

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

Yang Zhan

Abstract

BetrFS indexes data and metadata by their full-paths. Therefore, logically related data and metadata are close to each other in the key space. In addition, the underlying data structure of BetrFS, the B^ϵ -tree, stores data and metadata in large nodes. Therefore, the locality in BetrFS is much better than traditional inode-based file systems. However, renames in full-path indexing file systems are ~~deemed~~ expensive because all keys involved in the rename must be updated. In the preliminary work, we show renames can be done efficiently in BetrFS by operating directly on B^ϵ -trees. I propose that, by introducing more operations to B^ϵ -trees, we can accelerate more namespace operations while keeping the locality from full-path indexing in BetrFS.

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. However, this schema doesn't impose any constraints 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 indexed **contiguous** in the key space. 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 indexing file systems. For example, a rename must update all the keys of key/value pairs **related to the rename**. The cost of a naive implementation that copies all the key/value pairs with updated keys and deletes old key/value pairs grows unboundedly with the size of the renamed file or directory.

In BetrFS 0.4, we introduce a new operation to B^ϵ -trees, **range-rename**, that accomplishes the work of file system rename with bounded IO cost. A **range-rename** slices out a source subtree from the **Betree** and moves the subtree to the destination. The idea of **range-rename** can also be applied to other tree

structures, like B-trees, though it is not write-optimized, not utilizing the write-optimization of B^ε -trees.

We can extend the idea of **range-rename** to support other namespace operations, like file clones. A file clone creates a target file whose data are identical to the source file. Instead of moving the sliced out subtree to the destination, we can share the subtree between the source and the destination for cloning.

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.

By 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.

Thus I propose that we can implement both efficient namespace operations and good locality in the full-path indexing write-optimized file system, 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. ~~At last~~, Section 6 presents the plan of milestones.

2 Background

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

2.1 B^ε -tree

The B^ε -tree [2, 1] is a write-optimized variant of the B-tree that cascades writes (Figure 1). Similar to the B-tree, the B^ε -tree stores key/value pairs in its leaves. The B^ε -tree also has pivots and pointers from a parent to its children. However, each interior node in the B^ε -tree has a buffer that cascades messages. A fresh write is injected as a message into the buffer in the B^ε -tree root. When the buffer of an interior node is full, the B^ε -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 messages, 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^ε -tree is much smaller.

