Efficient Namespace Operations and Good Locality in Write-Optimized File Systems

Yang Zhan

Abstract

Full-path-indexed file systems index data and metadata by their full-paths. Therefore, logically related data and metadata are grouped together in the keyspace and full-path-indexed file systems have much better locality than traditional inode-based file systems. On the other hand, in full-path-indexed file systems, a rename must update all related keys, which can be very expensive. However, a full-path-indexed file system implemented on write-optimized data structures can have efficient namespace operations by transferring them into data structure-internal operations. We demonstrate this with two case studies: renaming (preliminary work) and cloning (proposed work).

1 Introduction

Most modern file systems are inode-based. The metadata and data of a file or directory are indexed by the inode number, assigned when the file or directory is created. Inodes add a level of indirection so that file systems can manage each file or directory independently. However, this schema doesn't impose any constraint on the placement of metadata and data under one directory. Thus, in the worst case, the metadata and data can end up scattered over the disk.

On the other hand, BetrFS uses full-path indexing on top of B^ε-trees. Metadata and data are stored as key/value pairs whose keys are the full-paths of the associated files and directories. Therefore, metadata and data under one directory are contiguous in the keyspace. Because of the large node size in B^ε-trees, most logically related metadata and data are stored close to each other on disk.

However, namespace operations are the longstanding problem of full-path-indexed file systems. For example, a rename must update all the keys of key/value pairs being renamed. A naive rename implementation would read all the old key/value pairs, insert them with new keys and delete them. When the size of the related key/value pairs is huge, such rename implementation is very expensive.

In the preliminary work [31], we introduce a new operation to B^ε-trees, `range-rename`, that accomplishes the work of file system renames with bounded IO cost. A `range-rename` slices out a source subtree from the B^ε-tree and moves the subtree to the destination. Unlike other write operations in B^ε-trees, `range-rename` is not write-optimized and the idea can be applied to other tree structures.

The idea of `range-rename` can be extended to other namespace operations, like file clones. A file clone creates a target file whose data are identical to the source file. To clone a file, BetrFS can slice out the subtree as in `range-rename` but share the subtree between the source and the destination.

Many modern file systems start to support file clones. Recently, Linux assigns a specific `ioctl` number for file clones (FICLONE) and detects file clones in the virtual file system (VFS) layer for all file systems. Also, the reflink copy (`cp --reflink`) in Linux tries to invoke file clones directly.

However, existing file clones struggle to deal with locality. When one block of a cloned file is overwritten, the file system has to either allocate an unused disk block, making the locality worse, or copy the whole cloned extent to another place, resulting in large write amplification.

In contrast, the full-path indexing in BetrFS ensures locality and the write-optimized B^ε-trees in BetrFS batch small overwrites to reduce write amplification.

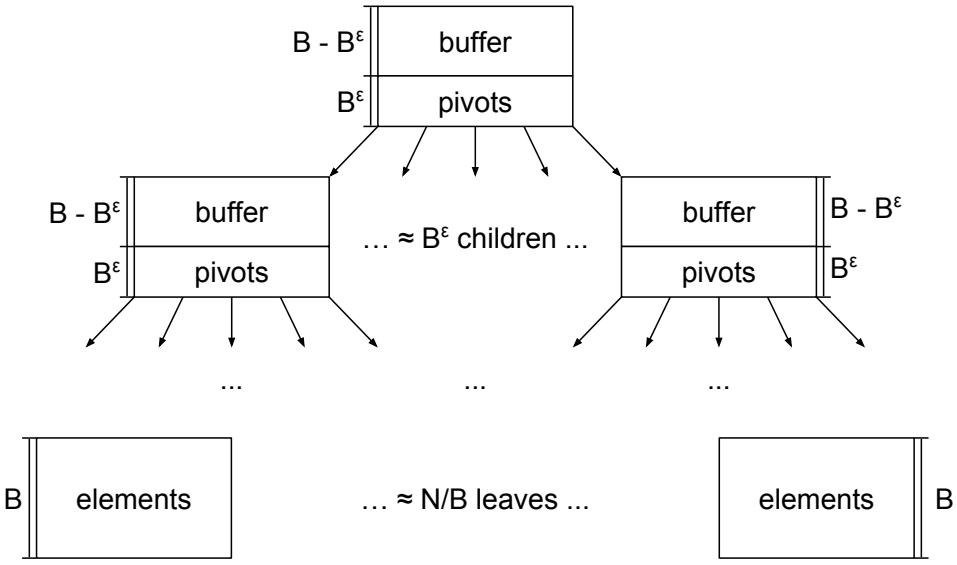


Figure 1: The B$^\varepsilon$-tree.

The thesis of this dissertation is: a full-path-indexed write-optimized file system can have b[illegible]ood locality and efficient namespace operations by implementing namespace operations as data structure-internal operations. We will demonstrate this thesis in the context of the BetrFS.

In the remainder of this proposal, Section 2 talks about the background of BetrFS. Section 3 shows how the rename problem was solved. Section 4 proposes our solution to cloning. Related works are shown in Section 5. Section 6 presents the plan of milestones.

2 Background

This section presents the background of BetrFS. We start by introducing the B$^\varepsilon$-tree, the data structure used by BetrFS. Then, we describe how BetrFS organizes its metadata and data in B$^\varepsilon$-trees. At last, we show the problems of the preliminary BetrFS and how they are addressed in later versions.

2.1 B$^\varepsilon$-tree

The B$^\varepsilon$-tree [1, 3] is a write-optimized variant of the B-tree that cascades writes (Figure 1). Similar to the B-tree, the B$^\varepsilon$-tree stores key/value pai[illegible] its leaves. The B$^\varepsilon$-tree also has pivots and pointers from a parent to its children. Unlike the B-tree where writes take effect on leaves immediately, each interior node in the B$^\varepsilon$-tree has a buffer and writes are gradually flushed down the B$^\varepsilon$-tree. A fresh write is injected as a message into the b[illegible]in the B$^\varepsilon$-tree root. When the buffer of an interior node is full, the B$^\varepsilon$-tree picks one child that can receive the most of its messages and flushes those messages to the child. Thus, each message is written multiple times along the root-to-leaf path. However, because each write is done in a batch with other m[illegible]es, the IO cost of each write is amortized. Compared to the B-tree where writes are put directly to leaves, the IO cost of a write in the B$^\varepsilon$-tree is much smaller.

Table 1 shows the comparison of asymptotic IO costs between the B-tree and the B$^\varepsilon$-tree. A B$^\varepsilon$-tree with node size B partitions each interior node to have one $(B - B^\varepsilon)$-size buffer and B^ε pivots. The resulting tree height is $O(log_{B^\varepsilon} N) = O(log_B N/\varepsilon)$. A point query, which follows a root-to-leaf path, needs to perform $O(log_B N/\varepsilon)$ IOs. Similarly, a write is flushed $O(log_B N/\varepsilon)$ times, while each flush

Data Structure	Insert	Point Query	Range Query
B-tree	$O(log_B N)$	$O(log_B N)$	$O(log_B N + k/B)$
B$^\varepsilon$-tree	$O(log_B N/\varepsilon B^{1-\varepsilon})$	$O(log_B N/\varepsilon)$	$O(log_B N/\varepsilon + k/B)$
B$^\varepsilon$-tree ($\varepsilon = 0.5$)	$O(log_B N/\sqrt{B})$	$O(log_B N)$	$O(log_B N + k/B)$

Table 1: The asymptotic IO costs of B-tree and B$^\varepsilon$-tree.

