree. In the dynamic case, where a buffered propagation tree is permitted to change its structure over time, it can also adapt to the temporal locality of the sequence of operations being performed.

2.2   Formally defining buffered propagation trees

We now formally define the class of buffered propagation trees. To simplify discussion, we restrict ourselves to queries/inserts/updates—discussion of deletes can be found in the extended version of the paper [11].

For each node x in a buffered propagation tree, let $\mathcal{K}(x)$ be the keys stored in the subtree rooted at x. Let $d(x)$ be the number of children that x has and call them $c_1, \ldots, c_{d(x)}$. Then x selects some subset $p_1 < \cdots < p_{d(x)-1} \in \mathcal{K}(v)$ to act as **pivots**. The children $c_1, \ldots, c_{d(x)}$ of x then have key sets

---

[3]As we will see later in this paper, the careful selection of pivots can have substantial asymptotic impact on the performance of the tree, even when fanouts are selected optimally.

$\mathcal{K}(c_1) = \mathcal{K}(x) \cap (-\infty, p_1], \mathcal{K}(c_2) = \mathcal{K}(x) \cap (p_1, p_2], \ldots, \mathcal{K}(c_j) = \mathcal{K}(x) \cap (p_{j-1}, \infty]$. The result is that each node x is associated with some interval of keys, called x's **key range**, such that any operation on that key range is routed through x.

The **size** of a node is the number of keys in the node's key range in the tree.[4] Every node in the tree has a **target size** (for leaves the target size is $B/2$), dictating what size the node is supposed to be: as a rule, at any given moment, if a node has target size s, then its true size must be in the range $[s, O(s)]$. All the children of a node must have the same target sizes as one another, and we refer to the target size of the children of a node as its **target child size**. Without loss of generality, the target child size of a node is smaller than the target size.

Flushing messages between nodes in a buffered propagation tree   The **target fanout** of an internal node is defined to be the target size divided by the target child size. If a node has target fanout f, then the node maintains a buffer of size B/f for each of its children. The buffer for each child c stores insert/update messages for that child—these messages keep track of insert/update operations that need to be performed on keys in c's key range.

To understand how insert/update messages work, it is helpful to think about the progression of a given message down the tree. Any given insert/update operation on some key k inserts a message into a buffer at the root. Over time, the message then travels to the leaf whose key range contains k, at which point the insert/update operation is finally applied. Whenever a buffer for some child y overflows in a node x, that buffer is flushed to the child y; and the messages in the buffer are distributed appropriately across y's buffers; this may then cause buffers in y to overflow, etc..

In order to perform a query on a key k, one traverses the root-to-leaf path to the leaf ℓ that contains k in its key range. By examining the messages in the buffers of the nodes in the root-to-leaf path, as well as the contents of leaf ℓ, the tree can answer the query on key k.

Modifying a buffered propagation tree and defining speed-limitation   A buffered propagation tree can dynamically change the fanouts and pivot-choices within nodes in order to adapt to the sequence of operations being performed.

The most basic operation that a buffered propagation tree can perform is to split a node x into two nodes x_1, x_2 whose target-sizes/target-child-sizes are the same as x's were. This is known as a **balanced split**. Balanced splits allow for the tree to perform weight-balancing, and we will treat balanced splits as being free (for our adversary), meaning they do not cost any I/Os, even if the tree is K-speed-limited for some K.

The other way that a buffered propagation tree can modify itself is through **batch rebuild**, in which some collection of nodes in the tree are replaced with new nodes (using possibly different pivots and fanouts than before).

In more detail, when performing a batch rebuild, we can take any set of nodes $X = \{x_1, \ldots, x_m\}$, and and replace them with a different set of nodes $Y = \{y_1, \ldots, y_{m'}\}$ arbitrarily, with the restriction that after the replacement, the tree should still be valid (i.e., each node meets its target size requirement, each child has target size equal to the parent's child target size, and pairs of consecutive key ranges are separated by a valid pivot). Note that after a batch rebuild, buffers may be significantly overflowed in some nodes, in which case the tree must perform a series of buffer flushes to fix this. If a buffer is overflowed by a factor of k, then flushing that buffer takes $\Theta(k)$ I/Os.[5]

A K-speed-limited buffered propagation tree is limited as follows: the tree is only permitted to devote a $\frac{1}{K}$ fraction of its I/Os to batch rebuilds. Another way to think about this is that I/Os spent on batch rebuilds are a factor-of-K more expensive than other I/Os. So a batch rebuild of a set of nodes X into a new set of nodes Y costs $(|X| + |Y|)K$ I/Os.

Defining K-smooth inputs   A **K-smooth input** is any sequence of operations with the following property: The optimal buffered propagation tree cost C for those operations is within a constant factor of the

---

[4]This can differ from $|\mathcal{K}(x)|$ for node x because insertions into x's key range can reside in a buffer above x.

[5]We can also think of the flush as being partitioned into $\lfloor k \rfloor$ distinct flushes, each of which flushes B/f items.

optimal K-speed-limited buffered propagation tree cost C' for those operations. Intuitively, this means that there is an optimal (or at least near-optimal) buffered propagation tree that, during the sequence of operations, spends only a $O(1/K)$-fraction of its I/Os on optimizing the structure of the tree for the sequence.

Note that we are intentionally generous in what we consider to be "optimizing the structure of the tree". Balanced splits are *not* counted against the adversary, are not affected by K-speed-limitation, and thus do not factor into K-smoothness. This is important because on an insertion-heavy workload, a tree may be *forced* to perform a large number of balanced splits, even if the tree is not changing its fanouts/pivots in any interesting way. Thus, insertion-heavy workloads would penalize the adversary unfairly for I/Os that the adversary has no choice but to spend.

To achieve dynamic optimality for K-smooth inputs, it suffices to achieve dynamic optimality (for all inputs) against K-speed-limited buffered propagation trees:

Observation 1. *If a data structure T is c-competitive against dynamic K-speed-limited buffered propagation trees, then T is $O(c)$-competitive on K-smooth input sequences against dynamic buffered propagation trees.*

Throughout the body of the paper, we shall focus on the problem of achieving dynamic optimality against K-speed-limited buffered propagation trees, since this problem is strictly more general than the problem of considering K-smooth inputs.

Defining Speed-Limited OPT In this paper, we will consider the class of K-speed-limited buffered propagation trees, where $K = B^{3\delta}$ for some parameter δ. Given a sequence of operations S, we define speed-limited OPT (or OPT for short) to be the K-limited buffered propagation tree that achieves the minimum total I/O cost on that sequence of operations. We will design a data structure, the Jεllo Tree, that is competitive with OPT.

Caching in OPT and in the Jεllo Tree OPT is assumed to a have a cache that stores the top of OPT's tree. As per the Disk-Access Model [2], any accesses to nodes that are cached are free, in the sense that they do not incur I/Os. We assume that the cache for OPT stores any node x whose target size is above N/C, for some caching parameter C. (Note that this means, w.l.o.g., that OPT may as well make such nodes be fully insert/update-optimized with $\Theta(1)$ fanouts—and furthermore, w.l.o.g., OPT does not perform batch rebuilds on cached nodes.) We will further assume $N/C = N^{1-\Omega(1)}$ (meaning that if OPT were to have uniform fanout, then at most a constant fraction of OPT's tree levels would be cached).

