The Full Path to Full-Path Indexing

Yang Zhan, Alex Conway[†], Yizheng Jiao, Eric Knorr[†], Michael A. Bender[‡],
Martin Farach-Colton[†], William Jannen[¶], Rob Johnson[*], Donald E. Porter, Jun Yuan[‡]
The University of North Carolina at Chapel Hill, [†]*Rutgers University,*
[‡]*Stony Brook University,* [¶]*Williams College,* [*]*VMware Research*

Abstract

Full-path indexing can improve I/O efficiency for workloads that operate on data organized using traditional, hierarchical directories, because data is placed on persistent storage in scan order. Prior results indicate, however, that renames in a local file system with full-path indexing are prohibitively expensive.

This paper shows how to use full-path indexing in a file system to realize fast directory scans, writes, and renames. The paper introduces a range-rename mechanism for efficient key-space changes in a write-optimized dictionary. This mechanism is encapsulated in the key-value API and simplifies the overall file system design.

We implemented this mechanism in BetrFS, an in-kernel, local file system for Linux. This new version, BetrFS 0.4, performs recursive greps 1.5x faster and random writes 1.2x faster than BetrFS 0.3, but renames are competitive with indirection-based file systems for a range of sizes. BetrFS 0.4 outperforms BetrFS 0.3, as well as traditional file systems, such as ext4, XFS, and ZFS, across a variety of workloads.

1 Introduction

Today's general-purpose file systems do not fully utilize the bandwidth of the underlying hardware. For example, ext4 can write large files at near disk bandwidth but typically creates small files at less than 3% of disk bandwidth. Similarly, ext4 can read large files at near disk bandwidth, but scanning directories with many small files has performance that ages over time. For instance, a git version-control workload can degrade ext4 scan performance by up to 15× [14, 55].

At the heart of this issue is how data is organized, or ***indexed***, on disk. The most common design pattern for modern file systems is to use a form of indirection, such as inodes, between the name of a file in a directory and its physical placement on disk. Indirection simplifies implementation of some metadata operations, such as renames or file creates, but the contents of the file system can end up scattered over the disk in the worst case. Cylinder groups and other best-effort heuristics [32] are designed to mitigate this scattering.

Full-path indexing is an alternative to indirection, known to have good performance on nearly all operations. File systems that use full-path indexing store data and metadata in depth-first-search order, that is, lexicographic order by the full-path names of files and directories. With this design, scans of any directory subtree (e.g., `ls -R` or `grep -r`) should run at near disk bandwidth. The challenge is maintaining full-path order as the file system changes. Prior work [15, 22, 23] has shown that the combination of write-optimization [5–7, 9–11, 36, 43, 44] with full-path indexing can realize efficient implementations of many file system updates, such as creating or removing files, but a few operations still have prohibitively high overheads.

The Achilles' heel of full-path indexing is the performance of renaming large files and directories. For instance, renaming a large directory changes the path to every file in the subtree rooted at that directory, which changes the depth-first search order. Competitive rename performance in a full-path indexed file system requires making these changes in an I/O-efficient manner.

The primary contribution of this paper is showing that one *can*, in fact, use full-path indexing in a file system without introducing unreasonable rename costs. A file system can use full-path indexing to improve directory locality—and still have efficient renames.

Previous full-path indexing. The first version of BetrFS [22, 23] (v0.1), explored full-path indexing. BetrFS uses a write-optimized dictionary to ensure fast updates of large and small data and metadata, as well as fast scans of files and directory-tree data and metadata. Specifically, BetrFS uses two B^ε-trees [7, 10] as persistent key-value stores, where the keys are full path names to files and the values are file contents and metadata, respectively. B^ε-trees organize data on disk such that logically contiguous key ranges can be accessed via large, sequential I/Os. B^ε-trees aggregate small updates into large, sequential I/Os, ensuring efficient writes.

This design established the promise of full-path indexing, when combined with B^ε-trees. Recursive greps run 3.8x faster than in the best standard file system. File creation runs 2.6x faster. Small, random writes to a file run 68.2x faster.

However, renaming a directory has predictably miserable performance [22, 23]. For example, renaming the Linux source tree, which must delete and reinsert all the data to preserve locality, takes 21.2s in BetrFS 0.1, as compared to 0.1s in btrfs.

Relative-path indexing. BetrFS 0.2 backed away from full-path indexing and introduced zoning [54, 55]. Zoning is a schema-level change that implements ***relative-***

path indexing. In relative-path indexing, each file or directory is indexed relative to a local neighborhood in the directory tree. See Section 2.2 for details.

Zoning strikes a "sweet spot" on the spectrum between indirection and full-path indexing. Large file and directory renames are comparable to indirection-based file systems, and a sequential scan is at least 2x faster than inode-based file systems but 1.5x slower than BetrFS 0.1.

There are, however, a number of significant, diffuse costs to relative-path indexing, which tax the performance of seemingly unrelated operations. For instance, two-thirds of the way through the TokuBench benchmark, BetrFS 0.2 shows a sudden, precipitous drop in cumulative throughput for small file creations, which can be attributed to the cost of maintaining zones.

Perhaps the most intangible cost of zoning is that it introduces complexity into the system and thus hides optimization opportunities. In a full-path file system, one can implement nearly all file system operations as simple point or range operations. Adding indirection breaks this simple mapping. Indirection generally causes file system operations to map onto more key/value operations and often introduces reads before writes. Because reads are slower than writes in a write-optimized file system, making writes depend upon reads forgoes some of the potential performance benefits of write-optimization.

Consider `rm -r`, for example. With full-path indexing, one can implement this operation with a single range-delete message, which incurs almost no latency and requires a single synchronous write to become persistent [54, 55]. Using a single range-delete message also unlocks optimizations internal to the key/value store, such as freeing a dead leaf without first reading it from disk. Adding indirection on some directories (as in BetrFS 0.2) requires a recursive delete to scatter reads and writes throughout the keyspace and disk (Table 3).

Our contributions. This paper presents a B^ε-tree variant, called a ***lifted B^ε-tree***, that can efficiently rename a range of lexicographically ordered keys, unlocking the benefits of full-path indexing. We demonstrate the benefits of a lifted B^ε-tree in combination with full-path indexing in a new version of BetrFS, version 0.4, which achieves:

- fast updates of data and metadata,
- fast scans of the data and metadata in directory subtrees and fast scans of files,
- fast renames, and
- fast subtree operations, such as recursive deletes.

We introduce a new key/value primitive called ***range rename***. Range renames are the keyspace analogue of directory renames. Given two strings, p_1 and p_2, range rename replaces prefix p_1 with prefix p_2 in all keys that have p_1 as a prefix. Range rename is an atomic modification to a contiguous range of keys, and the values are unchanged. Our main technical innovation is an efficient implementation of range rename in a B^ε-tree. Specifically, we reduce the cost from the size of the subtree to the height of the subtree.

Using range rename, BetrFS 0.4 returns to a simple schema for mapping file system operations onto key/value operations; this in turn consolidates all placement decisions and locality optimizations in one place. The result is simpler code with less ancillary metadata to maintain, leading to better performance on a range of seemingly unrelated operations.

The technical insight behind efficient B^ε-tree range rename is a method for performing large renames by direct manipulation of the B^ε-tree. Zoning shows us that small key ranges can be deleted and reinserted cheaply. For large key ranges, range rename is implemented by slicing the tree at the source and destination. Once the source subtree is isolated, a pointer swing moves the renamed section of keyspace to its destination. The asymptotic cost of such tree surgery is proportional to the height, rather than the size, of the tree.

Once the B^ε-tree has its new structure, another challenge is efficiently changing the pivots and keys to their new values. In a standard B^ε-tree, each node stores the full path keys; thus, a straightforward implementation of range rename must rewrite the entire subtree.

We present a method that reduces the work of updating keys by removing the redundancy in prefixes shared by many keys. This approach is called ***key lifting*** (§5). A lifted B^ε-tree encodes its pivots and keys such that the values of these strings are defined by the path taken to reach the node containing the string. Using this approach, the number of paths that need to be modified in a range rename also changes from being proportional to the size of the subtree to the depth of the subtree.

Our evaluation shows improvement across a range of workloads. For instance, BetrFS 0.4 performs recursive greps 1.5x faster and random writes 1.2x faster than BetrFS 0.3, but renames are competitive with standard, indirection-based file systems. As an example of simplicity unlocked by full path indexing, BetrFS 0.4 implements recursive deletion with a single range delete, significantly out-performing other file systems.

2 Background

This section presents background on BetrFS, relevant to the proposed changes to support efficient keyspace mutations. Additional aspects of the design are covered elsewhere [7, 22, 23, 54, 55].