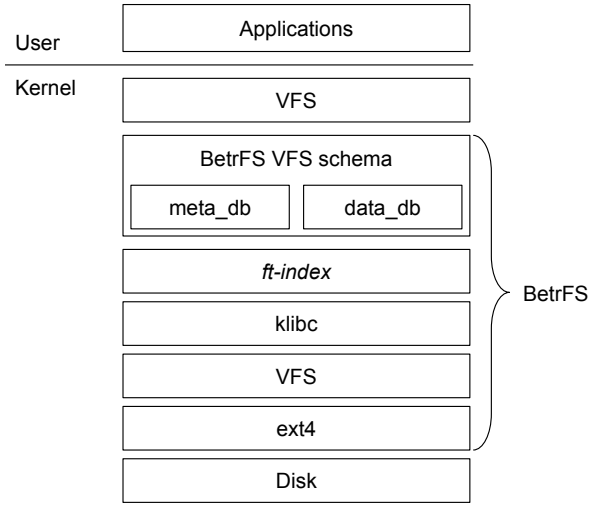


Figure 2: The BetrFS architecture.

is done for at least $O((B - B^\varepsilon)/B^\varepsilon) = O(B^{1-\varepsilon})$ writes. Thus, the amortized cost of each write is $O(log_B N/(\varepsilon B^{1-\varepsilon}))$. If we set $\varepsilon = 0.5$, the B$^\varepsilon$-tree has the same asymptotic IO costs for reads but much better IO costs for writes.

One important change in the B$^\varepsilon$-tree is that the IO costs of reads and writes are asymmetric. Read-before-write is free in B-trees because a write must go through the root-to-leaf path anyway, paying the IO cost of fetching all nodes along the way, and the read-before-write fetches exactly the same set of nodes. However, in the B$^\varepsilon$-tree, most writes can be resolved in the root, so read-before-write can ruin the performance of the B$^\varepsilon$-tree.

To this end, `upserts` are introduced in the B$^\varepsilon$-tree to reduce read-before-write. An `upsert` is a message describing how the old value of the key should be mutated to become the new value. Like other writes, new `upserts` are put to the root. When an `upsert` meets the old value during a node flush, the mutation of this `upsert` is applied to the old value. With `upserts`, reads need to collect `upsert` messages along the root-to-leaf path and apply these `upsert` when the old value is found, but this doesn't incur additional IOs. One example is using key/value pairs as counters, the B$^\varepsilon$-tree can have `upsert` messages describing the amount of increment so that it doesn't have to read the old value before incrementing.

2.2 BetrFS

BetrFS [5, 12, 13, 29, 30, 31] is a Linux in-kernel file system built upon *ft-index* [27], which implements the B$^\varepsilon$-tree and exposes a key-value interface. The architecture of BetrFS is shown in Figure 2. BetrFS interacts with ft-index through point operations, such as `put`, `get` and `del`, as well as range queries with cursors (`c_getf_set_range` and `c_getf_next`). BetrFS also uses the transaction interface of ft-index to execute multiple operations atomically. A redo log and periodic checkpoints (every 5s) in ft-index ensure

that changes can be made persistent on the underlying storage media.

Ft-index cannot be integrated into a Linux kernel module easily because it is a userspace library that calls `libc` functions and `syscalls`. We built a shim layer called `klibc` which implements all functions ft-index requires. By implementing those functions in `klibc` instead of directly modifying ft-index, we are able to migrate ft-index into the kernel module intact.

BetrFS uses two key/value indexes to store the metadata and data in a file system. One `meta_db` maps full-paths to `struct stat` structures. Another `data_db` maps (full-path + block number) to 4KB blocks. When the VFS needs the metadata, BetrFS queries the `meta_db` with the full-path and constructs the corresponding inode from the `struct stat`. Likewise, when a dirty inode needs to be written, the `struct stat` is assembled from the inode and written to the `meta_db` with the full-path key. Blocks of a file are fetched and written by the full-path and the indexes of blocks. Although other block granularity is possible, 4KB is the natural block size because it is the same as the page size in the Linux page cache.

Full-path indexing is desirable because it offers locality. With full-path indexing, all keys under one directory are contiguous in the keyspace. This, combined with the large node size (4MB) used in B^ε-trees, means a large amount of data are stored close to each other on disk. After BetrFS fetches one 4KB block of some file from the disk, all nodes along the root-to-leaf path are present in memory. Thus, a subsequent fetch to some other block in the same file or another file under the same directory is likely to be resolved in memory, which significantly increases performance and IO efficiency.

In BetrFS, in order to avoid read-before-write, writes can proceed without fetching the old block to memory. Conventional file systems must read the old block from the disk to the page cache before writing to that block. However, as described in Section 2.1, B^ε-trees have asymmetric read and write costs, so read-before-write should be avoided. In BetrFS, if the corresponding block is not in memory, an `upsert` message describing the offset and the length of this write. When this message meets the old value during a flush, the change is applied.

2.3 Making BetrFS Better

B^ε-trees and full-path indexing give the first version of BetrFS (BetrFS 0.1) good random write performance and good locality. For example, BetrFS 0.1 is 12.53x faster in creating 3 million small files and 6.86x faster in doing `find` in the Linux source directory than other file systems.

However, there are problems with full-path indexing. Deleting the Linux source directory takes 2.33x longer and renaming the Linux source directory takes 1.07x longer than other file systems. We tried to solve these problems in the second version (BetrFS 0.2).

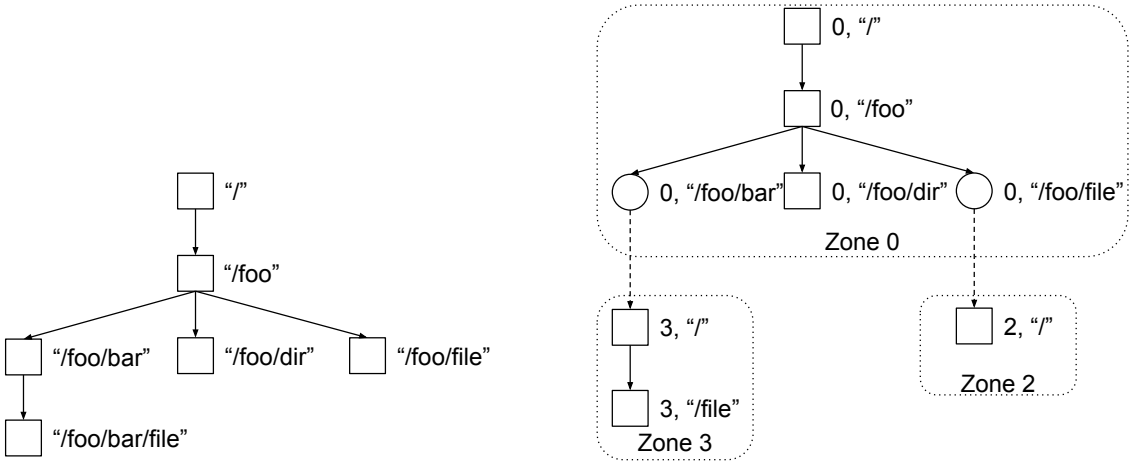
2.3.1 Delete

In BetrFS 0.1, each 4KB block is stored as one key/value pair in the B^ε-tree. Deleting a file requires one `del` call for each block. This becomes costly when the file is large.