In the same way that we assume a factor of $B^{3\delta}$ resource augmentation in terms of speed-limitation, the Jεllo Tree will is also given a factor of $B^{3\delta}$ cache-size resource augmentation against OPT. Namely, we will assume that the Jεllo Tree caches any node x whose size is above $\frac{N}{CB^{3\delta}}$. If both data structures were B^δ-trees, this would correspond with caching $O(1)$ more layers than OPT caches.

2.3 Results

Our main theorem is the following:

Suppose that $B \geq \Omega(\log N)$ and that B is sufficiently large as a function of $1/\delta$. Let α be the total I/O cost incurred by the Jεllo Tree, and let β be the total cost incurred by the optimal $B^{3\delta}$-speed-limited buffered propagation tree OPT using a factor of $B^{3\delta}$ smaller cache than does the Jεllo Tree. Then $\alpha \leq O(\beta/\delta)$.

We also present a version of the theorem that does not assume resource augmentation on cache size. As long as $N \gg B$, then the cost of removing the resource augmentation is only a small additive I/O cost per operation.

Suppose that $B \geq \Omega(\log N)$ and that B is sufficiently large as a function of $1/\delta$. Let α be the total I/O cost incurred by the Jεllo Tree, let I be the total number of inserts/updates performed on the Jεllo Tree, and let R be the total number of queries performed on the Jεllo Tree. Let β be the total cost incurred by the optimal $B^{3\delta}$-speed-limited buffered propagation tree OPT using the same cache size as the Jεllo Tree uses. Then

$$\alpha \leq O(\beta/\delta) + \min\{I/B^\delta, R\log B^\delta\}.$$

In the extended version of the paper [11], we also discuss how to incorporate deletes into both the definition of a speed-limited adversary and the design and analysis of the Jεllo Tree.

3   Technical Overview

Because both the Jεllo Tree itself and its analysis are quite intricate, in this section we present a sketch of the main ideas in the data structure and our proofs. The full data structure and its analysis appear in the extended version of the paper [11].

To simplify the presentation, we begin by considering optimality against a weakened version of OPT. As subsections proceed, we remove restrictions on OPT and work our way towards achieving dynamic optimality the optimal $B^{3\delta}$-speed-limited buffered propagation tree.

We begin in Subsection 3.1 by considering an OPT that has uniform fanouts (i.e., OPT is a B^ε-tree with optimal fanout). In Subsection 3.2, we consider an OPT that is allowed arbitrary fanouts, but is restricted in its ability to select pivots. In Subsection 3.3, we examine the obstacles that arise OPT is permitted to select pivots arbitrarily. Finally, in Subsection 3.4, we consider the full version of OPT, in which OPT gets to select both pivots and fanouts freely.

3.1   A warmup: designing a fanout-convergent tree

Suppose we are given an initial set of N records to be stored in a buffered propagation tree T with $L = \Theta(N/B)$ leaves, and we are given a sequence of operations $S = \langle s_1, s_2, \ldots, s_k \rangle$ of inserts/updates and queries. Let $C_f(S)$ be the cost that the operations S would incur if T were implemented as a B^ε-tree with fanout f. In this section, we present the **fanout-convergent tree**, which is a data structure for implementing the operations S so that the total cost is $O(\min_f C_f(S))$ (without knowing S ahead of time).

Problem: the cost of rebuilds   A natural approach to achieving cost $O(\min_f C_f(S))$ would be to treat the selection of f as a multi-armed bandit problem [4]. The problem with this approach is that, in order to offset the costs of rebuilding the tree in each trial of the multi-armed bandit problem, each individual trial must be very long. The result is that, in the time that it takes for the tree to change size by a constant factor we would only be able to perform a small number of trials, preventing the multi-armed bandit algorithm from converging fast enough to be useful.

Saving time by moving in only one direction   In order to keep the total costs of tree rebuilds small, we only adjust the fanout f in one direction. The tree begins as fully query-optimized, i.e., with fanout B, and over time the fanout decreases monotonically. A key insight is that, whenever a tree with fanout f is rebuilt as a new tree with smaller fanout, the number of I/Os needed to perform this is only $O(L/f)$, since only the *internal* nodes of the tree need to be reconstructed. It follows that if a tree begins with fanout B, and each successive rebuild shrinks the fanout by a factor of at least two, then the total cost of all of the rebuilds is a geometric series bounded by $O(L)$ I/Os. (Here we are treating the size of the tree as staying $O(N)$ at all times, but as we shall see momentarily, this assumption is without loss of generality.)

Any buffered propagation tree must incur at least $\Omega((\log N)/B) \geq \Omega(1/B)$ cost per insert/update and at least $\Omega(\log_B N) \geq \Omega(1)$ cost per query. Thus, whenever either (a) the total number of inserts/updates surpasses N or (b) the total number of queries surpasses L, then the $O(L)$ cost of rebuilds has been amortized away. Whenever either (a) or (b) occurs, we restart the entire procedure from scratch, returning to a fanout of B.

Since we restart our data structure each time that one of (a) or (b) occurs, we can assume without loss of generality that the number of inserts/updates in S is at most N, that the number of queries in S is at most L, and that one of the two inequalities is strict (there are either exactly N inserts/update or exactly L queries). Given such an S, our challenge is to decrease the fanout over time in such a way that we achieve total cost $O(\min_f C_f(S))$.

Selecting query-biased fanouts   Let $f_0, f_1, \ldots$ be the sequence of fanouts, where operation s_i is performed on a tree of fanout f_{i-1}. Note that $f_0 = B$, that $f_0 \geq f_1 \geq f_2 \geq \cdots$, and that the fanout f_{i-1} must be determined based only on the first $i-1$ operations. When selecting fanouts, we do not need to consider the costs of rebuilds, since in total they sum to at most $O(L)$.

We select the fanouts $f_0, f_1, f_2, \ldots$ so that they are always slightly *query-biased*. In particular, if the first i operations contain $R_i \leq L$ queries and $W_i \leq N$ inserts/updates, then we select the fanout f_i to be the optimal fanout for performing L queries and W_i inserts/updates. That is, we always treat the number of queries as L, even if it is much smaller. This rule ensures that the sequence $f_0 \geq f_1 \geq f_2 \geq \cdots$ is monotone decreasing.

Analyzing the performance   The analysis of each s_i is made slightly difficult by the fact that s_i is performed with fanout f_{i-1} instead of with fanout f_i. One useful observation is that, by slightly tweaking the algorithm, we can achieve performance asymptotically as good as if each operation i were performed with fanout f_i. This can be enforced by performing the i-th operation with fanout $\sqrt{f_{i-1}}$ (note that square-rooting the fanout only hurts the query cost by a constant factor and improves insert/update cost). As long as $\sqrt{f_{i-1}} \leq f_i$, then this is asymptotically as good as using fanout f_i. On the other hand, since the sequence $f_0, f_1, f_2, \ldots$ is monotone decreasing there can only be $O(\log \log B)$ indices i for which $\sqrt{f_{i-1}} \geq f_i$ and the cost of these $O(\log \log B)$ operations is negligible since each costs $O(\log L)$ I/Os.

