From B^{ε}-trees to BetrFS

B^{ε}-trees, like LSM-trees are an example of a write-optimized dictionary. BetrFS is a Linux file system that uses B^{ε}-trees as its on-disk data structure. The file system has gone through several iterations and is still under active development. We will focus on the first two versions, and discuss the trajectory of the ideas.

BetrFS v1

The key design principles of BetrFS are to:

- Avoid "*cryptosearches*". A cryptosearch is an implicit search triggered by an update.
  - To avoid cryptosearches, BetrFS tries to use "blind writes" whenever possible. Upserts enable blind writes in may situations
- Design a schema that lets BetrFS map efficient VFS operations to efficient B^{ε}-tree operations
  - BetrFS uses full-paths as its keys, and either the metadata (stat/inode) or a 4KiB data block as the value
  - This means that all data blocks for a file will be contiguous in the keyspace (and therefore contiguous in the leaf nodes)
  - This means that all metadata will be sorted in a breadth-first search order with respect to the file system directory structure
  - This also means that naively renaming or deleting an object will scale with the size of that object
    - Each data block must be moved from its old location in the tree structure to the new location, and there is no correspondence between B^{ε}-tree nodes and the tree structure that would let you just adjust pointers
    - Each key must be modified to reflect the new path
    - Each old value must be deleted

BetrFS v1 used B^{ε}-trees as a black box. It wanted to see how far the design could be pushed before changing (1) the OS and (2) the data structure.

BetrFS v2

BetrFS v2 set out to solve some of the biggest performance drawbacks in BetrFS v1. It introduced 3 new techniques:

- Range messages: range messages can be applied to multiple keys.
  - Range messages were used to speed up deletes by issuing one delete message regardless of the object size
- A late-binding journal. Logging data is key to B^{ε}-tree performance: small updates can be made persistent fast and enable batching in the data structure. But for large sequential writes, it may be cheaper to immediately write back a node rather than write the data twice (once to the log, and again in the tree).
- Tunable indirection with zones. Zones group regions of the file system directory tree, and the B^{ε}-trees keys are made relative to a zone's root
  - Renaming the root of the zone is fast, since no in-zone keys must be changed
  - Renaming within a zone is bounded by the size of the zone

BetrFS v2 still left the OS alone, but it made small changes to the data structure to alleviate some of the BetrFS v1 bottlenecks. Unfortunately it compromised on the full-path keys, which future versions addressed with more invasive data structure questions.

Thought Questions

2. Did the experiments clearly test one hypothesis?
3. Were the experiments fair?
4. What other tests would you like to see?
5. What are the best case operations for a path-based write-optimized file system?
6. What are the worst case operations for a path-based write-optimized file system?