2.1 B^ε-Tree Overview

The B^ε-tree is a ***write-optimized*** B-tree variant that implements the standard key/value store interface: in-

sert, delete, point query, and predecessor and successor queries (i.e., range queries). By write-optimized, we mean the insertions and deletions in B^ε-trees are orders of magnitude faster than in a B-tree, while point queries are just as fast as in a B-tree. Furthermore, range queries and sequential insertions and deletions in B^ε-trees can run at near disk bandwidth.

Because insertions are much faster than queries, the common read-modify-write pattern can become bottlenecked on the read. Therefore, B^ε-trees provide an ***upsert*** that logically encodes, but lazily applies a read-modify-write of a key-value pair. Thus, upserts are as fast as inserts.

Like B-trees, B^ε-trees store key/value pairs in nodes, where they are sorted by key order. Also like B-trees, interior nodes store pointers to children, and ***pivot keys*** delimit the range of keys in each child.

The main distinction between B^ε-trees and B-trees is that interior B^ε-tree nodes are augmented to include ***message buffers***. A B^ε-tree models all changes (inserts, deletes, upserts) as ***messages***. Insertions, deletions, and upserts are implemented by inserting messages into the buffers, starting with the root node. A key technique behind write-optimization is that messages can accumulate in a buffer, and are flushed down the tree in larger batches, which amortize the costs of rewriting a node. Most notably, this batching can improve the costs of small, random writes by orders of magnitude.

Since messages lazily propagate down the tree, queries may require traversing the entire root-to-leaf search path, checking for relevant messages in each buffer along the way. The newest target value is returned (after applying pending upsert messages, which encode key/value pair modifications).

In practice, B^ε-trees are often configured with large nodes (typically $\geq$4MiB) and fanouts (typically $\leq$16) to improve performance. Large nodes mean updates are applied in large batches, but large nodes also mean that many contiguous key/value pairs are read per I/O. Thus, range queries can run a near disk bandwidth, with at most one random I/O per large node.

The B^ε-tree implementation in BetrFS supports both ***point*** and ***range*** messages; range messages were introduced in v0.2 [54]. A point message is addressed to a single key, whereas a range message is applied to a contiguous range of keys. Thus far, range messages have only been used for deleting a range of contiguous keys with a single message. In our experience, range deletes give useful information about the keyspace that is hard to infer from a series of point deletions, such as dropping obviated insert and upsert messages.

The B^ε-tree used in BetrFS supports transactions and crash consistency as follows. The B^ε-tree internally uses a logical timestamp for each message and MVCC to implement transactions. Pending messages can be thought of as a history of recent modifications, and, at any point in the history, one can construct a consistent view of the data. Crash consistency is ensured using logical logging, i.e., by logging the inserts, deletes, etc, performed on the tree. Internal operations, such as node splits, flushes, etc, are not logged. Nodes are written to disk using copy-on-write. At a periodic checkpoint (every 5 seconds), all dirty nodes are written to disk and the log can be trimmed. Any unreachable nodes are then garbage collected and reused. Crash recovery starts from the last checkpoint, replays the logical redo log, and garbage collects any unreachable nodes; as long as an operation is in the logical log, it will be recoverable.

2.2 BetrFS Overview

BetrFS translates VFS-level operations into B^ε-tree operations. Across versions, BetrFS has explored schema designs that map VFS-level operations onto B^ε-tree operations as efficiently as possible.

All versions of BetrFS use two B^ε-trees: one for file data and one for file system metadata. The B^ε-tree implementation supports transactions, which we use internally for operations that require more than one message. BetrFS does not expose transactions to applications, which introduce some more complex issues around system call semantics [25, 35, 39, 46].

In BetrFS 0.1, the metadata B^ε-tree maps a full path onto the typical contents of a `stat` structure, including owner, modification time, and permission bits. The data B^ε-tree maps keys of the form (p, i), where p is the full path to a file and i is a block number within that file, to 4KB file blocks. Paths are sorted in a variant of depth-first traversal order.

This full-path schema means that entire sub-trees of the directory hierarchy are stored in logically contiguous ranges of the key space. For instance, this design enabled BetrFS 0.1 to perform very fast recursive directory traversals (e.g. `find` or recursive `grep`).

Unfortunately, with this schema, file and directory renames do not map easily onto key/value store operations. In BetrFS 0.1, file and directory renames were implemented by copying the file or directory from its old location to the new location. As a result, renames were orders of magnitude slower than conventional file systems.

BetrFS 0.2 improved rename performance by replacing the full-path indexing schema of BetrFS 0.1 with a ***relative-path*** indexing schema [54, 55]. The goal of relative path indexing is to get the rename performance of inode-based file systems and the recursive-directory-traversal performance of a full-path indexing file system.

BetrFS 0.2 accomplishes this by partitioning the directory hierarchy into a collection of connected regions called ***zones***. Each zone has a single root file or directory

and, if the root of a zone is a directory, it may contain sub-directories of that directory. Each zone is given a zone ID (analogous to an inode number).

Relative-path indexing made renames on BetrFS 0.2 almost as fast as inode-based file systems and recursive-directory traversals almost as fast as BetrFS 0.1.

However, our experience has been that relative-path indexing introduces a number of overheads and precludes other opportunities for mapping file-system-level operations onto B^ε-tree operations. For instance, must be split and merged to keep all zones within a target size range. These overheads became a first-order performance issue, for example, the Tokubench benchmark results for BetrFS 0.2.

Furthermore, relative-path indexing also has bad worst-case performance. It is possible to construct arrangements of nested directories that will each reside in their own zone. Reading a file in the deepest directory will require reading one zone per directory (each with its own I/O). Such a pathological worst case is not possible with full-path indexing in a B^ε-tree, and an important design goal for BetrFS is keeping a reasonable bound on the worst cases.

Finally, zones break the clean mapping of directory subtrees onto contiguous ranges of the key space, preventing us from using range-messages to implement bulk operations on entire subtrees of the directory hierarchy. For example, with full-path indexing, we can use range-delete messages not only to delete files, but entire subtrees of the directory hierarchy. We could also use range messages to perform a variety of other operations on subtrees of the directory hierarchy, such as recursive `chmod`, `chown`, and timestamp updates.

The goal of this paper is to show that, by making rename a first-class key/value store operation, we can use full-path indexing to produce a simpler, more efficient, and more flexible system end-to-end.

3 Overview

The goal of this section is to explain the performance considerations behind our data structure design, and to provide a high-level overview of that design.

Our high-level strategy is to simply copy small files and directories in order to preserve locality—i.e., copying a few-byte file is no more expensive than updating a pointer. Once a file or directory becomes sufficiently large, copying the data becomes expensive and of diminishing value (i.e., the cost of indirection is amortized over more data). Thus, most of what follows is focused on efficient rename of *large* files and directories, large meaning at least as large as a B^ε-tree node.

Since we index file and directory data and metadata by full path, a file or directory rename translates into a prefix replacement on a *contiguous* range of keys. For example, if we rename directory `/tmp/draft` to `/home/paper/final`, then we want to find all keys in the B^ε-tree that begin with `/tmp/draft` and replace that prefix with `/home/paper/final`. This involves both updating the key itself, and updating its location in the B^ε-tree so that future searches can find it.

Since the affected keys form a contiguous range in the B^ε-tree, we can move the keys to their new (logical) home without moving them physically. Rather, we can make a small number of pointer updates and other changes to the tree. We call this step ***tree surgery***. We then need to update all the keys to contain their new prefix, a process we call ***batched key update***.

In summary, the algorithm has two high-level steps:

Tree Surgery. We identify a subtree of the B^ε-tree that includes all keys in the range to be renamed (Figure 1). Any ***fringe*** nodes (i.e., on the left and right extremes of the subtree), which contain both related and unrelated keys, are split into two nodes: one containing only affected keys and another containing only un-affected keys. The number of fringe nodes will be at most logarithmic in the size of the sub-tree. At the end of the process, we will have a subtree that contains only keys in the range being moved. We then change the pivot keys and pointers to move the subtree to its new parent.

Batched Key Updates. Once a subtree has been logically renamed, full-path keys in the subtree will still reflect the original key range. We propose a B^ε-tree modification to make these key updates efficient. Specifically, we modify the B^ε-tree to factor out common prefixes from keys in a node, similar to prefix-encoded compression. We call this transformation ***key lifting***. This transformation does not lose any information—the common prefix of keys in a node can be inferred from the pivot keys along the path from the root to the node by concatenating the longest common prefix of enclosing pivots along the path. As a result of key lifting, once we perform tree surgery to isolate the range of keys affected by a rename, the prefix to be replaced in each key will already be removed from every key in the sub-tree. Furthermore, since the omitted prefixes are inferred from the sub-tree's parent pivots, moving the sub-tree to its new parent implicitly replaces the old prefix with the new one. Thus a large subtree can be left untouched on disk during a range rename. In the worst case, only a logarithmic number of nodes on the fringe of the subtree will have keys that need to be updated.