Let f be the optimal fanout for performing all of the operations in S (recall that, without loss of generality, S has at most L queries and N inserts/updates). Each of the fanouts $f_0, f_1, f_2, \ldots$ are query-biased in the sense that $f_i \geq f$. As a result, we need not worry about the performance of query operations, that is, we can perform the analysis as though queries take time 0. It follows that, without loss of generality, we may assume that all query operations are performed at the beginning of the workload (since the positions of the queries do not affect the fanouts used for the inserts/updates). Furthermore, we can assume that the number of queries is precisely L, since it turns out that adding $\leq L$ queries to a workload with N inserts/updates does not affect the asymptotic cost of the workload.

In summary, the following two simplifying assumptions are without loss of generality: that f_i is the fanout used to perform s_i, and that S starts with L queries followed only inserts/updates. One consequence of the second assumption is that the query-biased rule for selecting the fanouts is equivalent to:

$$f_i = \operatorname{argmin}_f C_f(\langle s_1, \ldots, s_i \rangle). \tag{1}$$

Using Eq. 1, we can prove that the first i operations cost at most $C_{f_i}(\langle s_1, \ldots, s_i \rangle)$ (which in turn is $\min_f C_f(\langle s_1, \ldots, s_i \rangle)$). If we assume that this holds for $i-1$, then by induction the cost of the first i operations is at most,

$$
\begin{aligned}
& C_{f_{i-1}}(\langle s_1, \ldots, s_{i-1} \rangle) + C_{f_i}(\langle s_i \rangle) \\
= {} & \min_f C_f(\langle s_1, \ldots, s_{i-1} \rangle) + C_{f_i}(\langle s_i \rangle) \\
\leq {} & C_{f_i}(\langle s_1, \ldots, s_{i-1} \rangle) + C_{f_i}(\langle s_i \rangle) \\
= {} & C_{f_i}(\langle s_1, \ldots, s_i \rangle).
\end{aligned}
$$

It follows that the cost of all the operations $s_1, \ldots, s_k$ is bounded by $\min_f C_f(\langle s_1, \ldots, s_k \rangle)$, as desired.

The guarantee achieved above, in which we are competitive with the best fixed fanout f, is the simplest adaptive guarantee that one could hope for. It does not adapt to the spatial-locality of where operations are performed in the tree, however, meaning it is still far from optimal.

3.2   Considering an OPT with Fixed Pivots and Keys

Before considering dynamic optimality over the class of speed-limited buffered propagation trees, we consider a simpler class of adversaries that we call **fixed-pivot buffered propagation trees** (or **fixed-pivot trees** for short). A fixed-pivot tree contains some fixed set of N records (where N is assumed to be a power of two)
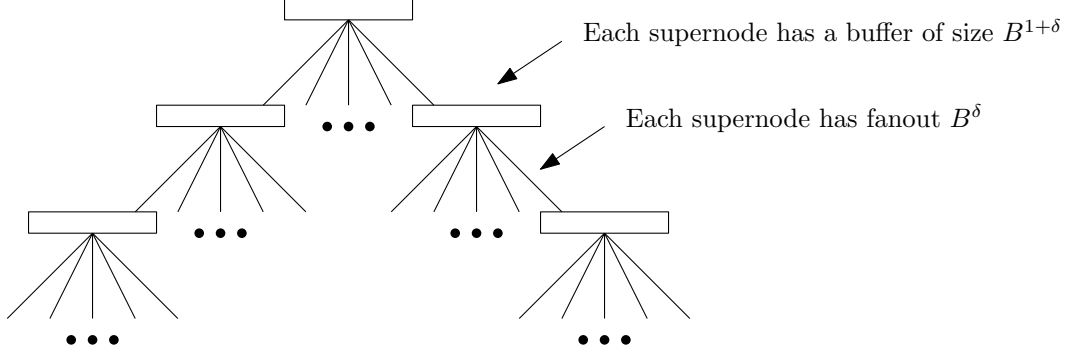
Figure 1: The fixed-pivot Jεllo Tree consists of supernodes with fixed fanouts B^δ. Each supernode has a buffer of size $B^{1+\delta}$. To handle the fact that the buffer is size $\omega(B)$, each buffer is itself implemented as a fanout-convergent tree.

and supports query and update operations (but not inserts and deletes). A fixed-pivot tree is any buffered propagation tree that satisfies the **fixed-pivot-structure property:** every internal node x has a power-of-two fanout, and each of x's children subtrees are exactly equal-size. The fixed-pivot-structure property ensures that there is essentially no freedom to select pivots in a fixed-pivot tree. In particular, each subtree has some power-of-two size 2^j and the rank of the subtree's final element (i.e., the pivot for the subtree) is forced to be a multiple of 2^j. We now describe the **fixed-pivot Jεllo Tree**, which is $O(\delta^{-1})$-competitive with any $B^{3\delta}$-speed-limited fixed-pivot tree.

The structure of a fixed-pivot Jεllo Tree One of the challenges of dynamically adapting the fanout of a node x is that, whenever the x's fanout changes, x's children must be split or merged accordingly, which consequently affects their fanouts (and, in particular, when you increase or decrease the fanout of x, the merging/splitting that this action forces upon the children has the opposite effect of decreasing or increasing their fanouts, respectively). The interdependence between each node x and its children complicates the task of dynamically adapting fanouts.

The fixed-pivot Jεllo Tree solves this issue by decomposing the tree into what we call **supernodes**. Every supernode has a fixed fanout of B^δ (which we will also assume is a power of two). Abstractly, each supernode x maintains a buffer of size $B^{1+\delta}$, allowing for the supernode to buffer up to B messages for each of its children. This large buffer allows for the supernode x to be fully insert/update-optimized (meaning that it flushes messages B at a time) while still having large fanout.

The downside of a large buffer is that the cost of maintaining and searching within the buffer is potentially substantial. In order to optimize this cost, we implement each supernode's buffer as a fanout-convergent tree (i.e., the data structure from Section 3.1) that is rebuilt from scratch every $B^\delta \log B^\delta$ I/Os.[6] The supernode structure of a fixed-pivot Jεllo Tree is illustrated in Figure 1.

Whereas the supernode structure of the fixed-pivot Jεllo Tree is static, the internals of each supernode (and namely the fanout-convergent tree that implements the buffer) adapt to the operations that go through the supernode. At first glance, the fixed-pivot Jεllo Tree may seem quite coarse-grained, in the sense that each supernode adapts as an entire unit rather than having individual nodes adapt their fanouts. Nonetheless, we will see that the adaptive power of the data structure is sufficient to make it $O(\delta^{-1})$-competitive with any $B^{3\delta}$-speed-limited fixed-pivot tree.