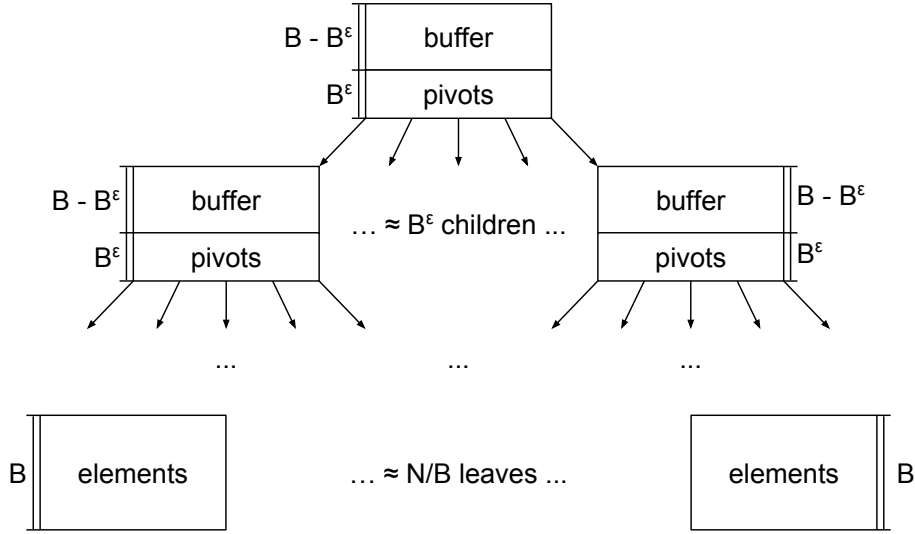


Figure 1: A B^ϵ -tree

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

Table 1: The asymptotic IO costs of B-tree and B^ϵ -tree

Table 1 shows the comparison of asymptotic IO costs between the B-tree and the B^ϵ -tree. A B^ϵ -tree with node size B partitions each interior node to have one $(B - B^\epsilon)$ -size buffer and B^ϵ pivots. The resulting tree height is $O(\log_{B^\epsilon} N) = O(\log_B N / \epsilon)$. A point query, which follows a root-to-leaf path, needs to perform $O(\log_B N / \epsilon)$ IOs. Similarly, a write is flushed $O(\log_B N / \epsilon)$ times, while each flush is done for at least $O((B - B^\epsilon) / B^\epsilon) = O(B^{1-\epsilon})$ writes. Thus, the amortized cost of each write is $O(\log_B N / (\epsilon B^{1-\epsilon}))$. If we set $\epsilon = 0.5$, the asymptotic IO costs of the B^ϵ -tree **are more promising than the B-tree**.

One important change in the B^ϵ -tree is that the IO costs of reads and writes are asymmetric. Read-before-write is deemed 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^ϵ -tree, most writes can be resolved in the root, so read-before-write can ruin the performance of the B^ϵ -tree.

To this end, **upserts** are introduced in the B^ϵ -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.

2.2 BetrFS

BetrFS [7, 8, 22, 23, 3, 24] is a Linux in-kernel file system built upon *ft-index* [20], which implements the B⁺-tree and exposes a key-value interface. BetrFS interacts with *ft-index* through point operations, such as **put**, **get** and **del**, as well as **range queries**. 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 achieve standard functionalities of 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 key space. This, combined with the large node size (4MB) used in B⁺-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. On the contrary, if the corresponding block is not in memory, BetrFS simply puts an **upsert** message describing the offset and the length of this write. As described in [section 2.1](#), when this message meets the old value during a flush, the change is applied.

2.3 Make BetrFS Better



Although the first version of BetrFS (BetrFS 0.1)  works great on certain benchmarks, for example, BetrFS 0.1 is 12.53x faster in creating 3 million small files (good small write performance) and 6.86x faster in doing `find` in the Linux source directory (good locality) than other file systems. There are problems from 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 ed down the B^ε -tree, it identifies all old messages in its range and **wipes them out** from the B^ε -tree.

2.3.2 Rename

Rename is a well-known 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. 

BetrFS 0.1 uses a simple approach that reads all related key/value pairs, **put**  them back with new keys and delete the old key/value pairs. The cost of this rename ~~is unbounded, it~~ 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, if 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 a 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 through 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 move

all related key/value pairs to a newly formed zone. Likewise, if the size of a zone becomes too small, BetrFS 0.2 does a zone merge.

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.

BetrFS 0.4 (BetrFS 0.3 contains bug fixes) introduces another rename solution that doesn't tax other operations and keeps full-path indexing. Unlike zoning, the new solution, **range-rename**, works directly on the B^ϵ -trees. BetrFS in the VFS layer simply 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 key space. Thus, if there exists an isolated subtree that contains and only contains all those indexed keys, we can move all the keys by moving the subtree around with a **pointer swing**.

There are two obstacles:

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

Range-rename addresses these two issues **by 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. The **caching** nature of the B^ϵ -tree means there can be the **remnant of deleted but uncleaned data**. 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) . Most nodes in the B^ϵ -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^ϵ -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**