Buffered Messages and Concurrency. Our range move operation must also handle any pending messages targeting the affected keys. These messages may be buffered in any node along a search path from the root to one of the affected keys. Our solution leverages the fact that messages have a logical timestamp and are ap-

plied in logical order. Thus, it is sufficient to ensure that pending messages for a to-be-renamed subtree must be flushed into the subtree before we begin the tree surgery for a range rename.

Note that most of the work in tree surgery involves node splits and merges, which are part of normal B^ε-tree operation. Thus the tree remains a "valid" B^ε-tree during this phase of the range move. Only the pointer swaps need to be serialized with other operations. Thus this approach does not present a concurrency bottleneck.

The following two sections explain tree surgery and lifting in more detail.

4 Tree Surgery

This section describes our approach to renaming a directory or large file via changes within the B^ε-tree, such that most of the data is not physically moved on disk. Files that are smaller than 4MiB reside in at most two leaves. We therefore move them by copying and perform tree surgery only on larger files and, for simplicity of the prototype, directories of any size.

For the sake of discussion, we assume that a rename is moving a source file over an existing destination file; the process would work similarly (but avoid some work) in the case where the destination file does not exist. Our implementation respects POSIX restrictions for directories (i.e., you cannot rename over a non-empty destination directory), but our technique could easily support different directory rename semantics. In the case of renaming over a file, where a rename implicitly deletes the destination file, we use transactions in the B^ε-tree to insert both a range delete of the destination and a range rename of the source; these messages are applied atomically.

This section also operates primarily at the B^ε-tree level, not the directory namespace. Unless otherwise noted, pointers are pointers within the B^ε-tree.

In renaming a file, the goal is to capture a range of contiguous keys and logically move these key/value pairs to a different point in the tree. For anything large enough to warrant using this rename approach, some B^ε-tree nodes will exclusively store messages or key/value pairs for the source or destination, and some may include unrelated messages or key/value pairs before and after in sort order, corresponding to the left and right in the tree.

An important abstraction for tree surgery is the ***Lowest Common Ancestor***, or ***(LCA)***, of two keys: the B^ε-tree node lowest in the tree on the search path for both keys (and hence including all keys in between). During a rename, the source and destination will each have an LCA, and they may have the same LCA.

The first step in tree surgery is to find the source LCA and destination LCA. In the process of identifying the LCAs, we also flush any pending messages for the source or destination key range, so that they are buffered at or below the corresponding LCAs.

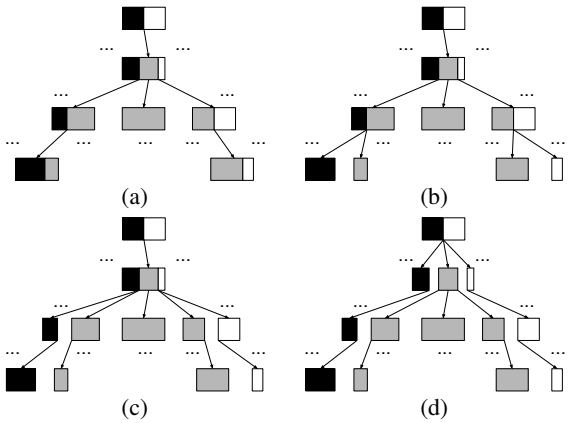


Figure 1: Slicing /gray between /black and /white.

Slicing. The second step is to slice out the source and destination from any shared nodes. The goal of slicing is to separate unrelated key-value pairs that are not being moved but are packed into the same B^ε-tree node as key-value pairs that are being moved. Slicing uses the same code used for standard B^ε-tree node splits, but slicing divides the node at the slicing key rather than picking a key in the middle of the node. As a result, slicing may result in nodes that temporarily violate constraints on target node size and fanout. However, these are performance, not correctness, constraints, so we can let queries continue concurrently, and we restore these invariants before completing the rename.

Figure 1 depicts slicing the sub-tree containing all gray keys from a tree with black, gray, and white keys. The top node is the parent of the LCA. Because messages have been flushed to the LCA, the parent of the LCA contains no messages related to gray. Slicing proceeds up the tree from the leaves, and only operates on the left and right fringe of the gray sub-tree. Essentially, each fringe node is split into two smaller B^ε-tree nodes (see steps b and c). All splits, as well as later transplanting and healing, happen when the nodes are pinned in memory. During surgery, they are dirtied and written at the next checkpoint. Eventually, the left and right edge of an exclusively-gray subtree (step d) is pinned, whereas interior, all-grey nodes may remain on disk.

Our implementation requires that the source and destination LCA be at the same height for the next step. Thus, if the LCAs are not at the same level of the tree, we slice up to an ancestor of the higher LCA. The goal of this choice is to maintain the invariant that all B^ε-tree leaves be at the same depth.

Transplant. Once the source and destination are both sliced, we then swap the pointers to each sub-tree in the respective parents of LCAs. We then insert a range-delete message at the source (which now points to a sub-

tree containing all the data in the file that used to reside at the destination of the rename). The B^ε-tree's builtin garbage collection will reclaim these nodes.

Healing. Our B^ε-tree implementation maintains the invariant that all internal nodes have between 4 and 16 children, which bounds the height of the tree. After the transplant completes, however, there may be a number of in-memory B^ε-tree nodes at the fringe around the source and destination that have fewer than 4 children.

We handle this situation by triggering a rebalancing within the tree. Specifically, if a node has only one child, the slicing process will merge it after completing the work of the rename.

Crash Consistency. In general, BetrFS ensures crash consistency by keeping a redo log of pending messages and applying messages to nodes copy-on-write. At periodic intervals, BetrFS ensures that there is a consistent checkpoint of the tree on disk. Crash recovery simply replays the redo log since the last checkpoint. Range rename works within this framework.

A range rename is *logically applied* and persisted as soon as the message is inserted into the root buffer and the redo log. If the system crashes after a range rename is logged, the recovery will see a prefix of the message history that includes the range rename, and it will be logically applied to any queries for the affected range.

Tree surgery occurs when a range rename message is flushed to a node that is likely an LCA. Until surgery completes, all fringe nodes, the LCAs, and the parents of LCAs are pinned in memory and dirtied. Upon completion, these nodes will be unpinned and written to disk, copy-on-write, no later than the next B^ε-tree checkpoint.

If the system crashes after tree surgery begins but before surgery completes, the recovery code will see a consistent checkpoint of the tree as it was before the tree surgery. The same is true if the system crashes after tree surgery but before the next checkpoint (as these post-surgery nodes will not be reachable from the checkpoint root). Because a B^ε-tree checkpoint flushes all dirty nodes, if the system crashes after a B^ε-tree checkpoint, all nodes affected by tree surgery will be on disk.

At the file system level, BetrFS has similar crash consistency semantics to metadata-only journaling in ext4. The B^ε-tree implementation itself implements full data journaling [54, 55], but BetrFS allows file writes to be buffered in the VFS, weakening this guarantee end-to-end. Specifically, file writes may be buffered in the VFS caches, and are only logged in the recovery journal once the VFS writes back a dirty page (e.g., upon an `fsync` or after a configurable period). Changes to the directory tree structure, such as a `rename` or `mkdir` are persisted to the log immediately. Thus, in the common pattern of writing to a temporary file and then renaming it, it is possible for the rename to appear in the log before the writes. In this situation and in the absence of a crash, the writes will eventually be logged with the correct, renamed key, as the in-memory inode will be up-to-date with the correct B^ε-tree key. If the system crashes, these writes can be lost; as with a metadata-journaled file system, the developer must issue an `fsync` before the `rename` to ensure the data is on disk.

Latency. A rename returns to the user once a log entry is in the journal and the root of the B^ε-trees are locked. At this point, the rename has been applied in the VFS to in-memory metadata, and as soon as the log is fsynced, the rename is durable.

We then hand off the rest of the rename work to two background threads to do the cutting and healing. The prototype in this paper only allows a backlog of one pending, large rename, since we believe that concurrent renames are relatively infrequent. The challenge in adding a rename work queue is ensuring consistency between the work queue and the state of the tree.

Atomicity and Transactions. The B^ε-tree in BetrFS implements multi-version concurrency control by augmenting messages with a logical timestamp. Messages updating a given key range are always applied in logical order. Multiple messages can share a timestamp, giving them transactional semantics.