BetrFS 0.2 introduces a `range-delete` message that deletes a whole range of key/value pairs. This is possible because the B^ε-tree, unlike the B-tree, is a message-based data structure. A `range-delete`, like all other write operations, injects a `range-delete` message of its range into the root of the B^ε-tree. While the `range-delete` message is flushed down the B^ε-tree, it identifies all old messages in its range and removes them from the B^ε-tree. Thus, when the `range-delete` message reach the leaf, all previous messages in its range are removed from the B^ε-tree.

2.3.2 Rename

Rename is a longstanding problem in full-path indexing. With full-path indexing, all data and metadata are associated with full-path keys. Therefore, a rename needs to update all keys in the source. Also, because all key/value pairs are sorted by their keys, they must be moved to the destination.



(a) Full-path indexing. (b) Zoning.

Figure 3: Full-path indexing and zoning.

BetrFS 0.1 uses a simple approach that reads all related key/value pairs, puts them back with new keys and deletes the old key/value pairs. The cost of this rename increases as the size of the source increases.

BetrFS 0.2 tries to bound the rename cost at the schema level. Instead of full-path indexing, BetrFS 0.2 uses zoning (relative-path indexing). Zoning divides the directory hierarchy into zones. Full-path indexing is kept inside a zone while indirection is used between zones.

For example, in Figure 3b, directory "/foo/bar" forms its own zone (zone-id 3). The metadata stored with key (zone-id 0, "/foo/bar") (root zone-id is always 0) directs BetrFS to the fetch the metadata of key (zone-id 3, "/"). And the file "/foo/bar/file" is stored with key (zone-id 3, "/file").

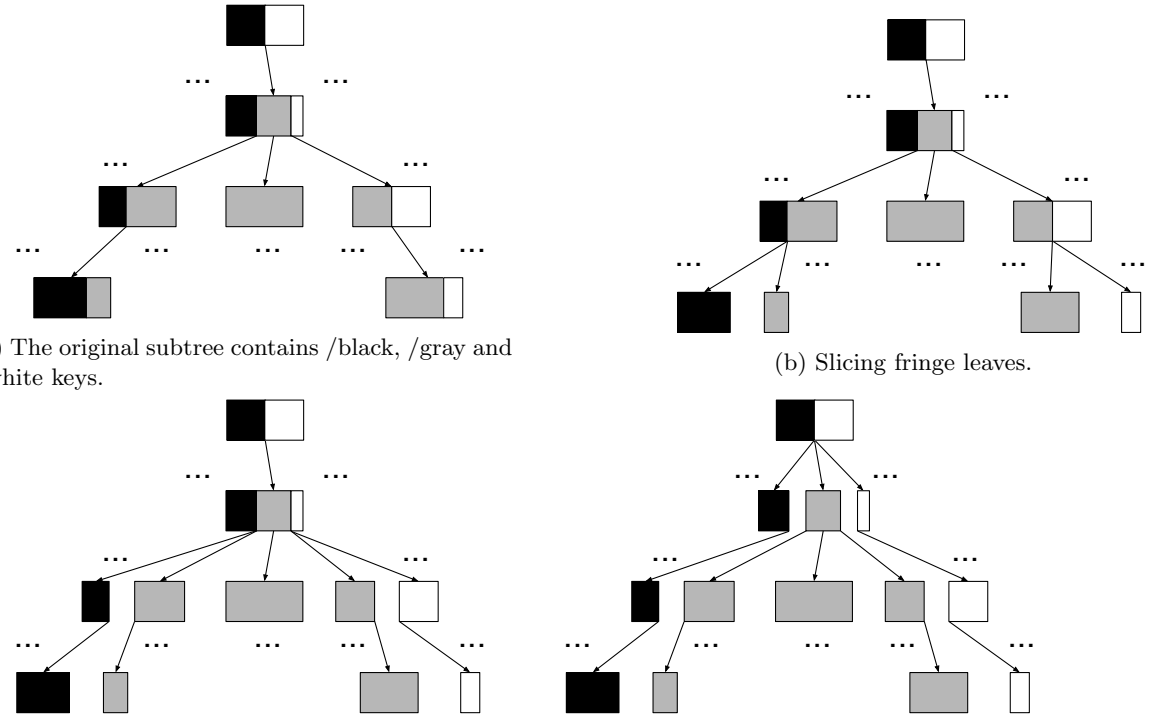
Zoning tries to balance locality and rename performance through the target zone size. Locality is still maintained within a zone with full-path indexing, so we want to have bigger zones to pack more things together. On the other hand, renaming something that is not a zone root still involves moving all related key/value pairs, so smaller zones mean lower upper-bound on rename cost.

BetrFS 0.2 picks a target zone size (128KB by default) and tries to keep each zone close to that size. If the size of metadata or data under something that is not a zone root grows too big, BetrFS 0.2 does a zone split which moves all related key/value pairs to a newly formed zone. Likewise, if the size of a zone becomes too small, BetrFS 0.2 merges the zone.

3 The full path to full-path indexing

Although zoning fixes rename performance in BetrFS, it has drawbacks. The locality is not as good as full-path indexing because full-path indexing is only maintained inside a zone. More importantly, zoning imposes zone maintenance cost. If an operation incurs a zone split or merge, the cost of the zone split or merge is charged to that operation. Thus, BetrFS may pay a significant amount of tax to zone maintenance on workloads that have nothing to do with renames. For example, in Tokubench that creates 10 million small files, the cumulative throughput has a sudden 56.6% drop when 2 million files are created because of zoning splits (BetrFS 0.3 in Figure 6).

Our paper in FAST'18 [31] presents BetrFS 0.4 (BetrFS 0.3 contains bug fixes) that introduces a new solution for renames. The new solution, `range-rename`, doesn't tax other operations and keeps full-path indexing. Unlike zoning, `range-rename` works directly on the B$^\varepsilon$-tree. BetrFS in the VFS layer simply



(a) The original subtree contains /black, /gray and /white keys.

(b) Slicing fringe leaves.

(c) Slicing fringe nodes which were parents of fringe leaves.

(d) Slicing the LCA to get an isolated subtree of /gray keys.

Figure 4: Slicing /gray between /black and /white.

calls `range-rename` in ft-index to complete renames.

3.1 Range-rename

`Range-rename` is based on the fact that the full-path indexing in BetrFS groups all keys involved in a rename in a contiguous keyspace. Thus, if there exists an isolated subtree that contains and only contains all those related keys, we can move all the keys by moving the subtree around with a pointer swing.

There are two obstacles in this approach:

- completely isolated subtrees are rare;
- keys in the subtree are not updated.

`Range-rename` addresses these two issues with **tree surgery** and **key lifting**, respectively.

3.1.1 Tree surgery

Tree surgery slices out one isolated subtree in the source and another isolated subtree in the destination, then uses a pointer swing to move the source subtree to the destination. The destination subtree needs to be sliced out because of two reasons: 1. POSIX allows file renames to overwrite files, so there can be data of the old file in the destination; 2. Because B^{ε}-tree injects new writes into the root and flushes them down the tree gradually, some delete messages may not reach their target key/value pairs, which

leaves deleted key/value pairs in the B$^\varepsilon$-tree. Slicing in the destination also helps to setup pivots for the pointer swing.

Slicing can be viewed as getting an isolated subtree of range (min, max). Figure 4 shows how an isolated subtree of /gray keys are sliced out. Most nodes in the B$^\varepsilon$-tree either fall completely out of the range or completely in the range. Slicing only needs to deal with other nodes that are partly in the range and partly out of the range, which are called **fringe nodes**.