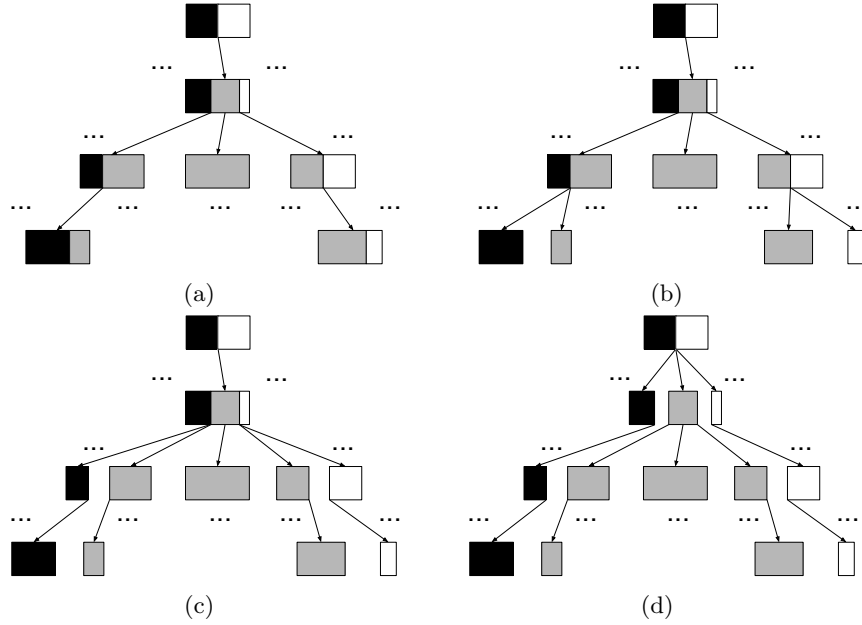


Figure 2: Slicing /gray between /black and /white



(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 rebalancing, 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 2 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^ε-tree is shown in Figure 2a, 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 2b where the leaves are split. At last, in Figure 2d, 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. **This is done during the 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^ε-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

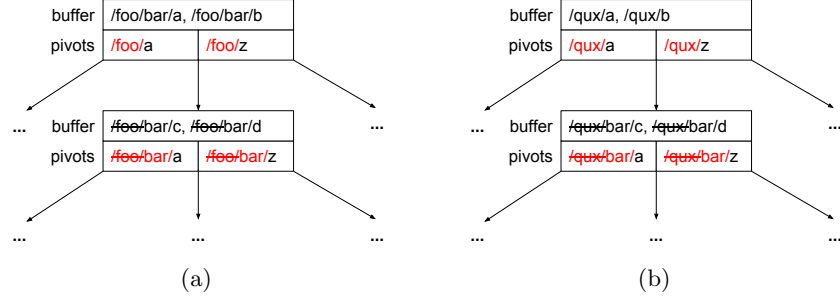


Figure 3: With key lifting, after moving the subtree to a new location, all prefix are updated automatically

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 $O(\text{tree height})$ 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 $O(\text{subtree size})$ IOs, the cost of tree surgery is bounded by the height of the B^ϵ -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^ϵ -tree correct. A naive approach would be traversing the subtree and updating all keys. This is costly and reverts the range-rename back to the unbounded $O(\text{subtree size})$.

Key lifting is introduced to deal with the key updating issue. Key lifting is based on observation that with the `memcmp` key comparison, all keys in a subtree that is bounded by two pivots in the parent must share the common prefix of the two pivots. This means that 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^ϵ -trees to lifted B^ϵ -trees. All operations walking down a root-to-leaf path must 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 3a, 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 3b shows the result of a subtree move. The subtree used to be bounded by “/foo/a” and “/foo/b” in Figure 3a, 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 **memorize** 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 $O(\text{tree height})$.