To ensure atomicity for a range rename, we create an MVCC "hazard": read transactions "before" the rename must complete before the surgery can proceed. Tree nodes in BetrFS are locked with reader-writer locks. We write-lock tree nodes hand-over-hand, and left-to-right to identify the LCAs. Once the LCAs are locked, this serializes any new read or write transactions until the rename completes. The lock at the LCA creates a "barrier"—operations can complete "above" or "below" this lock in the tree, although the slicing will wait for concurrent transactions to complete before write-locking that node. Once the transplant completes, the write-locks on the parents above LCAs are released.

For simplicity, we also ensure that all messages in the affected key range(s) that logically occur before the range rename are flushed below the LCA before the range rename is applied. All messages that logically occur after the rename follow the new pointer path to the destination or source. This strategy ensures that, when each message is flushed and applied, it sees a point-in-time consistent view of the subtree.

Complexity. At most 4 slices are performed, each from the root to a leaf, dirtying nodes from the LCA along the slicing path. These nodes will need to be read, if not in cache, and written back to disk as part of the checkpointing process. Therefore the number of I/Os is at most proportional to the height of the B^ε-tree, which is logarithmic in the size of the tree.

5 Batched Key Updates

After tree-surgery completes, there will be a subtree where the keys are not coherent with the new location in the tree. As part of a rename, the prefixes of all keys in this subtree need to be updated. For example, suppose we execute `mv /foo /bar`. After surgery, any messages and key/value pairs for file `/foo/bas` will still have a key that starts with `/foo`. These keys need to be changed to begin with `/bar`. The particularly concerning case is when `/foo` is a very large subtree and has interior nodes that would otherwise be untouched by the tree surgery step; our goal is to leave these nodes untouched as part of rename, and thus reduce the cost of key changes from the size of the rename tree to the height of the rename tree.

We note that keys in our tree are highly redundant. Our solution reduces the work of changing keys by reducing the redundancy of how keys are encoded in the tree. Consider the prefix encoding for a sequence of strings. In this compression method, if two strings share a substantial longest common prefix (lcp), then that lcp is only stored once. We apply this idea to B$^\varepsilon$-trees. The lcp of all keys in a subtree is removed from the keys and stored in the subtree's parent node. We call this approach ***key lifting*** or simply ***lifting*** for short.

At a high level, our lifted B$^\varepsilon$-tree stores a node's common, lifted key prefix in the node's parent, alongside the parent's pointer to the child node. Child nodes only store differing key suffixes. This approach encodes the complete key in the path taken to reach a given node.

Lifting requires a schema-level invariant that keys with a common prefix are adjacent in the sort order. As a simple example, if one uses `memcmp` to compare keys (as BetrFS does), then lifting will be correct. This invariant ensures that, if there is a common prefix between any two pivot keys, all keys in that child will have the same prefix, which can be safely lifted. More formally:

Invariant 1 *Let T' be a subtree in a B$^\varepsilon$-tree with full-path indexing. Let p and q be the pivots that enclose T'. That is, if T' is not the first or last child of its parent, then p and q are the enclosing pivots in the parent of T'. If T' is the first child of its parent, then q is the first pivot and p is the left enclosing pivot of the parent of T'.*

Let s be the longest common prefix of p and q. Then all keys in T' begin with s.

Given this invariant, we can strip s from the beginning of every message or key/value pair in T', only storing the non-lifted suffix. Lifting is illustrated in Figure 2, where the common prefix in the first child is "/b/", which is removed from all keys in the node and its children (indicated with strikethrough text). The common prefix (indicated with purple) is stored in the parent. As one moves toward leaves, the common prefix typically becomes longer ("/b/4/" in Figure 2), and each level of the tree can lift the additional common prefix.

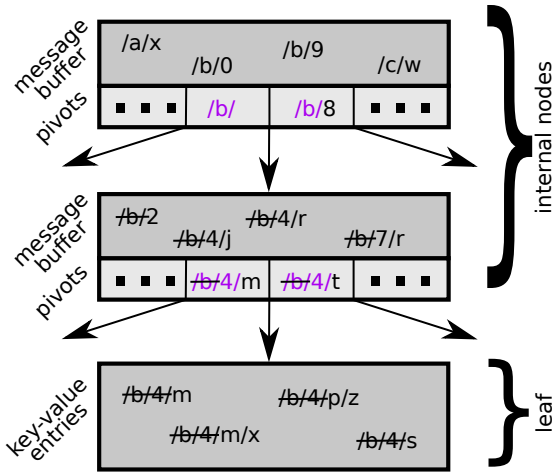


Figure 2: Example nodes in a lifted B$^\varepsilon$-tree. Since the middle node is bounded by two pivots with common prefix "/b/" (indicated by purple text), all keys in the middle node and its descendants must have this prefix in common. Thus this prefix can be omitted from all keys in the middle node (and all its descendants), as indicated by the strike-through text. Similarly, the bottom node (a leaf) is bounded by pivots with common prefix "/b/4/", so this prefix is omitted from all its keys.

Reads can reconstruct the full key by concatenating prefixes during a root-to-leaf traversal. In principle, one need not store the lifted prefix (s) in the tree, as it can be computed from the pivot keys. In our implementation, we do memoize the lifted prefix for efficiency.

As messages are flushed to a child, they are modified to remove the common prefix. Similarly, node splits and merges ensure that any common prefix between the pivot keys is lifted out. It is possible for all of the keys in T' to share a common prefix that is longer than s, but we only lift s because maintaining this amount of lifting hits a sweet spot: it is enough to guarantee fast key updates during renames, but it requires only local information at a parent and child during splits, merges, and insertions.

Lifting is completely transparent to the file system. From the file system's perspective, it is still indexing data with a key/value store that is keyed by full-path; the only difference from the file system's perspective is that the key/value store completes some operations faster.

Lifting and Renames. In the case of renames, lifting dramatically reduces the work to update keys. During a rename from a to b, we slice out a sub-tree containing exactly those keys that have a as a prefix. By the lifting invariant, the prefix a will be lifted out of the sub-tree, and the parent of the sub-tree will bound it between two pivots whose common prefix is a (or at least includes a—the pivots may have an even longer common pre-

fix). After we perform the pointer swing, the sub-tree will be bounded in its new parent by pivots that have b as a common prefix. Thus, by the lifting invariant, all future queries will interpret all the keys in the sub-tree has having b as a prefix. Thus, with lifting, the pointer swing implicitly performs the batch key-prefix replacement, completing the rename.

Complexity. During tree surgery, there is lifting work along all nodes that are sliced or merged. However, the number of such nodes is at most proportional to the height of the tree. Thus, the number of nodes that must be lifted after a rename is no more than the nodes that must be sliced during tree surgery, and proportional to the height of the tree.

6 Implementation Details

Simplifying key comparison. One small difference in the BetrFS 0.4 and BetrFS 0.3 key schemas is that BetrFS 0.4 adjusted the key format so that `memcmp` is sufficient for key comparison. We found that this change simplified the code, especially around lifting, and helped CPU utilization, as it is hard to compare bytes faster than a well-tuned `memcmp`.

Zone maintenance. A major source of overheads in BetrFS 0.3 is tracking metadata associated with zones. Each update involves updating significant in-memory bookkeeping; splitting and merging zones can also be a significant source of overhead (c.f., Figure 3). BetrFS 0.4 was able to delete zone maintenance code, consolidating this into the B^ε-tree's internal block management code.

Hard Links. BetrFS 0.4 does not support hard links. In future work, for large files, sharing sub-trees could also be used to implement hard links. For small files, zones could be reintroduced solely for hard links.

7 Evaluation

Our evaluation seeks to answer the following questions:

- (§7.1) Does full-path indexing in BetrFS 0.4 improve overall file system performance, aside from renames?
- (§7.2) Are rename costs acceptable in BetrFS 0.4?
- (§7.3) What other opportunities does full-path indexing in BetrFS 0.4 unlock?
- (§7.4) How does BetrFS 0.4 performance on application benchmarks compare to other file systems?

We compare BetrFS 0.4 with several file systems, including BetrFS 0.3 [14], Btrfs [42], ext4 [31], nilfs2 [34], XFS [47], and ZFS [8]. Each file system's block size is 4096 bytes. We use the versions of XFS, Btrfs, ext4 that are part of the 3.11.10 kernel, and ZFS 0.6.5.11, downloaded from `www.zfsonlinux.org`. We use default recommended file system settings unless otherwise noted. For ext4 (and BetrFS), we disabled lazy inode table and journal initialization, as these features accelerate file system creation but slow down some operations on a freshly-created file system; we believe this configuration yields more representative measurements of the file system in steady-state. Each experiment was run a minimum of 4 times. Error bars indicate minimum and maximum times over all runs. Similarly, error $\pm$ terms bound minimum and maximum times over all runs. Unless noted, all benchmarks are cold-cache tests and finish with a file-system sync. For BetrFS 0.3, we use the default zone size of 512 KiB.