Imposing a supernode structure on OPT Consider a sequence of query and update operations $S = \langle s_1, s_2, \ldots \rangle$ on the fixed-pivot Jεllo Tree, and let OPT be the optimal $B^{3\delta}$-speed-limited fixed-pivot tree for

---

[6]We remark that supernodes will continue to play a critical role in the design of the (non-fixed-pivot) Jεllo Tree later in this overview. The key difference will be that, in order to simulate optimal pivot-selection, the internal structure of each supernode will become substantially more sophisticated.
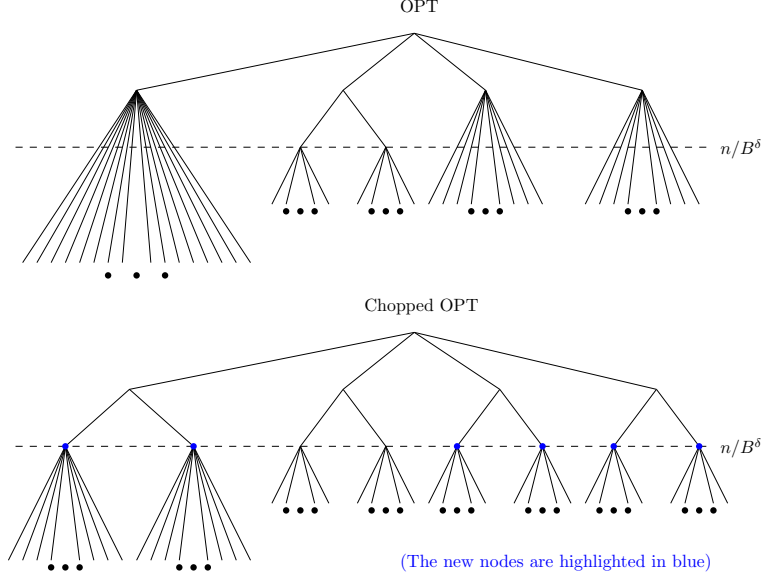
Figure 2: Chopped OPT is constructed from OPT by adding additional nodes so that every root-to-leaf path includes nodes of sizes $N, N/B^\delta, N/B^{2\delta}, \ldots$. Here, we show an example of nodes being added of sizes N/B^δ (the new nodes are in blue). Notice that, when a node is added, it takes some of the children of its parent.

S.

In order to compare the fixed-pivot Jεllo Tree to OPT, we begin by modifying OPT into a new structure that we call **Chopped OPT** that can be partitioned into supernodes. To do this, we add to OPT a layer of nodes whose sizes (i.e., the number of keys in their key range) are all exactly $B^{1+\delta}$, a layer of nodes whose sizes are all exactly $B^{1+2\delta}$, a layer of nodes whose sizes are all exactly $B^{1+3\delta}$, and so on; see Figure 2. (Note that some of these nodes may already be present in OPT, in which case they need not be added.) One can think of Chopped OPT as consisting of supernodes, where each supernode has a root of size $B^{1+h\delta}$ for some h and leaves of size $B^{1+(h-1)\delta}$.

Each root-to-leaf path in Chopped OPT is at most a factor of δ^{-1} longer than the same path in OPT. The result is that Chopped OPT is $O(\delta^{-1})$-competitive with OPT. In order to analyze the fixed-pivot Jεllo Tree, we show that it is $O(1)$-competitive with Chopped OPT.

Competitive analysis against Chopped OPT Each supernode in the fixed-pivot Jεllo Tree has a corresponding supernode in Chopped OPT that covers the same key range. For each (non-root) supernode x in the fixed-pivot Jεllo Tree, define the **Chopped-OPT parent** $p(x)$ of x to be the supernode in Chopped OPT whose key range contains x's key range, and whose size (in terms of the number of keys in its key range) is B^δ times larger than x's size. The size requirement means that $p(x)$ sits one layer higher in Chopped OPT than x sits in the fixed-pivot Jεllo Tree.

In order to analyze the performance of a supernode x, there are two cases to consider, depending on whether Chopped OPT modifies the structure of $p(x)$ during x's lifetime. We will see that if $p(x)$ is modified then the speed-limitation on Chopped OPT can be used to amortize the cost incurred by the Jεllo Tree, and otherwise a competitive analysis can be performed to compare the performance of supernode x to that of its parent $p(x)$ in Chopped OPT.

The first case is that, at some point during x's life (recall that each supernode x gets rebuilt from scratch after $B^\delta \log B^\delta$ I/Os), Chopped OPT modifies $p(x)$. In this case, because Chopped OPT is $B^{3\delta}$-speed-limited, one can think of the modification of $p(x)$ as costing Chopped OPT $B^{3\delta}$ I/Os. On the other hand, the supernode $p(x)$ in Chopped OPT is a parent supernode for at most B^δ supernodes x in the Jεllo

11

Tree. Thus we can think of Chopped OPT as paying $B^{(3-1)\delta}$ I/Os to our supernode x. In other words, the $B^{3\delta}$-speed-limitation on Chopped OPT pays for the $B^{\delta} \log B^{\delta}$ I/Os incurred by x during its life.

The second case is that, over the course of x's lifetime, Chopped OPT never modifies $p(x)$. In this case, we compare the total cost incurred by operations in x to the cost incurred by the same operations in $p(x)$.[7]

Define S_x to be the set of query and update operations that go through supernode x during x's lifetime. Whereas the operations in S_x may take different paths than each other down supernode x, all of the operations in S_x take the *same* root-to-leaf path P through supernode $p(x)$ in Chopped OPT. We show that the cost incurred by the operations S_x on the path P in Chopped OPT is minimized by setting all of the fanouts in P to be equal; we call this the ***equal-fanout observation***. Note that the equal-fanout observation *does not* mean that the entire supernode $p(x)$ is optimized by having all of its fanouts equal; the observation just means that for each *individual* path in $p(x)$, the cost of the operations that travel *all the way down* that path would be optimized by setting the fanouts in that path to be equal (different paths would have different optimal fanouts, however).

By the equal-fanout observation, the cost that operations S_x incur in $p(x)$ is asymptotically at least the cost that operations S_x would incur in a fanout-convergent tree containing $L = \Theta(B^{\delta})$ leaves. On the other hand, the buffer in x is implemented as a fanout-convergent tree with $\Theta(B^{\delta})$ leaves. It follows that the cost of operations S_x to x is $O(1)$-competitive with the cost of operations S_x to $p(x)$.

The analysis described above ignores the fact that update messages may propagate down the Jεllo Tree at different times than when they propagate down Chopped OPT. As a result, some of the operations in S_x may actually remain buffered above supernode $p(x)$ in Chopped OPT until well after the end of x's lifetime. By the time these buffered messages make it to $p(x)$, Chopped OPT may have already modified $p(x)$. It turns out that, whenever this occurs, one can extend the charging argument from the first case in order to pay for any I/Os incurred by x.