Fringe nodes can be identified by walking down the B$^\varepsilon$-tree with the *min* and *max* keys. This process results in two root-to-leaf path that contains all fringe nodes. One important fringe node is the **Lowest Common Ancestor (LCA)**, which is the lowest node whose range covers both the *min* and *max* keys. If the *min* and *max* keys lead to the same leaf node, that leaf is the LCA. If the *min* and *max* keys diverge after the root, the root is the LCA.

After all fringe nodes are identified, slicing starts separating related keys and unrelated keys from the bottom up until the LCA. Slicing uses the already available tree mechanism, node splits, to accomplish this job. Unlike node splits in tree rebalancing that usually split the node evenly, slicing splits the node with the *min* or *max* key. A fringe node splits into two nodes, one fully in the range and the other fully out of the range. After the LCA is split, we get an isolated subtree and can move that subtree around with a pointer swing.

Figure 4 shows how an isolated subtree is sliced out. In this example, the slicing wants to get an isolated subtree that contains and only contains /gray keys. The original B$^\varepsilon$-tree is shown in Figure 4a, all nodes that are not completely gray are fringe nodes and the child of the root is the LCA. The node split process starts in Figure 4b where the leaves are split. At last, in Figure 4d, the LCA is split and an isolated subtree is formed.

One caveat here is that because we want to move the subtree around, all pending messages above the LCA must be flushed at least to the LCA. Therefore, slicing flushes pending messages down the tree while walking down with the *min* and *max* keys. In fact, during the walking, we flushed all messages to the leaves to make node splits easier.

Another issue is that the B$^\varepsilon$-tree assumes all leaves to be at the same depth. This means the source subtree and the destination subtree must be with the same height. To this end, when one slicing reaches its LCA, it needs to keep slicing until the other slicing also reaches the LCA.

After the two subtrees are sliced out, tree surgery swaps the two subtrees with pointer swings. An ideal way is to garbage collect the destination subtree immediately because that tree will not be accessed anymore. But there is no such out-of-the-tree garbage collection mechanism in ft-index, so we swap the tree and let ft-index handle garbage collection using its own cleaning mechanism.

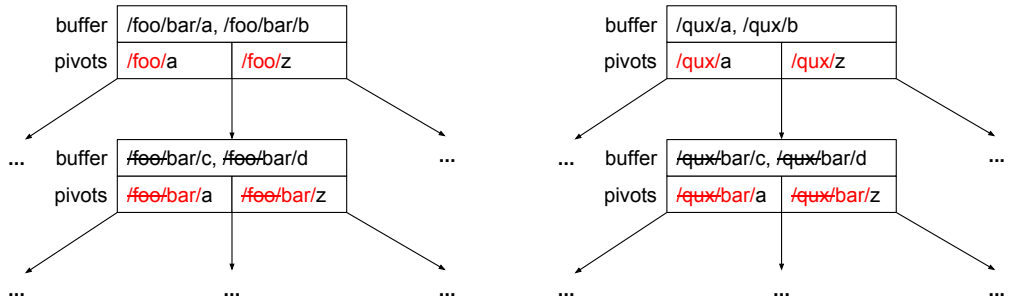
The IO cost of tree surgery is $O(tree\ height)$ because 4 root-to-leaf walks are performed, two in the source and two in the destination. Compared to the old way that touches all related keys in $O(subtree\ size)$ IOs, the cost of tree surgery is bounded by the height of the B$^\varepsilon$-tree and doesn't grow infinitely when the size of the related data grows.

3.1.2 Key lifting

After tree surgery, all keys in the subtree moved to the destination are still full-paths in the source. These keys must be updated to the new full-paths in the destination to make the B$^\varepsilon$-tree correct. A naive approach would traverse the subtree and update all keys. This is costly and reverts the `range-rename` back to the unbounded $O(subtree\ size)$.

Key lifting is introduced to deal with the key updating issue. Key lifting is based on the observation that with certain key comparison functions, the two pivots bounding a subtree in the parent restrict the prefixes of keys in the subtree. Specifically, BetrFS has `memcmp` as the key comparison function, so all keys in the subtree must share the common prefix of the pivots. This means that the common prefix is redundant information in the subtree and only the suffixes of keys need to be stored in the subtree.

Key lifting converts B$^\varepsilon$-trees to lifted B$^\varepsilon$-trees. All operations walking down a root-to-leaf path must



(a) Keys in the subtree have /foo prefixes.

(b) Keys int the subtree have /qux prefixes.

Figure 5: With key lifting, after moving the subtree to a new location, all prefix are updated automatically.

File system	find (sec)	grep (sec)
betrfs 0.4	0.227 ± 0.0	3.655 ± 0.1
betrfs 0.3	0.240 ± 0.0	5.436 ± 0.0
betrfs 0.4-no-rr	0.230 ± 0.0	3.600 ± 0.1
Btrfs	1.311 ± 0.1	8.068 ± 1.6
ext4	2.333 ± 0.1	42.526 ± 5.2
nilfs2	6.841 ± 0.1	8.399 ± 0.2
XFS	6.542 ± 0.4	58.040 ± 12.2
ZFS	9.797 ± 0.9	346.904 ± 101.5

Table 2: Time to perform recusive directory traversals of the Linux 3.11.10 source tree.

collect lifted prefixes along the way. It needs to concatenate lifted prefixes and the suffix in the node to recover the whole key.

An example can be seen in Figure 5a, the common prefixes of pivots are marked red in this figure. In the parent, the common prefix of the two pivots is "/foo/", so we know all keys follow that pointer must have that common prefix. We use strikethrough to mark lifted prefixes. These prefixes are not physically stored in the node, but virtually the keys are perceived with these prefixes.

With key lifting, key updates complete automatically with tree surgery. This is because now the prefix of keys in a subtree depends on the path from the root to the subtree. When the subtree is moved to a new location with tree surgery, the prefixes of the keys in that subtree are changed according to the new location.

Figure 5b shows the result of a subtree move. The subtree used to be bounded by "/foo/a" and "/foo/b" in Figure 5a, so all keys were viewed with "/foo/" prefixes. In the new location, the subtree is bounded by "/qux/a" and "/qux/b", which means all keys in the subtree now have "/qux" prefixes.

Key lifting incurs no additional IO cost. A root-to-leaf traversal just needs to collect lifted prefixes along the way, and this doesn't add any IO cost. To maintain key lifting, node splits or merges may need to modify keys because pivots are added or removed. But since node splits or merges already have to read all related nodes from disk, no additional IO needs to be done.

Therefore, with lifting, key updates are done without IO cost during the tree surgery process. The total cost of `range-rename` is still $O(\textit{tree height})$.

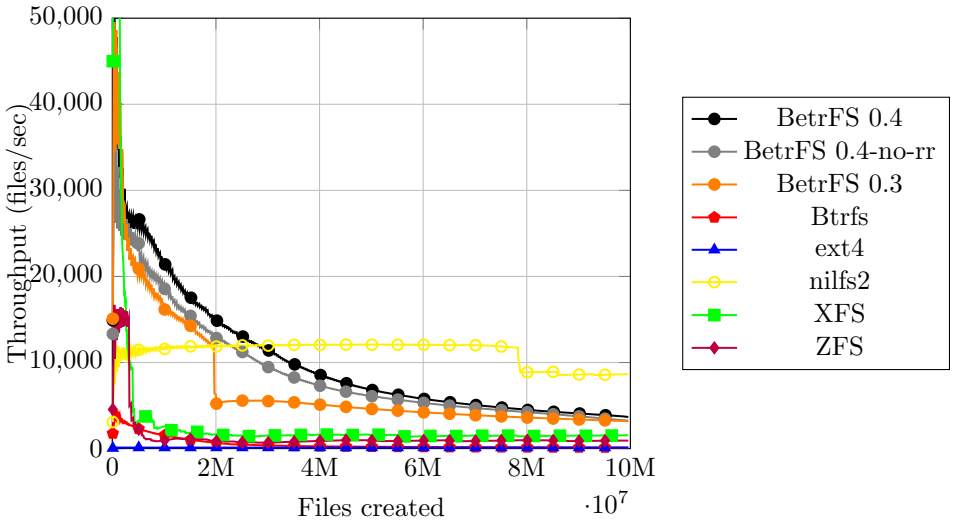


Figure 6: Cumulative file creation throughput during the Tokubench benchmark.

