A write-optimized cloning file system

Yang Zhan

Abstract

BetrFS uses write-optimized B^ε-trees and full-path indexing to achieve good random write performance and locality. However, full-path indexing has the longstanding problem of rename performance. In previous work, we show how we can operate directly on B^ε-trees to attain cheap renames with full-path indexing in BetrFS. I propose that, with the help of operations developed in the rename work, we can now extend BetrFS to support fast file or directory clones.

1 Introduction

Many modern file systems support snapshots, which create read-only copies of the whole file system. With snapshots, users can view files and directories in the file system at different times. Users can revert files and directories to earlier versions after they make mistakes.

Some file systems go further and support writable snapshots, clones, even at file-level granularity. File clones enable users to quickly duplicate a file without copying all data of the file and later modify both the original and cloned copies. File clones are particularly helpful for scenarios where some steady initial files are desired, for example, virtual machine images.

The desire for file clones can be seen in the recent trends of Linux interfaces. File clones are supported by a few file systems through the `ioctl` syscall. Now in the Linux kernel code, the file clone `ioctl` number used by btrfs, BTRFS_IOC_CLONE, is detected directly in the virtual file system (VFS) layer (as FICLONE), which means Linux wants to standardize the clone interface. Also, Linux supports reflink copy (`cp --reflink`) which tries to use cloning mechanism directly. System developers expect file clones to be more prevalent in the future.

In the BetrFS project, we explore the idea of using full-path indexing with write-optimized B^ε-trees. B^ε-trees offer BetrFS good random write performance, while full-path indexing provides good locality. However, rename performance is a notorious problem in full-path indexing file systems.

We addressed the rename issue while keeping full-path indexing intact through tree operations on B^ε-trees. These operations also open new opportunities for file clones as the rename we implemented can be viewed as a clone with a delete. Additionally, our clone approach can support directory clones, which must be done by traversing the whole directory in other file systems.

Data Structure	Insert	Point Query
B-tree	$O(log_B N)$	$O(log_B N)$
B$^\varepsilon$-tree	$O(log_B N/\varepsilon)$	$O(log_B N/\varepsilon B^{1-\varepsilon})$
B$^\varepsilon$-tree ($\varepsilon = 0.5$)	$O(2log_B N)$	$O(2log_B N/\sqrt{B})$

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

Thus, I propose implementing fast clones in BetrFS. In this work, we will make the cloning process more write-optimized, address the longstanding garbage collection problem in B$^\varepsilon$-trees and make node splits help the cloning process.

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$^\varepsilon$-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] is a write-optimized variant of the B-tree that c[illegible]des writes (Figure 1). Similar to the B-tree, the B$^\varepsilon$-tree stores key/value pairs on its leaves. The B$^\varepsilon$-tree also has pivots and pointers from a parent to its children. However, each interior node in the B[illegible]e has a buffer that cascades messages. A fresh write is injected as a message to the buffer [illegible]he 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 messages, the IO cost of each write is amortized. Compare to the B-tree where writes are put directly to leaves, the IO cost of 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 is done for at least $O((B - B^\varepsilon)/B^\varepsilon) = O(B^{1-\varepsilon})$ writes (the number of messages flushed to the child that can receive the most of messages should be no smaller than the average number of messages to each child). Thus, the amortized cost of each write is $O(log_B N/(\varepsilon B^{1-\varepsilon}))$. If we set $\varepsilon = 0.5$, the

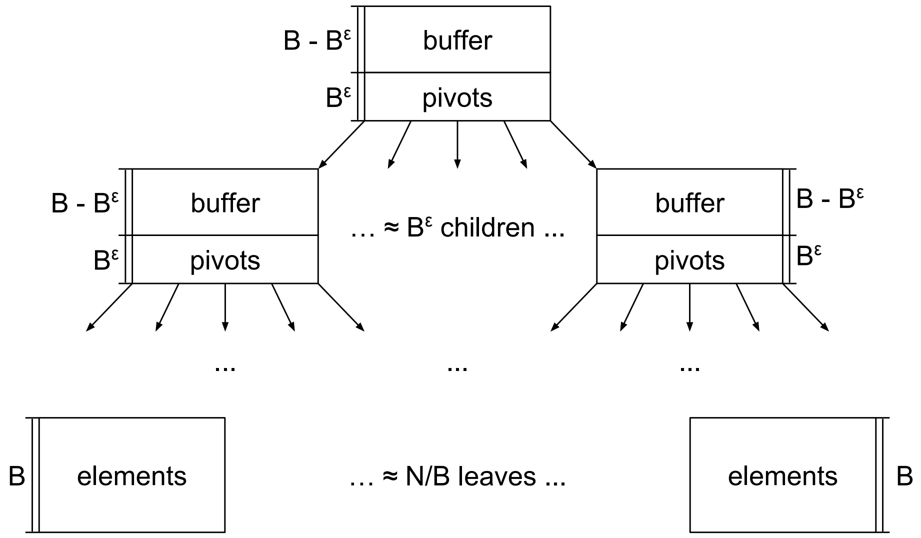

Figure 1: A B$^\varepsilon$-tree

asymptotic IO cost the B$^\varepsilon$-tree are more promising than the B-tree.