In general, we expect BetrFS 0.3 to be the closest competitor to BetrFS 0.4, and focus on this comparison but include other file systems for context. Relative-path indexing is supposed to get most of the benefits of full-path indexing, with affordable renames; comparing BetrFS 0.4 with BetrFS 0.3 shows the cost of relative-path indexing and the benefit of full-path indexing.

All experimental results were collected on a Dell Optiplex 790 with a 4-core 3.40 GHz Intel Core i7 CPU, 4 GB RAM, and a 500 GB, 7200 RPM SATA disk, with a 4096-byte block size. The system runs Ubuntu 14.04.5, 64-bit, with Linux kernel version 3.11.10. We boot from a USB stick with the root file system, isolating the file system under test to only the workload.

7.1 Non-Rename Microbenchmarks

Tokubench. The tokubench benchmark creates three million 200-byte files in a balanced directory tree, where no directory is allowed to have more than 128 children.

As Figure 3 shows, zone maintenance in BetrFS 0.3 causes a significant performance drop around 2 million files. This drop occurs all at once because, at that point in the benchmark, all the top-level directories are just under the zone size limit. As a result, the benchmark goes through a period where each new file causes its top-level directory to split into its own zone. If we continue the benchmark long enough, we would see this happen again when the second-level directories reach the zone-size limit. In experiments with very long runs of Tokubench, BetrFS 0.3 never recovers this performance.

With our new rename implementations, zone maintenance overheads are eliminated. As a result, BetrFS 0.4 has no sudden drop in performance. Only nilfs2 comes close to matching BetrFS 0.4 on this benchmark, in part because nilfs2 is a log-structured file system. BetrFS 0.4 has over 80$\times$ higher cumulative throughput than ext4 throughout the benchmark.

Recursive directory traversals. In these benchmarks, we run `find` and recursive `grep` on a copy of the Linux kernel 3.11.10 source tree. The times taken for these operations are given in Table 1. BetrFS 0.4 outperforms BetrFS 0.3 by about 5% on find and almost 30% on grep. In the case of grep, for instance, we found that roughly the same total number of bytes were read from disk in both

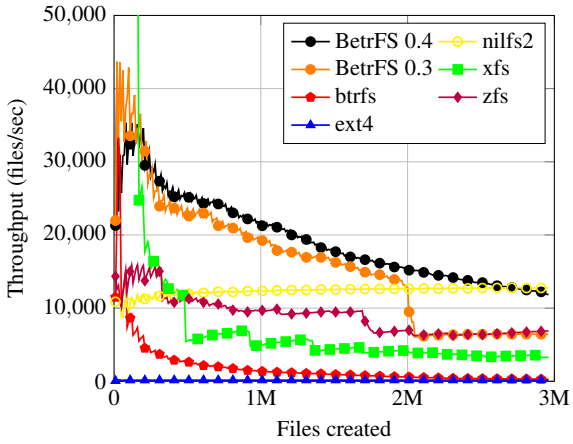


Figure 3: Cumulative file creation throughput during the Tokubench benchmark (higher is better). BetrFS 0.4 outperforms other file systems by orders of magnitude and avoids the performance drop that BetrFS 0.3 experience due to its zone-maintenance overhead.

File system	find (sec)	grep (sec)
BetrFS 0.4	0.233 ± 0.0	3.834 ± 0.2
BetrFS 0.3	0.247 ± 0.0	5.859 ± 0.1
btrfs	1.311 ± 0.1	8.068 ± 1.6
ext4	2.333 ± 0.1	42.526 ± 5.2
xfs	6.542 ± 0.4	58.040 ± 12.2
zfs	9.797 ± 0.9	346.904 ± 101.5
nilfs2	6.841 ± 0.1	8.399 ± 0.2

Table 1: Time to perform recursive directory traversals of the Linux 3.11.10 source tree (lower is better). BetrFS 0.4 is significantly faster than every other file system, demonstrating the locality benefits of full-path indexing.

versions of BetrFS, but that BetrFS 0.3 issued roughly 25% more I/O transactions. For this workload, we also saw higher disk utilization in BetrFS 0.4 (40 MB/s vs. 25 MB/s), with fewer worker threads needed to drive the I/O. Lifting also reduces the system time by 5% on grep, but the primary savings are on I/Os. In other words, this demonstrates the locality improvements of full-path indexing over relative-path indexing. BetrFS 0.4 is anywhere from 2 to almost 100 times faster than conventional file systems on these benchmarks.

Sequential IO. Figure 4 shows the throughput of sequential reads and writes of a 10GiB file (more than twice the size of the machine's RAM). All file systems measured, except ZFS, are above 100 MB/s, and the disk's raw read and write bandwidth is 132 MB/s.

Sequential reads in BetrFS 0.4 are essentially identical to those in BetrFS 0.3 and roughly competitive with other file systems. Both versions of BetrFS do not realize the full performance of the disk on sequential I/O, leaving up to 20% of the throughput compared to ext4 or Btrfs. This

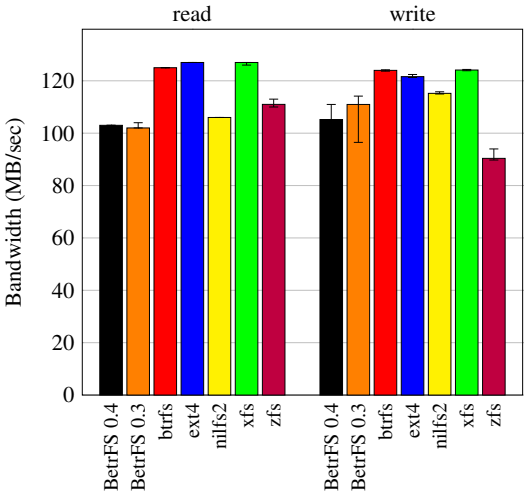


Figure 4: Sequential IO bandwidth (higher is better). BetrFS 0.4 performs sequential IO at over 100 MB/s but up to 19% slower than the fastest competitor. Lifting introduces some overheads on sequential writes.

is inherited from previous versions of BetrFS and does not appear to be significantly affected by range rename. Profiling indicates that there is not a single culprit for this loss but several cases where writeback of dirty blocks could be better tuned to keep the disk fully utilized. This issue has improved over time since version 0.1, but in small increments.

Writes in BetrFS 0.4 are about 5% slower than in BetrFS 0.3. Profiling indicates this is because node splitting incurs additional computational costs to re-lift a split child. We believe this can be addressed in future work by either better overlapping computation with I/O or integrating key compression with the on-disk layout, so that lifting a leaf involves less memory copying.

Random writes. Table 2 shows the execution time of a microbenchmark that issues 256K 4-byte overwrites at random offsets within a 10GiB file, followed by an `fsync`. This is 1 MiB of total data written, sized to run for at least several seconds on the fastest file system. BetrFS 0.4 performs small random writes approximately 400 to 650 times faster than conventional file systems and about 19% faster than BetrFS 0.3.

Summary. These benchmarks show that lifting and full-path indexing can improve performance over relative-path indexing for both reads and writes, from 5% up to 2×. The only case harmed is sequential writes. In short, lifting is generally more efficient than zone maintenance in BetrFS.

7.2 Rename Microbenchmarks

Rename performance as a function of file size. We evaluate rename performance by renaming files of different sizes and measuring the throughput. For each file

File system	random write (sec)	
BetrFS 0.4	4.9	± 0.3
BetrFS 0.3	5.9	± 0.1
btrfs	2147.5	± 7.4
ext4	2776.0	± 40.2
xfs	2835.7	± 7.9
zfs	3288.9	± 394.7
nilfs2	2013.1	± 19.1

Table 2: Time to perform 256K 4-byte random writes (1 MiB total writes, lower is better). BetrFS 0.4 is up to 600 times faster than other file systems on random writes.

size, we rename a file of this size 100 times within a directory and `fsync` the parent directory to ensure that the file is persisted on disk. We measure the average across 100 runs and report this as throughput, in Figure 5a.

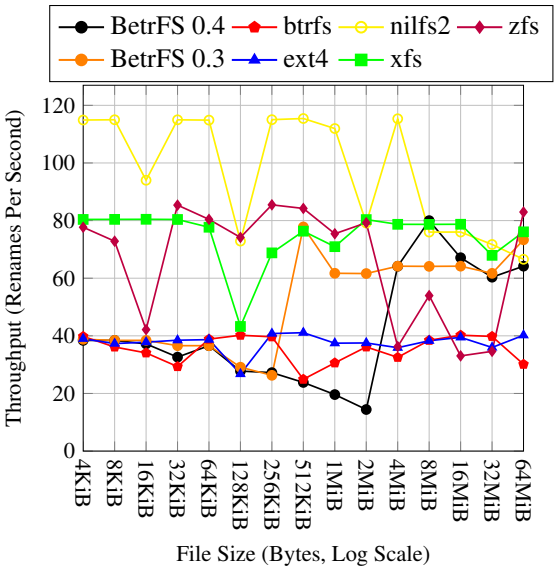
In both BetrFS 0.3 and BetrFS 0.4, there are two modes of operation. For smaller objects, both versions of BetrFS simply copy the data. At 512 KiB and 4 MiB, BetrFS 0.3 and BetrFS 0.4, respectively, switch modes—this is commensurate with the file matching the zone size limit and node size, respectively. For files above these sizes, both file systems see comparable throughput of simply doing a pointer swing.