3.2 Range-rename evaluation

We compare BetrFS 0.4 with several file systems, including BetrFS 0.3, BetrFS 0.4 without `range-rename` (as BetrFS 0.4-no-rr to represent BetrFS 0.1), Btrfs [22], ext4 [15], nilfs2 [14], XFS [26], and ZFS [2]. All benchmarks are run on a Dell Optiplex 790 with a 4-core 3.40 GHz i7 CPU, 4GB RAM, and a 500 GB hard disk.

Figure 6 shows the result of TokuBench. This benchmark creates 10 million files in a balanced tree directory where each directory contains at most 128 files or subdirectories and measures the cumulative throughput. The performance of BetrFS 0.3 plummets when about 2 million files are created due to zone maintenance cost. The instantaneous throughput remains between 200 to 600 files per second for 2 minutes, though it is more than 7000 files per second before and after that. On the contrary, BetrFS 0.4 is even faster than BetrFS 0.1 (measured as BetrFS 0.4-no-rr), probably because lifting eliminates long common prefixes. In all benchmarks we run, BetrFS 0.4 doesn't show significant regression as BetrFS 0.3 with zoning does.

In Table 2, we measure locality by performance directory scans (`find` and `grep`) in a linux-3.11.10 source directory. BetrFS 0.4 finishes much faster than other file systems except BetrFS 0.3 (5.77x faster in a find, and 2.21x faster in a grep). Though BetrFS 0.3 is still faster than other file systems with zoning, the locality is not as good as full-path indexing file systems.

Figure 7 presents the result of our rename benchmark. In this benchmark, we create a file and then rename it 100 times. We measure the throughput of renames with different file sizes. The performance of BetrFS 0.1 keeps dropping as the file size grows larger. However, BetrFS 0.4 starts to use `range-rename` when the file size is larger than the node size in the B^ε-tree (4MB) and the performance is competitive with other file systems. But, because inode-based file systems finish a rename with a pointer swing in $O(1)$ and BetrFS 0.4 has a higher upper bound, the rename performance of BetrFS 0.4 is not always as good as other file systems.

4 Proposed work: efficient clones

We can build efficient clones on top of `range-rename`. Essentially, a rename is a clone with a delete. `Range-rename` swaps the sliced-out source and destination subtrees and uses `range-delete` to garbage

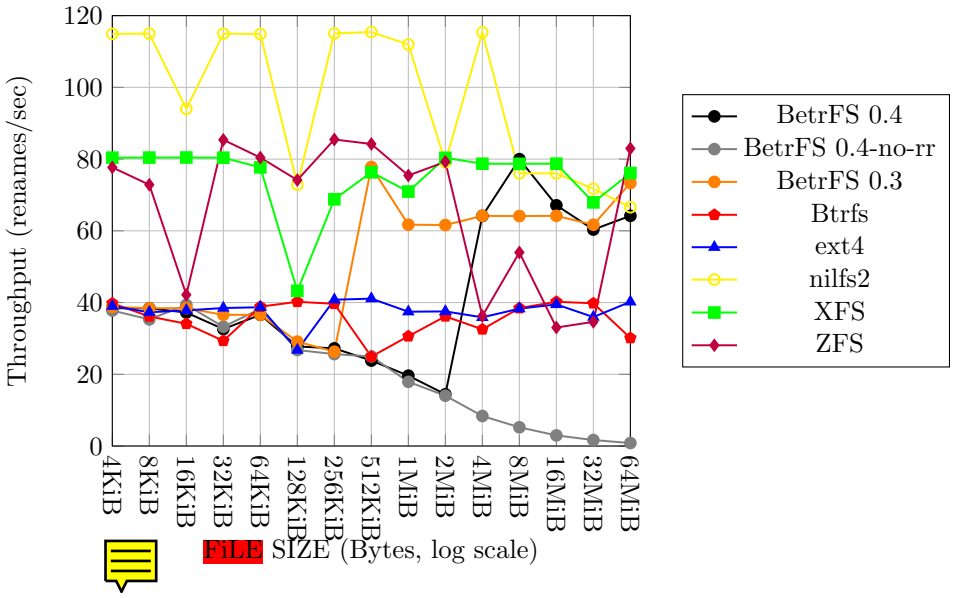


Figure 7: Rename throughput as a function of file size.

collect the destination subtree. Instead, clones can make the parent-to-child pointers in the source and destination point to the same subtree and garbage collect the other destination subtree separately. With that, the source and destination share the content of the subtree.

We propose to explore the balance of sharing versus copying a cloned subtree in a write-optimized dictionary. Sharing a subtree saves space, however, when there is enough difference, it might worth copying the subtree for better locality. B$^\varepsilon$-trees only flush messages from parent to child when there are enough messages in this flush. At this point, we can break the sharing of the root of the subtree and keep the sharing of the children. In this way, B$^\varepsilon$-trees can gradually unshare the whole subtree.

The challenge is constructing multiple views of a single subtree. Queries to the subtree following different root-to-subtree paths should fetch the same value but with different keys. Also, different root-to-subtree paths have different pending writes.

We also want to make `range-rename` more efficient. Currently, `range-rename` locks the whole subtree while slicing from bottom up, forbidding concurrent reads to the subtree. Instead, `range-rename` can use a top-down slicing, which unlocks parts of the subtree during slicing. Also, `range-rename` completes with no delayed work, which doesn't fully utilize the write-optimization of B$^\varepsilon$-trees. Instead, we can have a message-based `range-rename` that can be flushed down the B$^\varepsilon$-tree gradually, like other messages in B$^\varepsilon$-trees.

4.1 Smart pivots

In `range-rename`, both the source and the destination must be sliced out before the pointer swing. This approach is generic and applicable to other tree structures like B-trees. However, it doesn't fit into the write-optimization of the B$^\varepsilon$-tree where operations are inserted as messages and gradually flushed down in batches.