One important change in the B$^\varepsilon$-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$^\varepsilon$-tree, most writes can be resolved in the r so read-before-write can ruin the performance of the B$^\varepsilon$-tree, destroying the benefit of write optimization.

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 should be changed 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 change 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 [6, 7, 18, 19, 2, 20] is a Linux in-kernel file sys built upon *ft-index* [16], which implements B$^\varepsilon$-tree and exposes key-value interfaces. *Ft-index* su s point operations s as `put`, `get` and `del` as well as range es. It also provides transactions with which multiple operations can be executed atom y. A redo log and periodic checkpoints (every 5s) 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 assuming `libc` interfaces and `syscalls` are available. We built a shim layer called `klibc` which implements all functions *ft-index* requires. By implementing those functions in `klibc` instead of modifying all the functions used in *ft-index*, we are able to migrate ft-index into the kernel module without modifying it.

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 page cache.

Full-path indexing is desirable because of the locality it offers. With full-path indexing, all keys under one directory are contiguous in the key space. This, combined with the fact that the B$^\varepsilon$-tree uses large nodes (4MB), 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. This means 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. Before writing some data to a block, conventional file systems must first read the old block from the disk to the page cache and mark the page up-to-date. However, in BetrFS, read-before-write should be avoided. If the corresponding block is not in memory as an up-to-date page, instead of fetching the old block from the disk, BetrFS puts an `upsert` message describing the offset and the length of this write. As described in 2.1, when this message meets the old value during a flush, the change is applied. Thus, by removing this expensive read-modify-write, BetrFS reaches its full potential as a write-optimized file system.

2.3 Make BetrFS Better

Although the first version of BetrFS (BetrFS 0.1) works great on certain benchmarks, especially random write intensive ones, several issues immediately stand out. They are sequential write performance, delete performance and rename performance. We strived to solve these problems in the second version of BetrFS (BetrFS 0.2).

2.3.1 Sequential Write

One prominent problem in BetrFS 0.1 is the sequential write performance. Most conventional file systems can perform sequential writes close to disk bandwidth, but BetrFS 0.1 is much slower. For example, in the benchmark that writes a 1GB file, the throughput of BetrFS 0.1 is only 39% of that of ext4.

This is because we simply build on top of *ft-index*, which assumes itself working as a key-value store. In *ft-index*, every write is logged to ensure durability. This means all the 4KB writes are written twice, once to the redo log and once to the B$^\varepsilon$-tree. In a scenario as sequential writes where people usually want close-to-disk-bandwidth throughput, this double write quickly become the bottleneck.

This issue is resolved in BetrFS 0.2 by adding a `seq-put` interface to *ft-index*. When BetrFS figures out it is handling a huge amount of writes, it uses `seq-put` instead of `put`. A `seq-put`, unlike `put`, only generates a small entry on the redo log, which is supposed to point to the location in the B$^\varepsilon$-tree that stores the value. When a node in the B$^\varepsilon$-tree is written to the disk, it checks whether it contains `seq-put` messages and modifies the corresponding entries on the log. Another caveat is that before the redo log is flushed to disk, *ft-index* needs to figure out all unresolved `seq-put` entries and writes all related nodes to disk.

2.3.2 Delete

Another issue is file deletion. Because each file is split to 4KB blocks, 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` operation that deletes a whole range of key/value pairs. This is possible because the B$^\varepsilon$-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 to the root of the B$^\varepsilon$-tree. While the `range-delete` message is flushed down the B$^\varepsilon$-tree, it identifies all old messages in its range and wipes them out from the B$^\varepsilon$-tree.

2.3.3 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. And in this scenario, a rename means all keys in the source location should be updated. In a key-value store where all keys are sorted, all related key/value pairs must also be moved to the new location.

For renames, BetrFS 0.1 uses a simple approach that reads all related key/value pairs, put them back with the new keys and delete the old key/value pairs. This means the rename cost is unbounded, it increases when 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 the size of each zone according to that size. If the size of metadata or data associated to something that is not a zone grows too big (renaming it would cost more that zone size), BetrFS 0.2 does a zone split which move all related key/value pairs to a newly formed zone. If the size of a zone becomes too smaller, BetrFS 0.2 does a zone merge.