3.3 The Pivot-Selection Problem

The importance of pivot selection The selection of pivots in a buffered propagation tree can have a significant impact on asymptotic performance. Consider, for example, a sequence of operations $S = \langle s_1, s_2, \ldots \rangle$, where each operation is either a query for some key k_1 or an update for some key k_2, where k_2 is the successor of k_1. If the pivots in the tree are selected independently of the workload S, then all of the operations in S will (most likely) be sent down the same root-to-leaf path. On the other hand, if the buffered propagation tree uses k_1 as a pivot in the root node, then all of the queries to k_1 will be sent down one subtree, while all of the updates to k_2 will be sent down another, allowing for the tree to implement updates in amortized time $O((\log N)/B)$ and queries in amortized time $O(\log_B N)$. Therefore, in order to be competitive with OPT, one must be competitive even in the cases were OPT's pivots split the workload into natural sub-workloads, each of which is optimized separately with properly selected fanouts. As was the case in this example, the *exact* choice of pivot can be very important, meaning that there is no room to select a pivot that is "almost" in the right position.

What supernodes must guarantee Define the ***fanout-convergent cost*** for a set of operations S in a supernode x to be the cost of implementing S in a fanout-convergent tree that has $\Theta(B^{\delta})$ leaves. We say that a key-range $[k_1, k_2]$ in supernode x ***achieves fanout-convergence over some time window W*** if the cost of the operations S that apply to $[k_1, k_2]$ during W is within a constant factor of the fanout-convergent cost of those operations.

Consider what goes wrong in the analysis of the fixed-pivot Jεllo Tree if we allow Chopped OPT to select arbitrary pivots. Recall that in the competitive analysis, we compare each supernode x to its parent $p(x)$ in

---

[7]An important subtlety is the effect that caching may have on x and $p(x)$. We assume that OPT caches all nodes with key-range sizes N/C or larger for some parameter C, and that the Jεllo Tree caches all nodes with key-range sizes $N/(B^{3\delta}C)$ or larger. In other words, the Jεllo Tree caches $O(1)$ more layers of supernodes than does Chopped OPT. The resource augmentation on cache size ensures that, if x is (partially) uncached by the Jεllo Tree, then $p(x)$ is (completely) uncached by Chopped OPT. In Section 2, we also give a version of the analysis that does not assume any resource augmentation in caching, at the cost of incurring a small additional additive cost in the analysis.

Chopped OPT.

If Chopped OPT is permitted to select arbitrary pivots, however, then x may actually have *two* parents $p_1(x)$ and $p_2(x)$, each of which partially overlaps x's key range.[8] We need the supernode x to achieve fanout-convergence on *both* of the key ranges $x \cap p_1(x)$ and $x \cap p_2(x)$ (rather than simply achieving for the entire key range of x).

Since the (non-fixed-pivot) Jεllo Tree does not know what the pivot p is that separates $p_1(x)$ and $p_2(x)$, the Jεllo Tree must be able to provide a guarantee for all possible such pivots. For any pivot p, define the **p-split** cost of a supernode x to be the sum of (a) the fanout-convergent cost for the operations in x that involve keys $\leq p$, and (b) the fanout-convergent cost for the operations in x that involve keys $> p$. Each supernode x must provide what we call the **Supernode Guarantee:** for any p, x's total actual cost is $O(1)$-competitive with its p-split cost.

One additional requirement in the supernode guarantee: speed One of the aspects of pivot selection that makes it difficult is that a supernode x's lifetime may be relatively short. In particular, whenever a supernode x's size changes by a sufficiently large constant factor, the Jεllo Tree is forced to perform rebalancing on that supernode, thereby ending x's life. In the worst case, for supernodes x in the bottom layer of the tree, the lifetime of the supernode may consist of only $O(B^{1+\delta})$ inserts (and some potentially small number of queries), meaning that the total I/O-cost of the supernode could be as small as $O(B^{\delta} \log B^{\delta})$. Thus convergence to the supernode guarantee must be fast.

This issue is further exacerbated by the fact that the supernode guarantee requires not only pivot selection but also optimal fanout-convergence on each side of that pivot. But even just the time to achieve optimal fanout-convergence on a tree with B^{δ} leaves, using the approach in Subsection 3.1, may take $\Theta(B^{\delta} \log B^{\delta})$ I/Os. This means that the natural approach of achieving fanout convergence from scratch (on both sides of the pivot) every time that we modify our choice of pivot is not viable. Instead, the processes of pivot selection and fanout convergence must interact so that both pieces of the supernode guarantee can be achieved concurrently within a small time window.

The difficulty of a moving target One natural approach to pivot-selection is to keep a random sampling of the operations performed so far and to use this to determine an approximation of the pivot p_{opt} that is optimal for performing all of the operations so far. If the pivot p_{opt} is relatively static over time (e.g., if the operations being performed are drawn from some fixed stochastic distribution), then such an approach may work well. On the other hand, if p_{opt} shifts over time, then the approach of "following" p_{opt} fails.

To see why, suppose that $p_{\text{opt}}(t)$ is the optimal pivot choice for performing the first t operations and that for operation t we use $p_{\text{opt}}(t-1)$ as our pivot, i.e., we perfectly follow p_{opt}. Further suppose that the optimal pivot $p_{\text{opt}}(t)$ places the insert-heavy portion of the workload on its left side and the query-heavy portion on its right side. One example of what may happen is that p_{opt} drifts to the right over time, due to inserts being performed to the right of where p_{opt} just was. The result is that, for many insert operations t, the pivot $p_{\text{opt}}(t-1)$ may be to the left of the insert-key even though the pivot $p_{\text{opt}}(t)$ is to the right of the insert-key — this makes $p_{\text{opt}}(t-1)$ a poor pivot to use for operation t. One can attempt to mitigate this by overshooting and using a pivot to the right of $p_{\text{opt}}(t-1)$, but this then opens us up to other vulnerabilities (such as p_{opt} drifting to the left).

In the next subsection, where we describe our techniques for implementing the supernode guarantee, we will see an alternative approach to pivot selection that allows for our performance to converge to that of the optimal pivot, without having to "follow" it around. We will then also see how to integrate pivot selection with fanout convergence so that the supernode guarantee holds even for supernodes with short lifetimes.

3.4 Providing the supernode guarantee

As is the case for the fixed-pivot Jεllo Tree, the supernodes buffers in the (non-fixed-pivot) Jεllo Tree are implemented with a tree structure. To avoid ambiguity, we refer to the leaves of this tree structure as the

---

[8]Because each of x's parents covers a larger key-range than x, x can have at most two parents.

supernode's *leaves* (even though they are the children of the supernode, and are therefore other supernodes).

Simplifying pivot selection by shortcutting leaves A given supernode may have a large number of possible pivots (especially if the supernode is high in the tree). On the other hand, as discussed in Subsection 3.3, picking the wrong pivot (even by just a little) can be disastrous.

We can reduce the effective number of pivot options by adding a new mechanism called ***shortcutting***. In order to shortcut a leaf ℓ, we store the buffer for leaf ℓ directly in the root node of the supernode, meaning that the root takes 1 more block of space than it would normally. Whenever a leaf is shortcutted, all messages within the supernode destined for that leaf are stored within the root buffer (and not in any root-to-leaf paths). Queries that go through leaf ℓ incur only $O(1)$ I/Os in the supernode, since they can access ℓ directly in the root. Inserts/updates that go through leaf ℓ incur only $O(1/B)$ amortized cost in the supernode, since the leaf gets its own buffer of size B in the root of the supernode.