More generally, the rename throughput of all of these file systems is somewhat noisy, but ranges from 30–120 renames per second, with nilfs2 being the fastest. Both variants of BetrFS are within this range, except when a rename approaches the node size in BetrFS 0.4.

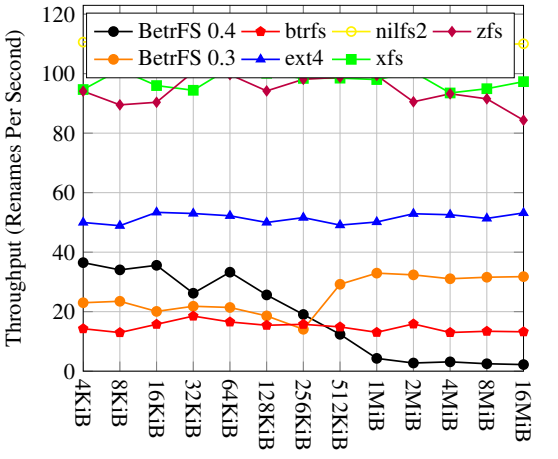
Figure 5b shows rename performance in a setup carefully designed to incur the worst-case tree surgery costs in BetrFS 0.4. In this experiment, we create two directories, each with 1000 files of the given size. The benchmark renames the interleaved files from the source directory to the destination directory, so that they are also interleaved with the files of the given size in the destination directory. Thus, when the interleaved files are 4MB or larger, every rename requires two slices at both the source and destination directories. We fsync after each rename.

Performance is roughly comparable to the previous experiment for small files. For large files, this experiment shows the worst-case costs of performing four slices. Further, all slices will operate on different leaves.

Although this benchmark demonstrates that rename performance has potential for improvement in some carefully constructed worst-case scenarios, the cost of renames in BetrFS 0.4 is nonetheless bounded to an average cost of 454ms. We also note that this line flattens, as the slicing overheads grow logarithmically in the size of the renamed file. In contrast, renames in BetrFS 0.1 were unboundedly expensive, easily getting into minutes; bounding this worst case is significant progress for



(a) Rename throughput as a function of file size. This experiment was performed in the base directory of an otherwise empty file system.



(b) Rename throughput as a function of file size. This experiment interleaves the renamed files with other files of the same size in both the source and destination directories.

Figure 5: Rename throughput

the design of full-path-indexed file systems.

7.3 Full-path performance opportunities

As a simple example of other opportunities for full-path indexing, consider deleting an entire directory (e.g., `rm -rf`). POSIX semantics require checking permission to delete all contents, bringing all associated metadata into memory. Other directory deletion semantics have been proposed. For example, HiStar allows an untrusted administrator to *delete* a user's directory but not *read* the contents [56].

We implemented a system call that uses range-delete messages to delete an entire directory sub-tree. This

File system	recursive delete (sec)	
BetrFS 0.4 (range delete)	0.053 ±	0.001
BetrFS 0.4	3.351 ±	0.5
BetrFS 0.3	2.711 ±	0.3
btrfs	2.762 ±	0.1
ext4	3.693 ±	2.2
xfs	7.971 ±	0.8
zfs	11.492 ±	0.1
nilfs2	9.472 ±	0.3

Table 3: Time to delete the Linux 3.11.10 source tree (lower is better). Full-path indexing in BetrFS 0.4 can remove a subtree in a single range delete, orders-of-magnitude faster than the recursive strategy of `rm -rf`.

system call therefore accomplishes the same goal as `rm -rf`, but it does not need to traverse the directory hierarchy or issue individual unlink/rmdir calls for each file and directory in the tree. The performance of this system call is compared to the performance of `rm -rf` on multiple file systems in Table 3. We delete the Linux 3.11.10 source tree using either our recursive-delete system call or by invoking `rm -rf`.

A recursive delete operation is orders of magnitude faster than a brute-force recursive delete on all file systems in this benchmark. This is admittedly an unfair benchmark, in that it foregoes POSIX semantics, but is meant to illustrate the *potential* of range updates in a full-path indexed system. With relative-path indexing, a range of keys cannot be deleted without first resolving the indirection underneath. With full-path indexing, one could directly apply a range delete to the directory, and garbage collect nodes that are rendered unreachable.

There is a regression in regular `rm -rf` performance for BetrFS 0.4, making it slower than Btrfs and BetrFS 0.3. A portion of this is attributable to additional overhead on un-lifting merged nodes (similar to the overheads added to sequential write for splitting); another portion seems to be exercising inefficiencies in flushing a large number of range messages, which is a relatively new feature in the BetrFS code base. We believe this can be mitigated with additional engineering. This experiment also illustrates how POSIX semantics, that require reads before writes, can sacrifice performance in a write-optimized storage system.

More generally, full-path indexing has the potential to improve many recursive directory operations, such as changing permissions or updating reference counts.

7.4 Macrobenchmark performance

Figure 6a shows the throughput of 4 threads on the Dovecot 2.2.13 mailserver. We initialize the mailserver with 10 folders, each contains 2500 messages, and use 4 threads, each performs 1000 operations with 50% reads and 50% updates (marks, moves, or deletes).

Figure 6b measures `rsync` performance. We copy the Linux 3.11.10 source tree from a source directory to a destination directory within the same partition and file system. With the `--in-place` option, `rsync` writes data directly to the destination file rather than creating a temporary file and updating via atomic rename.

Figure 6c reports the time to clone the Linux kernel source code repository [28] from a clone on the local system. The `git diff` workload reports the time to diff between the v4.14 and v4.07 Linux source tags.

Finally, Figure 6d reports the time to `tar` and un-`tar` the Linux 3.11.10 source code.

BetrFS 0.4 is either the fastest or a close second for 5 of the 7 application workloads. No other file system matches that breadth of performance.

BetrFS 0.4 represents a strict improvement over BetrFS 0.3 for these workloads. In particular, we attribute the improvement in the `rsync --in-place`, `git` and un-`tar` workloads to eliminating zone maintenance overheads. These results show that, although zoning represents a balance between full-path indexing and inode-style indirection, full path indexing can improve application workloads by 3-13% over zoning in BetrFS without incurring unreasonable rename costs.

8 Related Work

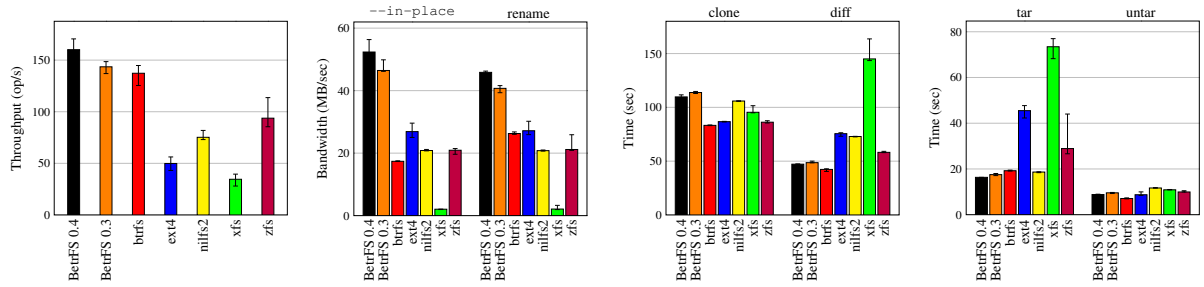
WODs. Write-Optimized Dictionaries, or WODs, including LSM-trees [36] and B^ε-trees [10], are widely used in key-value stores. For example, BigTable [12], Cassandra [26], LevelDB [20] and RocksDB [41] use LSM-trees; TokuDB [49] and Tucana [37] use B^ε-trees.

A number of projects have enhanced WODs, including in-memory component performance [4, 19, 44], write amplification [30, 53] and fragmentation [33]. Like the lifted B^ε-tree, the LSM-trie [53] also has a trie structure; the LSM-trie was applied to reducing write amplification during LSM compaction rather than fast key updates.

Several file systems are built on WODs through FUSE [17]. TableFS [40] puts metadata and small files in LevelDB and keeps larger files on ext4. KVFS [45] uses stitching to enhance sequential write performance on VT-trees, variants of LSM-trees. TokuFS [15], a precursor to BetrFS, uses full-path indexing on B^ε-trees, showing good performance for small writes and directory scans.