3 The full path to full-path indexing

Zoning, although it successfully fixes rename performance in BetrFS, has its drawbacks. The locality of zoning is not as good as in full-path indexing because, in zoning, full-path indexing is only maintained inside a zone. The zone maintenance cost is a more serious issue. If one operation incurs a zone split or merge, the cost of doing zone split or merge is charged to that operation. Thus, even a workload has nothing to do with renames, BetrFS 0.2 pays a significant amount of tax to zone maintenance.

Therefore, BetrFS 0.4 (BetrFS 0.3 contains bug fixes) introduces another way to deal with renames that keeps full-path indexing. Unlike zoning, the solution, `range-rename`, works directly on the underlying data structure, the B$^\varepsilon$-tree. BetrFS in the VFS layer simply calls `range-rename` implemented in *ft-index* to finish renames.

3.1 Range-rename

The idea comes from the observation that, with full-path indexing, all keys that need to be moved in a rename are contiguous in the key space. Thus, if there exists an isolated subtree that contains and only contains all those related keys. B$^\varepsilon$-tree is able to move that 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.

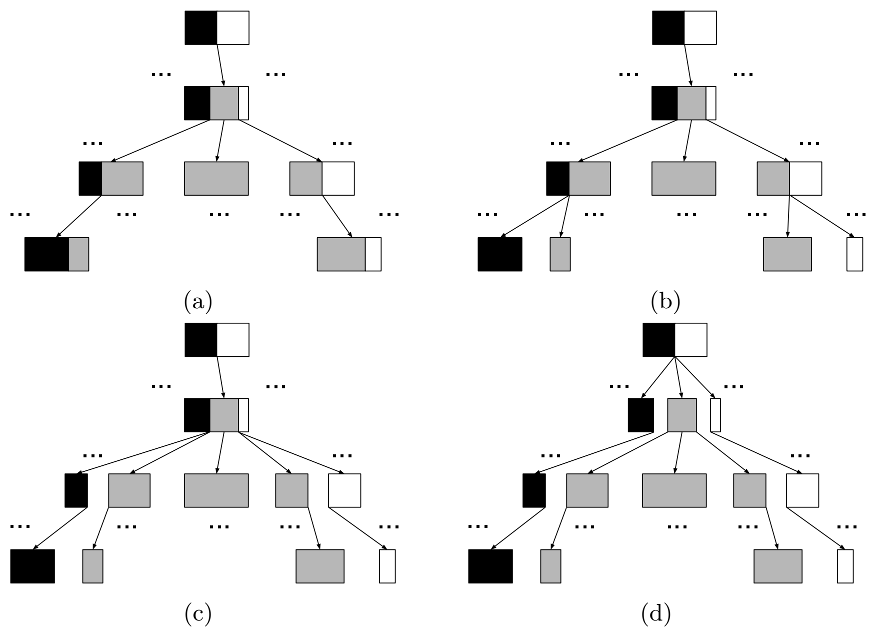

Figure 2: Slicing /gray between /black and /white

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 cascading nature of the B$^{\varepsilon}$-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$^{\varepsilon}$-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 common node that the *min* and *max* keys lead to. 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 separate related keys and unrelated keys from the bottom up until the LCA. Slicing uses the widely available tree mechanism, node splits, to accomplish this job. Unlike node splits in tree rebalancing that usually split the node evenly, slicing split the node with

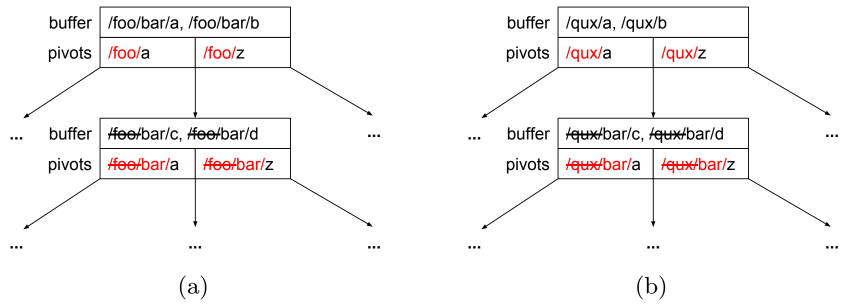

Figure 3: With key lifting, after moving the subtree to a new location, all prefix are updated automatically

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 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 a [illegible]ly contains and only contains gray keys. The original B$^\varepsilon$-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 changes 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$^\varepsilon$-tree assumes all leaves to be at the same depth. This means the source subtree and the destination subtree must be at 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 collect 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 and doesn't increase 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 be traversing the subtree and updating 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 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$^{\varepsilon}$-trees to lifted B$^{\varepsilon}$-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. All a root-to-leaf traversal needs to do is collecting lifted prefixes along the way, which 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 $O(tree\ height)$.

4 Towards fast clones

Our work on `range-rename` provides the building block for achieving fast file or directory clones in BetrFS. 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 sliced out source subtree, the content of the subtree is shared in the source and destination. When either source or destination accumulates enough changes and wants to flush to the subtree, the B$^{\varepsilon}$-tree can copy-on-write (COW) the root of the subtree while sharing the children in the two copies.