4.1.1 `GOTO` messages

We can introduce a new type of message, `GOTO` messages. A `GOTO` message has a range (sp_l, sp_r) and points to the root of a subtree. With `GOTO` messages, after the source subtree is sliced out, a `GOTO`

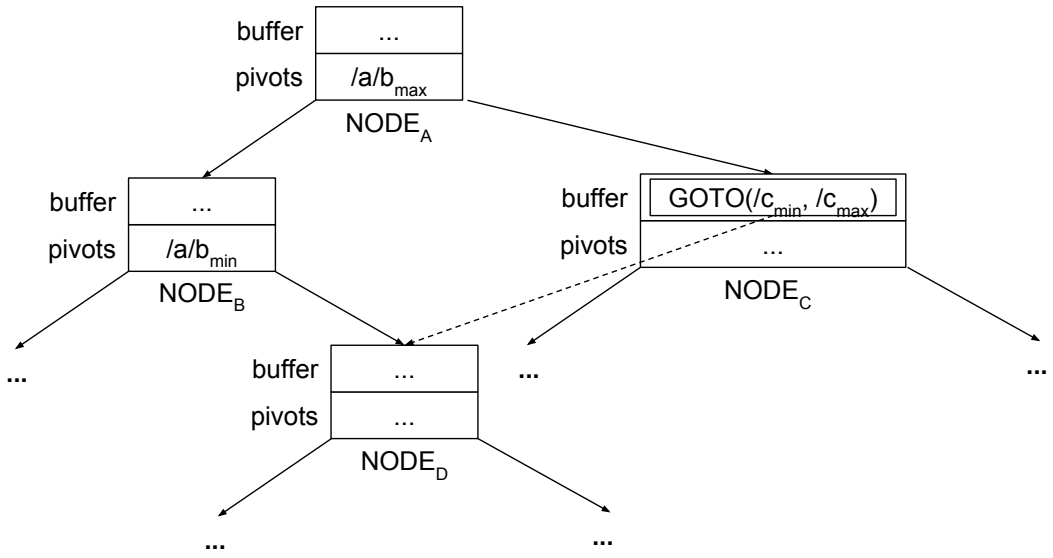


Figure 8: The `GOTO` message directs queries of /c to another subtree.

message, which specifies the destination range and the root of the subtree, is injected to the root.

`GOTO` messages would redirect queries. Ordinarily, when a query for *key* reaches one non-leaf node, it would find the child whose range covers *key* and continue to search that child. However, if the query sees a `GOTO` message in the node whose range covers *key*, it would be redirected to the root of the subtree the `GOTO` message points to. Essentially, a `GOTO` message acts as additional pivots in the node.

For example, "clone /a/b to /c" results in a `GOTO` Message that covers range $(/c_{min}, /c_{max})$ and points to the sliced out subtree whose range is $(/a/b_{min}, /a/b_{max})$. Figure 8 shows the B^{ε}-tree when the `GOTO` message is flushed to NODE_C. Because keys in the `GOTO` message are c_{min} and c_{max}, the message is flushed to the right child of NODE_A. A query for /c/d searches NODE_C after NODE_A, but when the query sees the `GOTO` message, it would then search NODE_D instead of any child of NODE_C.

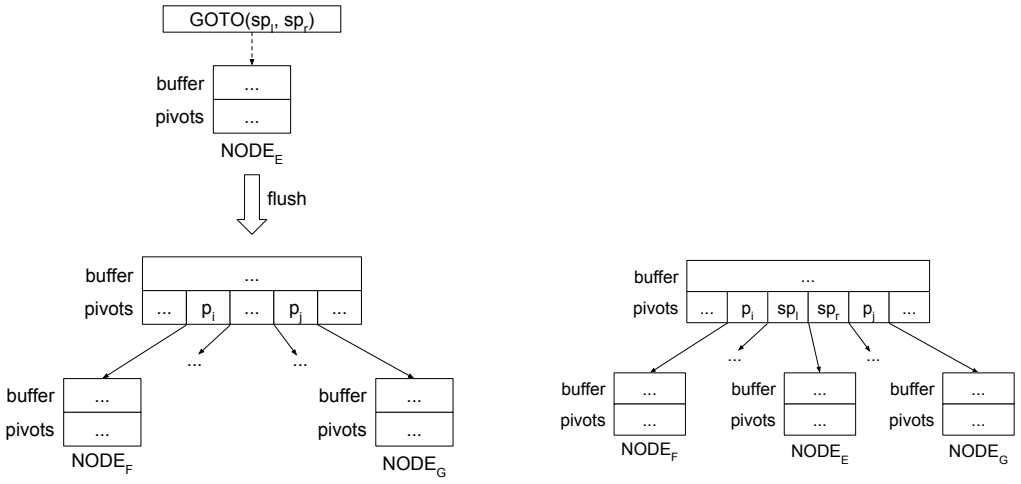
Like other messages, a `GOTO` message is injected to the root of the B^{ε}-tree and flushed down with other messages. While a `GOTO` message is flushed down the B^{ε}-tree, it can remove all old messages in its range, like a `range-delete` message. Additionally, we can treat `range-delete` messages as `GOTO` messages that point to nothing, thus generalizing range messages in the B^{ε}-tree.

To maintain the asymptotic IO cost for queries, all `GOTO` messages in a node of height h must point to subtrees whose heights are less than h. Otherwise, a query might have to search more than $O(\textit{tree height})$ nodes. Therefore, a `GOTO` message of height h (the `GOTO` message points to a subtree of height h) must transform into something else before it is flushed to a node of height h.

When a `GOTO` message of height h is flushed into a node of height $h+1$, instead of injecting the message to the node buffer, we add two real pivots and a child pointer to the node. Figure 9 illustrates the process. In Figure 9a, the node contains pivots $p_0, ..., p_n$, the `GOTO` message has range (sp_l, sp_r) and $p_0 < ... < p_i < sp_l < sp_r < p_j < p_n$. In Figure 9b, pivots $p_{i+1}, ..., p_{j-1}$ are removed from and sp_l and sp_r are added to the node. The child pointer between sp_l and sp_r points to the subtree the `GOTO` message points to. After that, the `GOTO` message can be discarded.

4.1.2 `Smart-pivot` messages

Although `GOTO` messages can be flushed down the B^{ε}-tree, they still require slicing out the source and destination subtrees. The source subtree needs to be sliced out because the `GOTO` message has to point



(a) Before flushing.

(b) After flushing.

Figure 9: The `GOTO` message of height h transforms to real pivots when it is flushed to the node of height $h+1$ ($p_0 < ... < p_i < sp_l < sp_r < p_j < p_n$).

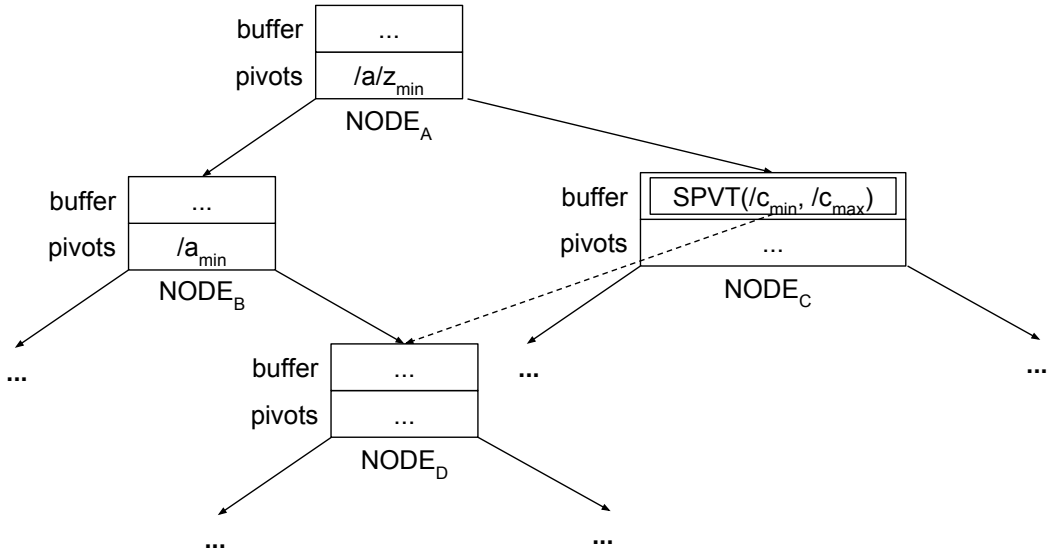


Figure 10: The `smart-pivot` message (SPVT) after "clone /a/b to /c". Note the subtree covers range ($/a_{min}$, $/a/z_{min}$), not ($/a/b_{min}$, $/a/b_{max}$).

to a subtree that is perfectly in range. And when the B$^\varepsilon$-tree transform the `GOTO` message, it needs to add pivots to the node, which requires adjusting other children in the node to have the right range.