Trading writes for reads. IBM VSAM storage system, in the Key Sequenced Data Set (KSDS) configuration, can be thought of as an early key-value store using a B+ tree. One can think of using KSDS as a full-path indexed file system, optimized for queries. Unlike a POSIX file system, KSDS does not allow keys to be renamed, only deleted and reinserted [29].

In the database literature, a number of techniques have been developed that optimize for read-intensive work-



(a) IMAP server throughput. Higher is better.

(b) Rsync throughput. Higher is better.

(c) Git latency. Lower is better.

(d) Tar latency. Lower is better.

Figure 6: Application benchmarks. BetrFS 0.4 is the fastest file system, or essentially tied for fastest, in 4 out of the 7 benchmarks. No other file system offered comparable across-the-board performance. Furthermore, BetrFS 0.4's improvements over BetrFS in the in-place rsync, git clone, and untar benchmarks demonstrate that eliminating zone-maintenance overheads can benefit real application performance.

loads, but make schema changes or data writes more expensive [1–3, 13, 21, 24]. For instance, denormalization stores redundant copies of data in other tables, which can be used to reduce the costs of joins during query, but make updates more expensive. Similarly, materialized views of a database can store incremental results of queries, but keeping these views consistent with updates is more expensive.

Tree surgery. Most trees used in storage systems only modify or rebalance nodes as the result of insertions and deletions. Violent changes, such as tree surgery, are uncommon. Order Indexes [16] introduces relocation updates, which moves nodes in the tree, to support dynamic indexing. Ceph [52] performs dynamic subtree partitioning [51] on the directory tree to adaptively distribute metadata data to different metadata servers.

Hashing full paths. A number of systems store *metadata* in a hash table, keyed by full path, to lookup metadata in one I/O. The Direct Lookup File System (DLFS) maps file metadata to on-disk buckets by hashing full paths [27]. Hashing full paths creates two challenges: files in the same directory may be scattered across disk, harming locality, and DLFS directory renames require deep recursive copies of both data and metadata.

A number of distributed file systems have stored file metadata in a hash table, keyed by full path [18, 38, 48]. In a distributed system, using a hash table for metadata has the advantage of easy load balancing across nodes, as well as fast lookups. We note that the concerns of indexing *metadata* in a distributed file system are quite different from keeping logically contiguous *data* physically contiguous on disk. Some systems, such as the Google File System, also do not support common POSIX operations, such as listing a directory.

Tsai et al. [50] demonstrate that indexing the *in-memory* kernel directory cache by full paths can improve path lookup operations, such as `open`.

9 Conclusion

This paper presents a new on-disk indexing structure, the lifted B^ε-tree, which can leverage full-path indexing without incurring large rename overheads. Our prototype, BetrFS 0.4, is a nearly strict improvement over BetrFS 0.3. The main cases where BetrFS 0.4 does worse than BetrFS 0.3 are where node splitting and merging is on the critical path, and the extra computational costs of lifting harm overall performance. We believe these costs can be reduced in future work.

BetrFS 0.4 demonstrates the power of consolidating optimization effort into a single framework. A critical downside of zoning is that multiple, independent heuristics make independent placement decisions, leading to sub-optimal results and significant overheads. By using the keyspace to communicate information about application behavior, a single codebase can make decisions such as when to move data to recover locality, and when the cost of indirection can be amortized. In future work, we will continue exploring additional optimizations and functionality unlocked by full-path indexing.

Source code for BetrFS is available at `betrfs.org`.

Acknowledgments

We thank the anonymous reviewers and our shepherd Ethan Miller for their insightful comments on earlier drafts of the work. Part of this work was done while Yuan was at Farmingdale State College of SUNY. This research was supported in part by NSF grants CNS-1409238, CNS-1408782, CNS-1408695, CNS-1405641, IIS 1251137, IIS-1247750, CCF 1617618, CCF 1439084, CCF-1314547, and by NIH grant NIH grant CA198952-01. The work was also supported by VMware, by EMC, and by NetApp Faculty Fellowships.

References

[1] AHMAD, Y., KENNEDY, O., KOCH, C., AND NIKOLIC, M. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB 5*, 10 (2012), 968–979.

[2] AHMAD, Y., AND KOCH, C. Dbtoaster: A SQL compiler for high-performance delta processing in main-memory databases. *PVLDB 2*, 2 (2009), 1566–1569.

[3] ARASU, A., BABCOCK, B., BABU, S., DATAR, M., ITO, K., NISHIZAWA, I., ROSENSTEIN, J., AND WIDOM, J. STREAM: the stanford stream data manager. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003* (2003), A. Y. Halevy, Z. G. Ives, and A. Doan, Eds., ACM, p. 665.

[4] BALMAU, O., GUERRAOUI, R., TRIGONAKIS, V., AND ZABLOTCHI, I. Flodb: Unlocking memory in persistent key-value stores. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017* (2017), G. Alonso, R. Bianchini, and M. Vukolic, Eds., ACM, pp. 80–94.

[5] BENDER, M. A., COLE, R., DEMAINE, E. D., AND FARACH-COLTON, M. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings* (2002), R. H. Möhring and R. Raman, Eds., vol. 2461 of *Lecture Notes in Computer Science*, Springer, pp. 139–151.

[6] BENDER, M. A., FARACH-COLTON, M., FINEMAN, J. T., FOGEL, Y. R., KUSZMAUL, B. C., AND NELSON, J. Cache-oblivious streaming b-trees. In *SPAA 2007: Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, California, USA, June 9-11, 2007* (2007), P. B. Gibbons and C. Scheideler, Eds., ACM, pp. 81–92.

[7] BENDER, M. A., FARACH-COLTON, M., JANNEN, W., JOHNSON, R., KUSZMAUL, B. C., PORTER, D. E., YUAN, J., AND ZHAN, Y. An introduction to B^e-trees and write-optimization. *:login; Magazine 40*, 5 (Oct 2015), 22–28.

[8] BONWICK, J. ZFS: the last word in file systems. `https://blogs.oracle.com/video/entry/zfs_the_last_word_in`, Sept. 2004.

[9] BRODAL, G. S., DEMAINE, E. D., FINEMAN, J. T., IACONO, J., LANGERMAN, S., AND MUNRO, J. I. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010* (2010), M. Charikar, Ed., SIAM, pp. 1448–1456.

[10] BRODAL, G. S., AND FAGERBERG, R. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA.* (2003), ACM/SIAM, pp. 546–554.

[11] BUCHSBAUM, A. L., GOLDWASSER, M. H., VENKATASUBRAMANIAN, S., AND WESTBROOK, J. On external memory graph traversal. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, CA, USA.* (2000), D. B. Shmoys, Ed., ACM/SIAM, pp. 859–860.

[12] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (2008), 4:1–4:26.

[13] CIPAR, J., GANGER, G. R., KEETON, K., III, C. B. M., SOULES, C. A. N., AND VEITCH, A. C. Lazybase: trading freshness for performance in a scalable database. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012* (2012), P. Felber, F. Bellosa, and H. Bos, Eds., ACM, pp. 169–182.

[14] CONWAY, A., BAKSHI, A., JIAO, Y., JANNEN, W., ZHAN, Y., YUAN, J., BENDER, M. A., JOHNSON, R., KUSZMAUL, B. C., PORTER, D. E., AND FARACH-COLTON, M. File systems fated for senescence? nonsense, says science! In *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017* (2017), G. Kuenning and C. A. Waldspurger, Eds., USENIX Association, pp. 45–58.

[15] ESMET, J., BENDER, M. A., FARACH-COLTON, M., AND KUSZMAUL, B. C. The tokufs streaming file system. In *4th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage'12, Boston, MA, USA, June 13-14, 2012* (2012), R. Rangaswami, Ed., USENIX Association.

[16] FINIS, J., BRUNEL, R., KEMPER, A., NEUMANN, T., MAY, N., AND FÄRBER, F. Indexing highly dynamic hierarchical data. *PVLDB 8*, 10 (2015), 986–997.

[17] File system in userspace. `http://fuse.sourceforge.net/`, Last Accessed May 16, 2015, 2015.

[18] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003* (2003), M. L. Scott and L. L. Peterson, Eds., ACM, pp. 29–43.

[19] GOLAN-GUETA, G., BORTNIKOV, E., HILLEL, E., AND KEIDAR, I. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015* (2015), L. Réveillère, T. Harris, and M. Herlihy, Eds., ACM, pp. 32:1–32:14.

[20] GOOGLE, INC. LevelDB: A fast and lightweight key/value database library by Google. `http://github.com/leveldb/`, Last Accessed May 16, 2015, 2015.