Because each shortcutted leaf increases the size of the root by 1 block, we can only support $O(1)$ short-cutted leaves at a time. We prove that, to simulate optimal pivot-selection, one can instead select $O(1)$ shortcutted leaves in a way so that one of those shortcutted leaves contains the optimal pivot. This means that, rather than satisfying the supernode guarantee directly, it suffices to instead satisfy the following "shortcutted" version of the guarantee:

- **The Shortcutted Supernode Guarantee:** Consider a sequence of operations S on a supernode x. For any possible shortcutted leaf ℓ, define the ***ℓ-split cost*** of S to be the sum of (a) the fanout-convergent cost of the operations in S that are on keys smaller than those in ℓ; (b) the fanout-convergent cost of the operations in S that are on keys larger than those in ℓ; and (c) the shortcutted cost of implementing the operations in S that apply to leaf ℓ. The total cost of all operations S on a supernode x in its lifetime is guaranteed to be $O(1)$-competitive with the ℓ-split cost of S.

The shortcutted supernode guarantee implies the standard supernode guarantee, but the former is more tractable because now, rather than selecting a specific pivot (out of a possibly very large number of options), we only have to select one of $O(B^\delta)$ leaves to shortcut. Moreover, we get to select *multiple* such leaves at a time (we will end up selecting 3 at a time), which will allow for us to perform an algorithm in which we "chase" the optimal shortcutted leaf from multiple directions at once.

An algorithm for shortcut selection Fix a supernode x and consider the task of implementing the shortcutted supernode guarantee. The algorithm breaks the supernode's lifetime into short ***shortcut convergence windows***, where each shortcut convergence window satisfies the shortcutted supernode guarantee.

Each shortcut convergence window is broken into phases, where the first phase has some length T (in I/Os), and then each subsequent phase i is defined to consist of 1/8-th as many I/Os as the sum of phases $1, 2, \ldots, i-1$. If we think of I/Os as representing time, then each phase i extends the length of the shortcut convergence window by a factor of $1 + 1/8$. At the beginning of each phase i, our algorithm will select three leaves $q < r < s$ to be shortcutted during that phase. These are the only leaves that are shortcutted during the phase; see Figure 3.

At any given time t, define the ***optimal static shortcut*** $\ell_{\text{opt}}(t)$ for the supernode x to be the leaf ℓ that minimizes the ℓ-split cost of the operations performed up until time t (during the current shortcut convergence window). During each shortcut convergence window, we keep track of the optimal static shortcut ℓ_{opt} as it changes over time.

For now, we will describe the shortcut selection algorithm with two simplifying assumptions. The first is that the key-ranges between shortcuts[9] each achieve fanout-convergence during each phase. The second is that the set of leaves in supernode x does not change during the shortcut convergence window (i.e., no node-splits occur). Later we will see how to modify the algorithm to remove both of these assumptions.

We can now describe how the algorithm works. At the beginning of each phase $i > 1$, there are two ***anchor shortcuts*** q and s that have already been shortcutted for all of phase $i - 1$. The key property

---

[9]By this we mean the four key ranges corresponding with the four sets of leaves, $\{1, 2, \ldots, q-1\}$, $\{q+1, \ldots, r-1\}$, $\{r+1, \ldots, s-1\}$, $\{s+1, \ldots\}$.
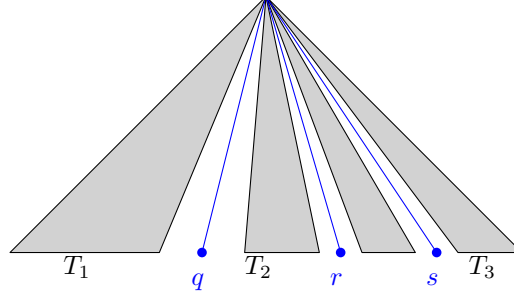
Figure 3: Each supernode selects three children q, r, s (at a time) to shortcut. These children have their buffers stored at the top of the supernode. The four key ranges between q, r, s are implemented as fanout-convergent trees T_1, T_2, T_3, T_4.

that the anchor shortcuts satisfy is that the optimal static shortcut ℓ_{opt} is between them. The two anchor shortcuts q and s remain shortcutted for all of phase i. If, at any point during phase i, the optimal static shortcut ℓ_{opt} crosses one of q or s (so that it is no longer between them), then we **terminate** the entire shortcut convergence window and begin the next one starting with phase 1 again — in a moment, we will argue that whenever the shortcut convergence window terminates, it satisfies the shortcutted supernode guarantee.

In addition to shortcutting the anchors q and s during phase i, we also shortcut the leaf r that is half-way between q and s. At the end of the phase, we then select the anchor shortcuts for phase $i + 1$ to be $\{q, r\}$ if ℓ_{opt} is between q and r, and to be $\{r, s\}$ if ℓ_{opt} is between r and s. The result is that, if the shortcut convergence window does not terminate during a phase i, then the distance between the anchor shortcuts used in phase $i + 1$ will be half as large as the distance between the anchor shortcuts in phase i.

Analyzing running time   Before proving the shortcutted supernode guarantee, we first bound the running time of the shortcut convergence window W. Because the distance between the anchor shortcuts halves between consecutive phases, the window W is guaranteed to terminate within $O(\log B^\delta)$ phases. This means that the number of I/Os incurred by x is at most $O(T(1 + 1/8)^{\log B^\delta}) \leq O(TB^{\delta/5})$. As long as T is reasonably small (e.g., less than $B^{\delta/2}$), then the length of each shortcut convergence window is small enough to fit in a supernode's lifetime.

Proving the shortcutted guarantee   We now argue that, whenever a shortcut convergence window W terminates, the supernode x satisfies the shortcutted supernode guarantee. Recall that the window terminates when the static optimal shortcut ℓ_{opt} crosses over one of the anchors q or s (let's say it crosses q). The fact that ℓ_{opt} crosses over q can be used to argue that the q-split cost of x during W is $O(1)$-competitive with the ℓ_{opt}-split cost of the x during W. Thus our goal is to compare the total cost incurred on x during the shortcut convergence window by our data structure to the q-split cost of x for the same time window.

Let $A_1, A_2, \ldots, A_i$ be the costs in I/Os of phases $1, 2, \ldots, i$ to supernode x, and let $B_1, B_2, \ldots, B_i$ be the q-split costs of the operations in each of phases $1, 2, \ldots, i$ (here q is the shortcut from the final phase i). We wish to show that $A_1 + \cdots + A_i \leq O(B_1 + \cdots + B_i)$. In particular, this will establish that the cost incurred by x during window W is constant-competitive with the q-split cost during the same window.