All these problems are from the constraint that a subtree must cover the range specified by its ancestors or the `GOTO` message. If this constraint can be removed, cloning can be further optimized.

Therefore, we introduce another type of messages, `smart-pivot`. Like `GOTO` messages, a `smart-pivot` message also has range (sp_l, sp_r) but it can point to a subtree whose range is larger than (sp_l, sp_r). This might break lifting because a large range can lift less. Thus, the `smart-pivot` message must also store what is lifted in the subtree.

Figure 10 shows the `smart-pivot` message of "clone /a/b to /c". Note, the subtree is of range $(/a_{min}, /a/b_{min})$, whereas in Figure 8, the subtree must be of range $(/a/b_{min}, /a/b_{max})$. The subtree may contain keys that are not in the range of the `smart-pivot` mesasge, for example, /a/a. But queries can remember the range of `smart-pivot` messages and ignore those keys.

Now, a `smart-pivot` message can be injected immediately after the source LCA is found. There is no need to slice the source subtree. Also, `smart-pivot` messages can be added when the B$^\varepsilon$-tree transforms the `smart-pivot` message in the destination.

4.2 Garbage collection

Since the introduction of `range-delete` messages, garbage collection has long been a problem in BetrFS. If a subtree is completely in the range of a `range-delete` message, unless BetrFS creates some new entries with the same path, the `range-delete` message might never be flushed to the root of the subtree. And the subtree will never be cleaned.

A background garbage collection thread can solve the problem. If a `range-delete` message covers a whole subtree, this thread should recycle all nodes in the subtree. The thread only needs to read the header of each node, which contains the block numbers of its children and is much smaller than the whole node.

`Smart-pivot` makes garbage collection more complicated. A `smart-pivot` message can hold a whole subtree before it becomes real pivots while the `smart-pivot` message only needs a small fraction of the subtree. The garbage collection thread needs a way to figure out what parts of the subtree are in use and what parts are not.

To this end, we can introduce `range-refc` messages that, instead of incrementing the reference count of the whole node, increment the reference count of a certain key range. When the garbage collection thread tries to clean up the whole subtree, it can figure out which parts are still in use by `smart-pivot` messages through `range-refc` messages.

Alternatively, the `range-refc` information can be stored in the block table so we don't need to write two copies of the node to remove the `range-refc` message from the node when the `smart-pivot` message unshares the node.

4.3 Preferential splitting

A lot of work in both `range-rename` and `smart-pivot` attributes to handling fringe nodes, nodes that contains both related and unrelated keys. Therefore, it would be beneficial to reduce the number of fringe nodes.

The goal of preferential splitting is to reduce fringe nodes as much as possible by carefully picking pivots in node splits. Suppose range (sp_l, sp_r) is cloned. If one pivot in a node happens to be sp_l or sp_r, this pivot already does the work of separating keys and unrelated keys in the descendants of the node. And there is no need to cut the B$^\varepsilon$-tree with sp_l or sp_r from this node on.

With preferential splitting, a leaf split lets BetrFS decide what the new pivot is. This split should not create unbalanced leaves, i.e., all resulting leaves should be at least 1/4 of the full size.

A naive way would be comparing all keys in the range of [1/4, 3/4] of the leaf and picking the pair of two adjacent keys that share the shortest common prefix. But this scan can be costly. For example, in BetrFS, a full leaf is 4MB and each key/value pair in `meta_db` is less than 200 Bytes, which means more than 10000 key comparisons in preferential splitting..

We can do preferential splitting that only requires the reading two keys. Because the shortest common prefix of adjacent keys is the same as the common prefix of the smallest (at 1/4 of the leaf) and the largest (at 3/4 of the leaf) candidate keys, we can construct a good pivot from these two keys.

Nonleaf splits can be viewed as promoting a pivot from the child to the parent. Because the limit on the number of pivots in ft-index is 16, we can afford one-to-one comparisons.

4.4 Proposed evaluation

We should focus on two aspects of `smart-pivot` messages. First, does `smart-pivot` rename and clone fast enough? Second, do BetrFS clones have good locality? We can measure rename performance with the rename benchmark in our `range-rename` work. And we can modify this benchmark to measure clone performance. We can measure locality by performing a file clone with subsequent writes to two copies and then measuring the time to read two copies. Besides, we can use Docker [6] to benchmark clones in real applications. Docker is a container application that clones images for the starting contents. Therefore, we can evaluate the common cases of file clones.

We should compare BetrFS with `smart-pivot` to other cloning file systems. Both Btrfs and XFS can do file clones. ZFS and nilfs2 can create snapshots (immutable clones of the whole file system).

5 Related work

5.1 Write-optimized file systems

Most write-optimized file systems are built on FUSE [10] and LSM-trees [18].

KVFS [24] is built on VT-trees, LSM-trees with stitching. When VT-trees flush and merge two SSTables (similar to nodes in B^ε-trees), it identities overlapping key ranges and only merges these ranges. This technique, which is called stitching, avoids writing the same block multiple times but causes fragmentation. Therefore, KVFS needs to defragment when the disk is nearly full.

TableFS [20] keeps metadata and small files (less than 4KB) on the LSM-tree and large files in the underlying ext4 file system. Therefore, TableFS outperforms other file systems on metadata and small file benchmarks.

TokuFS [9] is a full-path indexing file system with B^ε-trees on FUSE. It is the precursor to BetrFS, showing good performance for small writes and directory scans.

5.2 File systems with snapshots

Many modern file systems provide snapshot mechanism, making read-only copies of the whole file system.

The WAFL file system [11] organizes all blocks in a tree structure. By copying the vol_info block, which is the root of the tree structure, WAFL creates a snapshot. Later, WAFL introduces a level of indirection between the file system and the underlying disks [8]. Therefore, multiple active instances of the file system can exist at the same time and WAFL can create writable snapshots of the whole file system by creating a new instance and copying the vol_info block.

FFS [16] creates snapshots by suspending all operations and creating a snapshot file whose inode is stored in the superblock. The size of the snapshot file is the same as the disk. Upon creation, most block pointers in the snapshot inode are marked “not copied” or “not used” while some metadata blocks are copied to new addresses. Reading a “not copied” address in the snapshot file results in reading the

address on the disk. When a "not copied" block is modified in the file system, FFS copies the block to a new address and updates the block pointer in the snapshot inode.

NILFS [14] is a log-structured file system that organizes all blocks in a B-tree. In NILFS, each logical segment contains modified blocks and a checkpoint block used as the root of the B-tree. NILFS gets the current view of the file system from the checkpoint block of the last logical segment. NILFS can create a snapshot by making a checkpoint block permanent.

ZFS [2] also stores the file system in a tree structure and creates snapshots by copying the root of the tree (uberblock). To avoid maintaining one block bitmap for each snapshot, ZFS keeps birth time in each pointer. A block should not be freed if its birth time is earlier than any snapshot. In that case, the block is added to the dead list of the most recent snapshot. When a snapshot is deleted, all blocks in its dead list are checked again before being freed.

GCTree [7] implements snapshots on top of ext4. It chains different versions of a metadata block with GCTree metadata. Also, each pointer in the metadata block has a "borrowed bit" indicating whether the target block is inherited from the previous version. Therefore, GCTree can check whether to free a block by inspecting GCTree metadata and doesn't need to maintain reference counts.