3.2 Range-rename evaluation

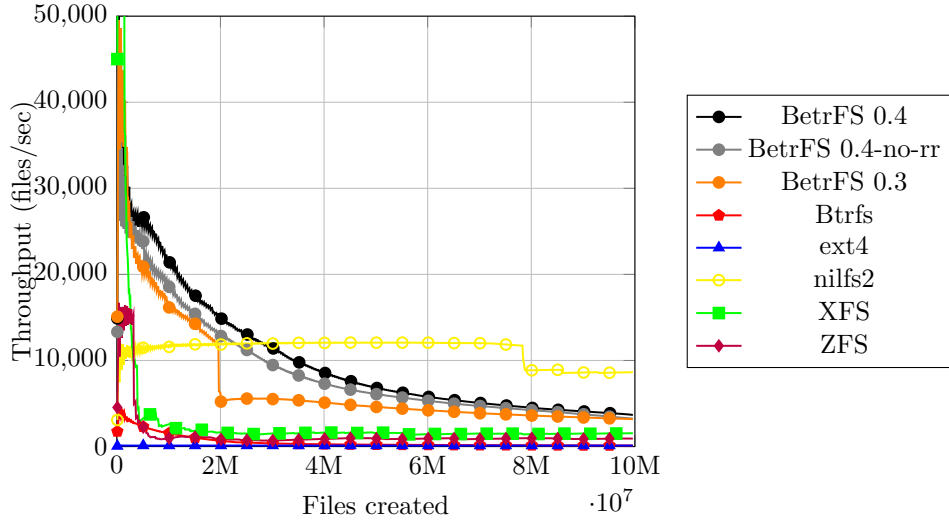


Figure 4: Cumulative file creation throughput during the Tokubench benchmark.

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 [16], ext4 [10], nilfs2 [9], XFS [19], and ZFS [13]. 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.

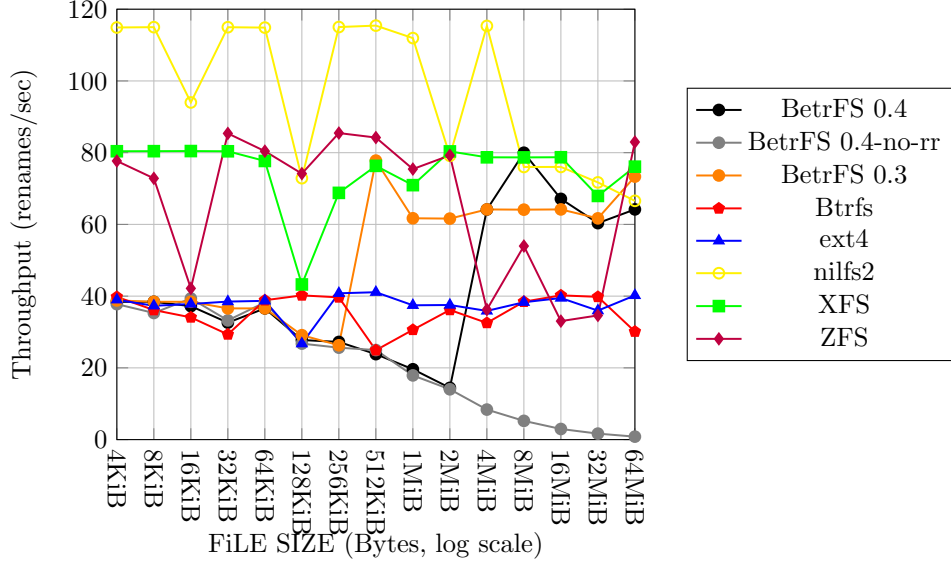



Figure 5: Rename throughput as a function of file size.

Figure 4 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 5 presents the result of our rename benchmark. In this benchmark, we create a file and then rename it 100 times.  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⁺-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.

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

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

4 Fast clones

Fast file or directory clones can be built in BetrFS on top of **range-rename**. Essentially, a rename can be viewed as a clone with a delete. And in **range-rename**, if we make both the source and destination point to the subtree, the content of the subtree is shared in the source and destination. When either the source or destination accumulates enough pending messages and wants to flush to the subtree, the B ^{ϵ} -tree can copy-on-write (COW) the root of the subtree and share the children in the two copies. In this way, BetrFS can gradually unshare the whole subtree.

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 can be applied to other tree structures like B-trees. However, it doesn't fit into the write-optimization of the betree 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** message, which specifies the destination range and the root of the subtree, is injected to the root.

GOTO messages would redirect queries. Normally, when a query for *key* reaches one non-leaf node, it would find the child whose range covers the *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