Recall that each phase is defined to take a constant-fraction more I/Os than the previous phase, meaning that $A_1, \ldots, A_i$ are a geometric series (except for the final term A_i which may be smaller). Thus, rather then proving that $A_1 + \cdots + A_i \leq O(B_1 + \cdots + B_i)$, it suffices to show that $A_{i-1} \leq O(B_{i-1})$.

The fact that q is an anchor shortcut in the final phase i implies that q was shortcutted in x for all of phase $i - 1$. This means that the cost of supernode x to our data structure during phase $i - 1$ is at most the q-split cost of the operations in phase $i - 1$, that is, $A_{i-1} \leq O(B_{i-1})$. As observed above, it follows that $A_1 + \cdots + A_i \leq O(B_1 + \cdots + B_i)$, which completes the proof of the shortcutted supernode guarantee.

15

Removing the simplifying assumptions   At this point, we have completed a high-level overview of how an algorithm can perform shortcut selection in order to achieve the shortcutted supernode guarantee. As noted earlier, however, the analysis makes several significant simplifying assumptions: (1) that the key ranges between shortcuts each achieve fanout-convergence during each phase; and (2) that the set of leaves for supernode x is a static set. Removing these simplifications requires several significant additional technical ideas which we give an overview of in the rest of this subsection.

Handling a dynamically changing leaf set   We begin by removing the assumption that x's leaf set is static. For supernodes x in the bottom layer of the tree (which are the supernodes we will focus on for the rest of this section), the leaf-set of the supernode may change dramatically over the course of the supernode's lifetime, due to inserts causing leaves to split. Thus, during a given phase i of a shortcut convergence window, the number of leaves between the two anchor shortcuts q and s may increase by more than factor of two. This means that the distance between the anchor shortcuts that are used in phase $i+1$ (in terms of number of leaves between them) could be *larger* than the distance between the anchor shortcuts in phase i. If this happens repeatedly, then the shortcut convergence window may never terminate.

To combat this issue, we modify the supernode guarantee as follows. Rather than comparing the cost of a supernode x to the p-split cost of x for every possible pivot p, we only compare the cost of x to the p-split costs for pivots p that were *already present in the tree* at the beginning of x's lifetime. We call these the ***valid pivot options*** for x.

Similarly, we modify the shortcutted supernode guarantee to only consider the ℓ-split cost for leaves ℓ that contain at least one valid pivot option. One can show that the weakened version of the two guarantees still suffices for performing a competitive analysis on the Jεllo Tree.

In order to provide the new version of the shortcutted supernode guarantee, we modify the pivot-selection algorithm as follows. Rather than halving the number of leaves between the anchor shortcuts in each phase, we instead halve the number of valid pivot options contained in the leaves between the anchor shortcuts. That is, rather than selecting the shortcutted leaf r to be half way between the two anchor shortcuts q and s, we instead select r so that it evenly splits the set of valid pivot options between q and s.

With this new algorithm, each shortcut convergence window is guaranteed to terminate within $O(\log B^{1+\delta})$ phases. In order to keep the length of each shortcut convergence window small, we make it so that each phase i is only an $O(\delta)$-fraction as large as the sum of phases $1, 2, \ldots, i-1$ (rather than a 1/8-fraction). One side-effect of this is that, for supernodes in the bottom layer of the tree, the competitive ratio for the supernode guarantee ends up being $O(\delta^{-1})$ (rather than $O(1)$). Nonetheless, this weakened guarantee still turns out to be sufficient for the competitive analysis of the Jεllo Tree.[10]

Efficiently combining pivot selection with fanout-convergence   Next we remove the assumption that, within a given phase of a shortcut convergence window, the key ranges between consecutive shortcuts each achieve fanout-convergence. Recall that a fanout-convergent tree with B^δ leaves requires up to $\Theta(B^\delta \log B^\delta)$ I/Os to converge. Since we cannot afford to make the minimum phase length T be $\Theta(B^\delta \log B^\delta)$, we cannot simply perform fanout-convergence blindly within each phase.

In order to perform fanout-convergence and shortcut selection concurrently, we modify the structure of a supernode as follows. Each supernode now consists of two layers of ***micro-supernodes***, where each micro-supernode has fanout $\Theta(B^{\delta/2})$. Each micro-supernode has the same structure as what we previously gave to supernodes: each micro-supernode can have up to three shortcut leaves, and each micro-supernode implements the key ranges between shortcut leaves as fanout-convergent trees. A leaf ℓ in the full supernode x is considered to be shortcutted in x if ℓ is shortcutted in the micro-supernode ℓ' containing ℓ, and if ℓ' is, in turn, shortcutted in the root micro-supernode; see Figure 4.

Whenever the shortcut selection algorithm for a supernode x selects a new shortcut at the beginning of the phase, it does this by clobbering and rebuilding only two of the micro-supernodes (specifically, the root

---

[10]Importantly, it is only in the bottom layer of the tree where we have this weakened supernode guarantee. That is, the guarantee continues to hold with $O(1)$-competitiveness for all supernodes in higher layers.
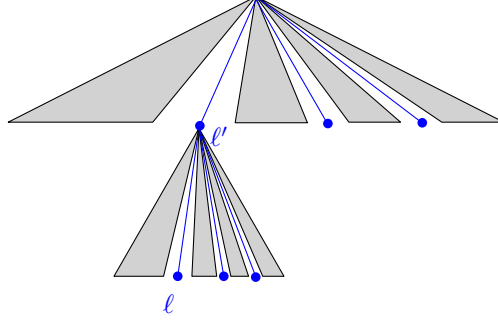
Figure 4: We break each supernode x into two levels, each of which is implemented using micro-supernodes with fanouts $B^{\delta/2}$. To shortcut a leaf ℓ in x, we find the micro-supernode ℓ' containing ℓ; we then rebuild the root micro-supernode r so that ℓ' is shortcutted in r; and finally we rebuild ℓ' so that ℓ is shortcutted in ℓ'. Importantly, this process only disrupts fanout-convergence in two of the $\Theta(B^{\delta/2})$ micro-supernodes.

micro-supernode and one micro-supernode in the bottom layer of x). Critically, this means that the other micro-supernodes continue to perform fanout convergence without interruption.

If a micro-supernode y survives for $cB^{\delta/2}\log B^{\delta/2}$ I/Os, for some sufficiently large constant c, and the shortcuts of y are never changed by the shortcut selection algorithm during those I/Os, then each of the fanout-convergent trees in y are guaranteed to have achieved fanout convergence (or to have incurred negligibly few I/Os). In this case, we say that y also achieves fanout convergence.

When a new shortcut is selected, the actual cost of rebuilding the two micro-supernodes is only $O(B^{\delta/2})$ I/Os. Additionally, the fact that we clobber two micro-supernodes (possibly before they have a chance to achieve fanout-convergence) may disrupt fanout convergence for up to $O(B^{\delta/2}\log(B^{\delta/2}))$ I/Os. In this sense, the total cost of selecting a new shortcut (both the cost in terms of I/Os expended to rebuild the micro-supernodes, and the cost in terms of the I/Os that those supernodes had incurred prior to being clobbered) at the beginning of a phase is $O(B^{\delta/2}\log(B^{\delta/2}))$ I/Os. By setting the minimum phase length T to $cB^{\delta/2}\log(B^{\delta/2})$ for a sufficiently large constant c, we can amortize away this cost using the I/Os incurred in other micro-supernodes during the phase.