[21] HONG, M., DEMERS, A. J., GEHRKE, J., KOCH, C., RIEDEWALD, M., AND WHITE, W. M. Massively multi-query join processing in publish/subscribe systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007* (2007), C. Y. Chan, B. C. Ooi, and A. Zhou, Eds., ACM, pp. 761–772.

[22] JANNEN, W., YUAN, J., ZHAN, Y., AKSHINTALA, A., ESMET, J., JIAO, Y., MITTAL, A., PANDEY, P., REDDY, P., WALSH, L., BENDER, M. A., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. Betrfs: A right-optimized write-optimized file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015* (2015), J. Schindler and E. Zadok, Eds., USENIX Association, pp. 301–315.

[23] JANNEN, W., YUAN, J., ZHAN, Y., AKSHINTALA, A., ESMET, J., JIAO, Y., MITTAL, A., PANDEY, P., REDDY, P., WALSH, L., BENDER, M. A., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. Betrfs: Write-optimization in a kernel file system. *TOS 11*, 4 (2015), 18:1–18:29.

[24] JOHNSON, C., KEETON, K., III, C. B. M., SOULES, C. A. N., VEITCH, A. C., BACON, S., BATUNER, O., CONDOTTA, M., COUTINHO, H., DOYLE, P. J., EICHELBERGER, R., KIEHL, H., MAGALHAES, G. R., MCEVOY, J., NAGARAJAN, P., OSBORNE, P., SOUZA, J., SPARKES, A., SPITZER, M., TANDEL, S., THOMAS, L., AND ZANGARO, S. From research to practice: experiences engineering a production metadata database for a scale out file system. In *Proceedings of the 12th USENIX conference on File and Storage Technologies, FAST 2014, Santa Clara, CA, USA, February 17-20, 2014* (2014), B. Schroeder and E. Thereska, Eds., USENIX, pp. 191–198.

[25] KIM, S., LEE, M. Z., DUNN, A. M., HOFMANN, O. S., WANG, X., WITCHEL, E., AND PORTER, D. E. Improving server applications with system transactions. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 15–28.

[26] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *Operating Systems Review 44*, 2 (2010), 35–40.

[27] LENSING, P. H., CORTES, T., AND BRINKMANN, A. Direct lookup and hash-based metadata placement for local file systems. In *6th Annual International Systems and Storage Conference, SYSTOR '13, Haifa, Israel - June 30 - July 02, 2013* (2013), R. I. Kat, M. Baker, and S. Toledo, Eds., ACM, pp. 5:1–5:11.

[28] Linux kernel source tree. `https://github.com/torvalds/linux`.

[29] LOVELACE, M., DOVIDAUSKAS, J., SALLA, A., AND SOKAI, V. VSAM Demystified. `http://www.redbooks.ibm.com/redbooks/SG246105/wwhelp/wwhimpl/js/html/wwhelp.htm`, 2004.

[30] LU, L., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Wisckey: Separating keys from values in ssd-conscious storage. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016.* (2016), pp. 133–148.

[31] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., AND VIVIER, L. The new ext4 filesystem: current status and future plans. In *Ottowa Linux Symposium (OLS)* (Ottowa, ON, Canada, 2007), vol. 2, pp. 21–34.

[32] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for UNIX. *ACM Trans. Comput. Syst. 2*, 3 (1984), 181–197.

[33] MEI, F., CAO, Q., JIANG, H., AND TIAN, L. Lsm-tree managed storage for large-scale key-value store. In *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, September 24-27, 2017* (2017), pp. 142–156.

[34] NILFS: Continuous Snapshotting Filesystem. `https://nilfs.sourceforge.io/en/`.

[35] OLSON, J. Enhance your apps with file system transactions. *MSDN Magazine* (July 2007). http://msdn2.microsoft.com/en-us/magazine/cc163388.aspx.

[36] O'NEIL, P. E., CHENG, E., GAWLICK, D., AND O'NEIL, E. J. The log-structured merge-tree (lsm-tree). *Acta Inf. 33*, 4 (1996), 351–385.

[37] PAPAGIANNIS, A., SALOUSTROS, G., GONZÁLEZ-FÉREZ, P., AND BILAS, A. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.* (2016), A. Gulati and H. Weatherspoon, Eds., USENIX Association, pp. 537–550.

[38] PEERY, C., CUENCA-ACUNA, F. M., MARTIN, R. P., AND NGUYEN, T. D. Wayfinder: Navigating and sharing information in a decentralized world. In *Databases, Information Systems, and Peer-to-Peer Computing - Second International Workshop, DBISP2P 2004, Toronto, Canada, August 29-30, 2004, Revised Selected Papers* (2004), W. S. Ng, B. C. Ooi, A. M. Ouksel, and C. Sartori, Eds., vol. 3367 of *Lecture Notes in Computer Science*, Springer, pp. 200–214.

[39] PORTER, D. E., HOFMANN, O. S., ROSSBACH, C. J., BENN, A., AND WITCHEL, E. Operating systems transactions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009* (2009), J. N. Matthews and T. E. Anderson, Eds., ACM, pp. 161–176.

[40] REN, K., AND GIBSON, G. A. TABLEFS: enhancing metadata efficiency in the local file system. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013* (2013), A. Birrell and E. G. Sirer, Eds., USENIX Association, pp. 145–156.

[41] RocksDB. `rocksdb.org`, 2014. Viewed April 19, 2014.

[42] RODEH, O., BACIK, J., AND MASON, C. BTRFS: the linux b-tree filesystem. *TOS 9*, 3 (2013), 9:1–9:32.

[43] SEARS, R., CALLAGHAN, M., AND BREWER, E. A. *Rose*: compressed, log-structured replication. *PVLDB 1*, 1 (2008), 526–537.

[44] SEARS, R., AND RAMAKRISHNAN, R. blsm: a general purpose log structured merge tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012* (2012), K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, Eds., ACM, pp. 217–228.

[45] SHETTY, P., SPILLANE, R. P., MALPANI, R., ANDREWS, B., SEYSTER, J., AND ZADOK, E. Building workload-independent storage with vt-trees. In *Proceedings of the 11th USENIX conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12-15, 2013* (2013), K. A. Smith and Y. Zhou, Eds., USENIX, pp. 17–30.

[46] SPILLANE, R. P., GAIKWAD, S., CHINNI, M., ZADOK, E., AND WRIGHT, C. P. Enabling transactional file access via lightweight kernel extensions. In *7th USENIX Conference on File and Storage Technologies, February 24-27, 2009, San Francisco, CA, USA. Proceedings* (2009), M. I. Seltzer and R. Wheeler, Eds., USENIX, pp. 29–42.

[47] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS file system. In *Proceedings of the USENIX Annual Technical Conference, San Diego, California, USA, January 22-26, 1996* (1996), USENIX Association, pp. 1–14.

[48] THOMSON, A., AND ABADI, D. J. Calvinfs: Consistent WAN replication and scalable metadata management for distributed file systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015* (2015), pp. 1–14.

[49] TOKUTEK, INC. TokuDB v6.5 for MySQL and MariaDB. `http://www.tokutek.com/products/tokudb-for-mysql/`, 2013. See `https://web.archive.org/web/20121011120047/http://www.tokutek.com/products/tokudb-for-mysql/`.

[50] TSAI, C., ZHAN, Y., REDDY, J., JIAO, Y., ZHANG, T., AND PORTER, D. E. How to get more value from your file system directory cache. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015* (2015), E. L. Miller and S. Hand, Eds., ACM, pp. 441–456.

[51] WEIL, S., POLLACK, K., BRANDT, S. A., AND MILLER, E. L. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)* (Nov. 2004).

[52] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 307–320.

[53] WU, X., XU, Y., SHAO, Z., AND JIANG, S. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. In *Proceedings of the USENIX Annual Technical Conference* (Santa Clara, CA, USA, July 8–10 2015), pp. 71–82.

[54] YUAN, J., ZHAN, Y., JANNEN, W., PANDEY, P., AKSHINTALA, A., CHANDNANI, K., DEO, P., KASHEFF, Z., WALSH, L., BENDER, M. A., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. Optimizing every operation in a write-optimized file system. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016.* (2016), A. D. Brown and F. I. Popovici, Eds., USENIX Association, pp. 1–14.

[55] YUAN, J., ZHAN, Y., JANNEN, W., PANDEY, P., AKSHINTALA, A., CHANDNANI, K., DEO, P., KASHEFF, Z., WALSH, L., BENDER, M. A., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. Writes wrought right, and other adventures in file system optimization. *TOS 13*, 1 (2017), 3:1–3:26.

[56] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in histar. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA* (2006), B. N. Bershad and J. C. Mogul, Eds., USENIX Association, pp. 263–278.