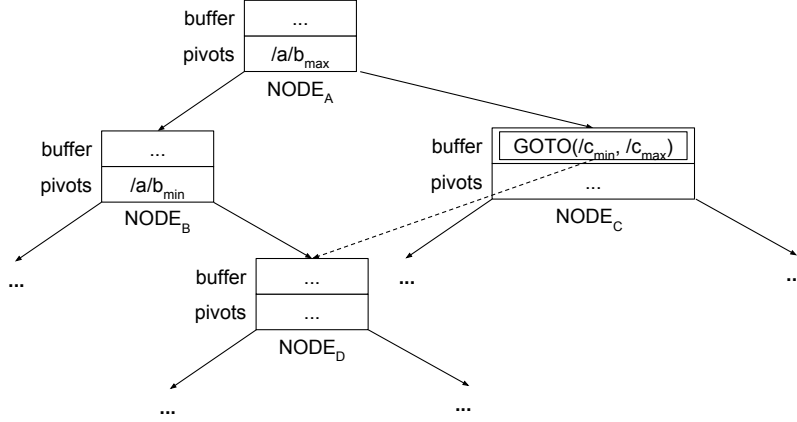


Figure 6: the GOTO message directs queries of $/c$ to another subtree

($/c_{min}, /c_{max}$) and points to the sliced out subtree whose range is $(/a/b_{min}, /a/b_{max})$. Figure 6 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 wipe out all old messages in its range, like a **range-delete** message. Additionally, we call **range-delete** messages as GOTO messages that point to nothing, thus generalize 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 $O(\text{tree height})$ 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. For example, the node contains pivots p_0, \dots, p_n , and $p_0 < \dots < p_i < sp_l < sp_r < p_j < p_n$ (sp_l and sp_r are keys in the GOTO message), the B^ϵ -tree would remove pivots p_{i+1}, \dots, p_{j-1} and add sp_l and sp_r 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 now GOTO messages can be flushed down the B^ϵ -tree, they still requires slicing out the source and destination subtrees. The source subtree needs to be sliced out because the GOTO message has to point to a subtree that is perfectly

in range. And when the B^ϵ -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 stem 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 the range (sp_l, sp_r) but it is allowed to 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. 

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^ϵ -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.

This problem can be solved by adding a background garbage collection thread that recycles all nodes in the subtree when a subtree is cover by a **range-delete** message. This thread only needs to read the header of a node, which contains the block numbers of its children and is much smaller than the whole node.

With the introduction of **smart-pivot** messages, the garbage collection problem becomes more complex. A **smart-pivot** message can hold a whole subtree before it becomes real pivots while only a small fraction of the subtree is used in the **smart-pivot** message. 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.

Alternative, 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 COWed 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 keys and unrelated keys. There,

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 related keys and unrelated keys in the descendants of the node. And there is no need to cut the B^ϵ -tree with sp_l or sp_r from this node on.

With preferential splitting, a leaf split, instead of selecting the key in the middle of the node as the new pivot, 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, if all key/value pairs are 4KB (in reality, this can be much less), there are 512 keys in the range, which means more than 500 key comparisons are needed.

We can do preferential splitting that only requires **reading** 2 keys, the smallest candidate key (at the $1/4$ of the leaf), the largest candidate key (at the $3/4$ of the leaf). This is based on the observation that the shortest common prefix of adjacent keys should be the same as the common prefix of the smallest candidate key and the largest candidate key. Based on that common prefix, we can construct a good pivot that falls in the range and **provides** as a nature boundary in the candidate keys. Additionally, we can also **provides** the middle key to make sure the pivot doesn't make the split too unbalanced.

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

5 Related work

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

WAFL [6] makes a snapshot by duplicate its root inode, which is the root of its whole file system tree structure, and uses block-map to record the block usage in the current file system and all snapshots. Later, FlexVol [5] in WAFL supports writable snapshots (cloning) of the whole file system by adding another layer of indirection between virtual block addresses in the file system and the physical block addresses on disks. Cloning is done by copying the `vol_info` block.

FFS [11] does snapshots by suspending all operations and creating a snapshot file. If a block shared by the snapshot and the current file system is modified, FFS copies the block to another location and store that location in the snapshot file.

In Nilfs2 [9], each logical segment ends with a checkpoint block, which points to the root of its B-tree structure. A snapshot is essentially a permanent checkpoint block.

Btrfs [16] supports cloning the whole file system by copying the root of the sub-volume tree in its COW friendly B-trees [15]. File cloning is achieved by copying all metadata about the file.