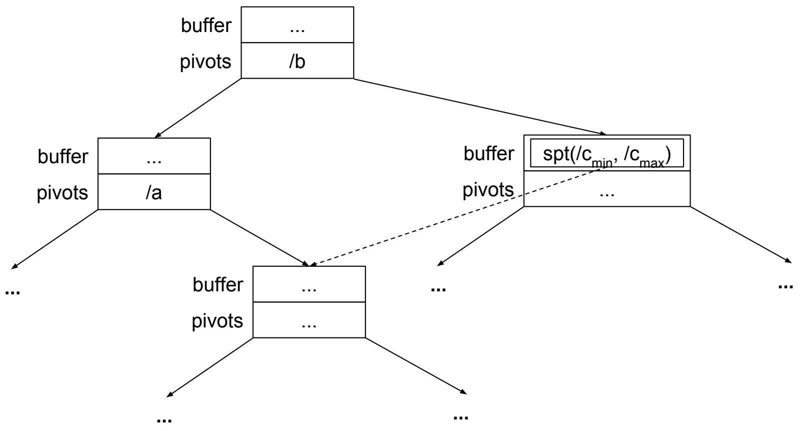

Figure 4: the `smart-pivot` message (spt) directs queries of /c to another subtree

4.1 Smart pivots

In `range-rename`, both the source and the destination subtrees need to be sliced out. 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 B$^\varepsilon$-tree.

Alternatively, we can introduce a new type of message, `smart-pivot`. A `smart-pivot` message points to a subtree with a certain range (sp_l, sp_r). To achieve better performance, the subtree doesn't need to be sliced out. Thus, a clone just flushes messages until the LCA in the source and inject a `smart-pivot` message pointing to the LCA.

A `smart-pivot` message would redirect operations. All reads that fall in the range of the `smart-pivot` message would be redirected to the subtree upon reaching the `smart-pivot` message. But because we don't want to do slicing, the lifted part in the subtree might not be the same as the common prefix of sp_l and sp_r and reads need to take care of the situation. Because writes are cascaded in the B$^\varepsilon$-tree, they would not be flushed to a deeper node in the B$^\varepsilon$-tree than the `smart-pivot` message.

As shown in Figure 4, suppose we clone /a/b to /c and the LCA of all keys in /a/b is the deepest node in the figure. All queries to /c go to the right child of the root, but when they encounter the `smart-pivot` message, they are directed to the LCA of /a/b.

Like other messages, a `smart-pivot` message is injected to the root of the B$^\varepsilon$-tree and flushed down with other messages. While it is flushed down the B$^\varepsilon$-tree, a `smart-pivot` message should wipe out all old messages, like a `range-delete` message. Eventually, a `smart-pivot` message should establish real pivots when it reaches the node of desired height (the height of its subtree).

When a `smart-pivot` message reaches the desired node, it should delete some pivots and add some pivots from the root of the subtree to the node. It might also need garbage collection some children of the desired node. We also need to take care of fringe nodes in the source and the destination. This might be done by a `range-delete` message and a `smart-pivot` message.

4.2 Garbage collection

Since the introduction of `range-delete` messages, ~~the~~ 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 by 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. In order to avoiding reading all nodes in the subtree from disk, we can store the children information in the block table (the block table maps node ids to physical locations). In this scenario, the garbage collection thread needs only consult the block table, which is much smaller and may fit in memory, to clean up all nodes.

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

As we've seen in previous sections, a lot of work attributes to handling fringe nodes, nodes that contains both related keys and unrelated keys. Therefore, it would be beneficial to reduce the number of such nodes.

Preferential splitting targets at making more pivots in the B$^\varepsilon$-tree be keys that might be used as boundary keys in clones or renames. Suppose we are cloning range (sp_0, sp_1), if one pivot in a node happens to be sp_0 or sp_1, 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$^\varepsilon$-tree with sp_0 or sp_1 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 readding 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 serves 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 [5] 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 [4] 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 [9] 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 [8], 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 [13] supports cloning the whole file system by copying the root of the sub-volume tree in its COW friendly B-trees [12]. 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, HMVFS [21] 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 [3] eliminate the need to store refcounts by chaining different versions of metadata block together.

NOVA-Fortis [17] also targets at NVMs. In NOVA-Fortis, each inode has private log that sotres 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 [11] 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 [14] 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 [15] 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 [10] 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 2: Plan for the research

References

[1] 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.

[2] 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.

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

[4] 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 *USENIX 2008 Annual Technical Conference* (2008), ATC'08, pp. 129–142.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

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

[14] 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.

[15] 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.

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

[17] 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), SOSP '17, pp. 478–496.

[18] 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.

[19] 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.

[20] 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.

[21] 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).