NOVA-Fortis [28] targets at non-volatile main memory (NVMM). In NOVA-Fortis, each inode has a private log with log entries pointing to data blocks. To enable snapshots, NOVA-Fortis keeps a global snapshot ID and adds the creating snapshot ID to log entries in inodes. NOVA-Fortis decides whether to free a block based on the snapshot ID in the log entry and active snapshots. It also deals with DAX-style `mmap` by stalling page faults when marking all pages read-only.

5.3 File systems with file or directory clones

Some file systems go futher to support cloning one single directory or file.

The Episode file system [4] groups everything under a directory into a fileset. Episode can create an immutable fileset clone by copying all the anodes (inodes) and marking all block pointers in the anodes copy-on-write. When Episode changes a block with a copy-on-write pointer, it allocates a new block and updates the block pointer in the active fileset.

Ext3cow [19] creates immutable file clones by maintaining a system-wide epoch and inode epochs. When an inode is modified, ext3cow allocates a new inode if the epoch in the inode is older than the last snapshot epoch. Also, each directory stores birth and death epochs for each entry. Ext3cow can render the view of the file system in a certain epoch by fetching entries alive at that epoch.

Btrfs [22] supports creating writable snapshot of the whole file system by copying the root of the sub-volume tree in its COW friendly B-trees [21]. Btrfs clones a file by sharing all extents of the file. The extent allocation tree records extents with back pointers to the inodes. Therefore, Btrfs is able to move an extent at a later time.

5.4 Versioning file systems

There are also versioning file systems that versions files and directories.

EFS [23] automatically versions files and directories. By allocating a new inode, it creates and finalizes a new file version when the file is opened and closed. Each versioned file has an inode log that keeps all versions of the file. All entries in directories have creation and deletion time.

CVFS [25] tries to store metadata versions more compactly. CVFS suggests two ways to save space in versioning file systems: 1. journal-based metadata that keeps a current version and an undo log to recover previous versions; 2. multiversion B-trees that keep all versions of metadata in a B-tree.

Versionfs [17] builds a stackable versioning file system. Instead of building a new file system, Versionfs adds the functionality of versioning on an existing file system by transferring certain operations to other operations. In this way, all versions of a file are maintained as different files in the underlying file system.

6 Research plan

Timeline	Work	Status
Apr. 2018	Preferential splitting	Done
Jun. 2018	`GOTO` messages	Done
Aug. 2018	`smart-pivot` messages	
Oct. 2018	Garbage collection	
Nov. 2018	Thesis writing	
Dec. 2018	Thesis defence	

Table 3: Plan for the research

References

[1] 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^{ε}-trees and write-optimization. *:login; Magazine 40*, 5 (Oct 2015), 22–28.

[2] Bonwick, J., and Moore, B. Zfs: The last word in file systems.

[3] Brodal, G. S., and Fagerberg, R. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (2003), pp. 546–554.

[4] Chutani, S., Anderson, O. T., Kazar, M. L., Leverett, B. W., Mason, W. A., Sidebotham, R. N., et al. The episode file system. In *Proceedings of the USENIX Winter 1992 Technical Conference* (1992), pp. 43–60.

[5] Conway, A., Bakshi, A., Jiao, Y., Zhan, Y., Bender, M. A., Jannen, W., Johnson, R., Kuszmaul, B. C., Porter, D. E., Yuan, J., and Farach-Colton, M. File systems fated for senescence? nonsense, says science! In *Proceedings of the 15th USENIX Conference on File and Storage Technologies* (2017), pp. 45–58.

[6] Docker. `https://www.docker.com`. Accessed: 2018-07-02.

[7] Dragga, C., and Santry, D. J. Gctrees: Garbage collecting snapshots. *Transactions on Storage 12*, 1 (2016), 4:1–4:32.

[8] Edwards, J. K., Ellard, D., Everhart, C., Fair, R., Hamilton, E., Kahn, A., Kanevsky, A., Lentini, J., Prakash, A., Smith, K. A., and Zayas, E. Flexvol: Flexible, efficient file volume virtualization in wafl. In *Proceedings of the 2008 USENIX Annual Technical Conference* (2008), pp. 129–142.

[9] Esmet, J., Bender, M. A., Farach-Colton, M., and Kuszmaul, B. C. The tokufs streaming file system. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage and File Systems* (2012).

[10] FUSE. `https://github.com/libfuse/libfuse`. Accessed: 2018-07-05.

[11] Hitz, D., Lau, J., and Malcolm, M. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference* (1994), pp. 19–19.

[12] Jannen, W., Yuan, J., Zhan, Y., Akshintala, A., Esmet, J., Jiao, Y., Mittal, A., Pandey, P., Reddy, P., Walsh, L., Bender, M., 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* (2015), pp. 301–315.

[13] 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. *ACM Transactions on Storage 11*, 4 (2015), 18:1–18:29.

[14] Konishi, R., Amagai, Y., Sato, K., Hifumi, H., Kihara, S., and Moriai, S. The linux implementation of a log-structured file system. *SIGOPS Operating Systems Review 40*, 3 (2006), 102–107.

[15] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., and Vivier, L. The new ext4 filesystem: current status and future plans. In *Proceedings of the 2007 Ottowa Linux Symposium* (2007), vol. 2, pp. 21–34.

[16] McKusick, M. K., and Ganger, G. R. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the 1999 USENIX Annual Technical Conference* (1999), pp. 24–24.

[17] Muniswamy-Reddy, K.-K., Wright, C. P., Himmer, A., and Zadok, E. A versatile and user-oriented versioning file system. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (2004), pp. 115–128.

[18] ONeil, P., Cheng, E., Gawlick, D., and ONeil, E. The log-structured merge-tree (lsm-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[19] Peterson, Z., and Burns, R. Ext3cow: A time-shifting file system for regulatory compliance. *Transactions on Storage 1*, 2 (2005), 190–212.

[20] Ren, K., and Gibson, G. A. Tablefs: Enhancing metadata efficiency in the local file system. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013), pp. 145–156.

[21] Rodeh, O. B-trees, shadowing, and clones. *ACM Transactions on Storage 3*, 4 (2008), 2:1–2:27.

[22] Rodeh, O., Bacik, J., and Mason, C. Btrfs: The linux b-tree filesystem. *Transactions on Storage 9*, 3 (2013), 9:1–9:32.

[23] Santry, D. S., Feeley, M. J., Hutchinson, N. C., Veitch, A. C., Carton, R. W., and Ofir, J. Deciding when to forget in the elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (1999), pp. 110–123.

[24] 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* (2013), pp. 17–30.

[25] Soules, C. A. N., Goodson, G. R., Strunk, J. D., and Ganger, G. R. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (2003), pp. 43–58.

[26] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G. Scalability in the xfs file system. In *Proceedings of the 1996 USENIX Annual Techinical Conference* (1996), pp. 1–1.

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

[28] Xu, J., Zhang, L., Memaripour, A., Gangadharaiah, A., Borase, A., Da Silva, T. B., Swanson, S., and Rudoff, A. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 478–496.

[29] 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 *Proceedings of the 14th USENIX Conference on File and Storage Technologies* (2016), pp. 1–14.

[30] 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. *ACM Transactions on Storage 13*, 1 (2017), 3:1–3:26.

[31] Zhan, Y., Conway, A., Jiao, Y., Knorr, E., Bender, M. A., Farach-Colton, M., Jannen, W., Johnson, R., Porter, D. E., and Yuan, J. The full path to full-path indexing. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies* (2018), pp. 123–138.