Copy-on-write a leaf in COW friendly B-trees needs to copy-on-write the whole root-to-leaf-path, HMOVFS [25] uses the stratified file system tree to keep the impact of an update to the minimal in NVMs. Also the refcount maintenance in COW friendly B-trees could be expensive, GCtrees [4] eliminate the need to store refcounts by chaining different versions of metadata block together.

NOVA-Fortis [21] also targets at NVMs. In NOVA-Fortis, each inode has private log that stores all changes in a linked list of 4KB pages. Snapshots of the file system are taken by incrementing the global snapshot ID. To recover the file system to a certain snapshot ID means in each inode log, only changes before that snapshot ID are played.

Ext3cow [14] supports file clones by using different inodes to store different versions of one file or directory. All versions have an epoch number and are chained in a linked list, thus ext3cow can restore the file system to a certain epoch.

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

EFS [17] automatically versions files and directories. File versioning is realized by appending the new inode to the file's inode log, and directories are stored as name logs, essentially storing all changes in the directory.

CVFS [18] tries to store metadata versions more compactly. CVFS suggests two ways to save space in versioning file systems: 1. journal-based metadata keeps a current version and an undo log to recover previous versions; 2. multi-version B-trees keeps all versions of metadata in a B-tree.

Versionfs [12] builds a versioning stackable file systems. Instead of building a new file system, versionfs add the functionality of versioning on an existing file system by transferring certain operations to other operations. In this way, all versions of a file is maintained as different files in the underlying file system.

6 Research plan

Timeline	Work
Apr. 2018	Preferential splitting
Jun. 2018	Smart pivots
Aug. 2018	Garbage collection
Sep. 2018	Thesis writing
Oct. 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] 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), SODA '03, pp. 546–554.
- [3] 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), FAST'17, pp. 45–58.
- [4] DRAGGA, C., AND SANTRY, D. J. Gctrees: Garbage collecting snapshots. *Transactions on Storage* 12, 1 (2016), 4:1–4:32.
- [5] 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 waff. In *USENIX 2008 Annual Technical Conference* (2008), ATC'08, pp. 129–142.
- [6] 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 on USENIX Winter 1994 Technical Conference* (1994), WTEC'94, pp. 19–19.
- [7] 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), FAST'15, pp. 301–315.
- [8] 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.
- [9] 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.

- [10] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., AND VIVIER, L. The new ext4 filesystem: current status and future plans. In *Ottawa Linux Symposium (OLS)* (2007), vol. 2, pp. 21–34.
- [11] MCKUSICK, M. K., AND GANGER, G. R. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (1999), ATEC '99, pp. 24–24.
- [12] 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), FAST '04, pp. 115–128.
- [13] ON LINUX, Z. <http://zfsonlinux.org/>.
- [14] PETERSON, Z., AND BURNS, R. Ext3cow: A time-shifting file system for regulatory compliance. *Transactions on Storage* 1, 2 (2005), 190–212.
- [15] RODEH, O. B-trees, shadowing, and clones. *ACM Transactions on Storage* 3, 4 (2008), 2:1–2:27.
- [16] RODEH, O., BACIK, J., AND MASON, C. Btrfs: The linux b-tree filesystem. *Transactions on Storage* 9, 3 (2013), 9:1–9:32.
- [17] 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 Seventeenth ACM Symposium on Operating Systems Principles* (1999), SOSP '99, pp. 110–123.
- [18] 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), FAST '03, pp. 43–58.
- [19] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the xfs file system. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference* (1996), ATEC '96, pp. 1–1.
- [20] TOKUTEK INC. ft-index. <https://github.com/Tokutek/ft-index>. Accessed: 2018-01-28.
- [21] XU, J., ZHANG, L., MEMARIPOUR, A., GANGADHARAI, 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), SOSP '17, pp. 478–496.

- [22] 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), FAST'16, pp. 1–14.
- [23] 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.
- [24] 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), FAST'18, pp. 123–138.
- [25] ZHENG, S., LIU, H., HUANG, L., SHEN, Y., AND ZHU, Y. Hmvfs: A versioning file system on dram/nvm hybrid memory. *Journal of Parallel and Distributed Computing* (2017).