Analyzing pivot selection and fanout-convergence concurrently   The two-level structure of a supernode, described above, allows for us to perform shortcut selection and fanout-convergence concurrently with minimal interference. One issue, however, is that the time frame in which a given micro-supernode achieves fanout convergence may overlap multiple phases (and even multiple shortcut convergence windows) of the shortcut selection algorithm. Thus, the introduction of micro-supernodes misaligns fanout-convergence and pivot selection so that the individual shortcut convergence windows may no longer satisfy the shortcutted supernode guarantee.

In order to get around these issues, we define what we call the ***p-re-shortcutted cost*** of a supernode x with respect to a given pivot p. Roughly speaking, the re-shortcutted cost of the supernode x with respect to a pivot p is just the sum of (a) the actual costs incurred by micro-supernodes in x that do not contain p in their key range, and (b) the p-split cost of each micro-supernode that does contain p in its key range. Rather than proving that each shortcut convergence window satisfies the supernode guarantee, we instead prove a weaker property: for each pivot p, the cost of x in each shortcut convergence window is $O(1)$-competitive with the p-re-shortcutted cost of x during the same window. Combining this guarantee across all shortcut convergence windows, we get that the cost of x over its entire lifetime is $O(1)$-competitive with the p-re-shortcutted cost of x during its entire lifetime. Then, using the fact that (almost all of) the micro-supernodes in x achieve fanout-convergence by the end of x's lifetime, we conclude that the p-re-shortcutted cost of x during its lifetime is constant-competitive with the p-split cost of x. Thus, even though each individual shortcut convergence window may not satisfy the supernode guarantee, the supernode x *does* satisfy the supernode guarantee over the course of its lifetime.

References

[1] George M Adel'son-Vel'skii and Evgenii Mikhailovich Landis. An algorithm for organization of information. In *Doklady Akademii Nauk*, volume 146, pages 263–266. Russian Academy of Sciences, 1962.

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

[3] Lars Arge. The buffer tree: A new technique for optimal i/o-algorithms. In *Workshop on Algorithms and Data structures*, pages 334–345. Springer, 1995.

[4] Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E Schapire. Gambling in a rigged casino: The adversarial multi-armed bandit problem. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 322–331. IEEE, 1995.

[5] Mihai Bădoiu, Richard Cole, Erik D Demaine, and John Iacono. A unified access bound on comparison-based dynamic dictionaries. *Theoretical Computer Science*, 382(2):86–96, 2007.

[6] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta informatica*, 1(4):290–306, 1972.

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

[8] Michael A Bender, Rathish Das, Martín Farach-Colton, Rob Johnson, and William Kuszmaul. Flushing without cascades. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 650–669. SIAM, 2020.

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

[10] Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, Jun Yuan, and Yang Zhan. And introduction to be-trees and write-optimization. *Login; Magazine*, 40(5), 2015.

[11] Michael A Bender, Martin Farach-Colton, and William Kuszmaul. What does dynamic optimality mean in external memory? *arXiv preprint*, 2021.

[12] Dhruba Borthakur. Under the hood: Building and open-sourcing rocksdb. *Facebook Engineering Notes*, 2013.

[13] Presenjit Bose, Karim Douïeb, John Iacono, and Stefan Langerman. The power and limitations of static binary search trees with lazy finger. In *International Symposium on Algorithms and Computation*, pages 181–192. Springer, 2014.

[14] Prosenjit Bose, Karim Douïeb, and Stefan Langerman. Dynamic optimality for skip lists and b-trees. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1106–1114. Citeseer, 2008.

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

[16] Adam L Buchsbaum, Michael H Goldwasser, Suresh Venkatasubramanian, and Jeffery R Westbrook. On external memory graph traversal. In *SODA*, pages 859–860, 2000.

[17] Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. Multi-finger binary search trees. *arXiv preprint arXiv:1809.01759*, 2018.

[18] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. part i: Splay sorting log n-block sequences. *SIAM Journal on Computing*, 30(1):1–43, 2000.

[19] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. part i: Splay sorting log n-block sequences. *SIAM Journal on Computing*, 30(1):1–43, 2000.

[20] Douglas Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[21] Alex Conway, Martin Farach-Colton, and Philip Shilane. Optimal hashing in external memory. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*, page 39. Schloss Dagstuhl-Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing, 2018.

[22] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 2001.

[23] Erik D Demaine, Dion Harmon, John Iacono, and Mihai P a ˘ traşcu. Dynamic optimality—almost. *SIAM Journal on Computing*, 37(1):240–251, 2007.

[24] Erik D Demaine, John Iacono, Grigorios Koumoutsos, and Stefan Langerman. Belga b-trees. In *International Computer Science Symposium in Russia*, pages 93–105. Springer, 2019.

[25] John Howat, John Iacono, and Pat Morin. The fresh-finger property. *arXiv preprint arXiv:1302.6914*, 2013.

[26] John Iacono. Alternatives to splay trees with o (log n) worst-case access times. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 516–522. Society for Industrial and Applied Mathematics, 2001.

[27] John Iacono. In pursuit of the dynamic optimality conjecture. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 236–250. Springer, 2013.

[28] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. Betrfs: Write-optimization in a kernel file system. *ACM Transactions on Storage (TOS)*, 11(4):1–29, 2015.

[29] Patrick O'Neil, Edward Cheng, Dieter Gawlic, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[30] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514, 2017.

[31] Murray Sherk. Self-adjusting k-ary search trees. *Journal of Algorithms*, 19(1):25–44, 1995.

[32] Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391, 1983.

[33] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[34] Robert Endre Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5(4):367–378, 1985.

[35] Tokutek, Inc. TokuDB® for MySQL Storage Engine, 2009. http://www.tokutek.com. URL: http://www.tokutek.com.

[36] Tokutek Inc. TokuDB. http://www.tokutek.com/, 2011.

[37] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael Bender, et al. Optimizing every operation in a write-optimized file system. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pages 1–14, 2016.

[38] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A Bender, et al. Writes wrought right, and other adventures in file system optimization. *ACM Transactions on Storage (TOS)*, 13(1):1–26, 2017.

[39] Yang Zhan, Alex Conway, Yizheng Jiao, Eric Knorr, Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E Porter, and Jun Yuan. The full path to full-path indexing. In *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*, pages 123–138, 2018.

[40] Yang Zhan, Alexander Conway, Yizheng Jiao, Nirjhar Mukherjee, Ian Groombridge, Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E Porter, et al. How to copy files. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 75–89, 2020.

[41] Yang Zhan, Yizheng Jiao, Donald E Porter, Alex Conway, Eric Knorr, Martin Farach-Colton, Michael A Bender, Jun Yuan, William Jannen, and Rob Johnson. Efficient directory mutations in a full-path-indexed file system. *ACM Transactions on Storage (TOS)*, 14(3):1–27, 2018.