

Leveraging btrfs transactions

Sage Weil
new dream network / DreamHost

Overview

- ❑ Btrfs background
- ❑ Ceph basics, storage requirements
- ❑ Transaction start/finish
- ❑ Snapshots
- ❑ Journaling

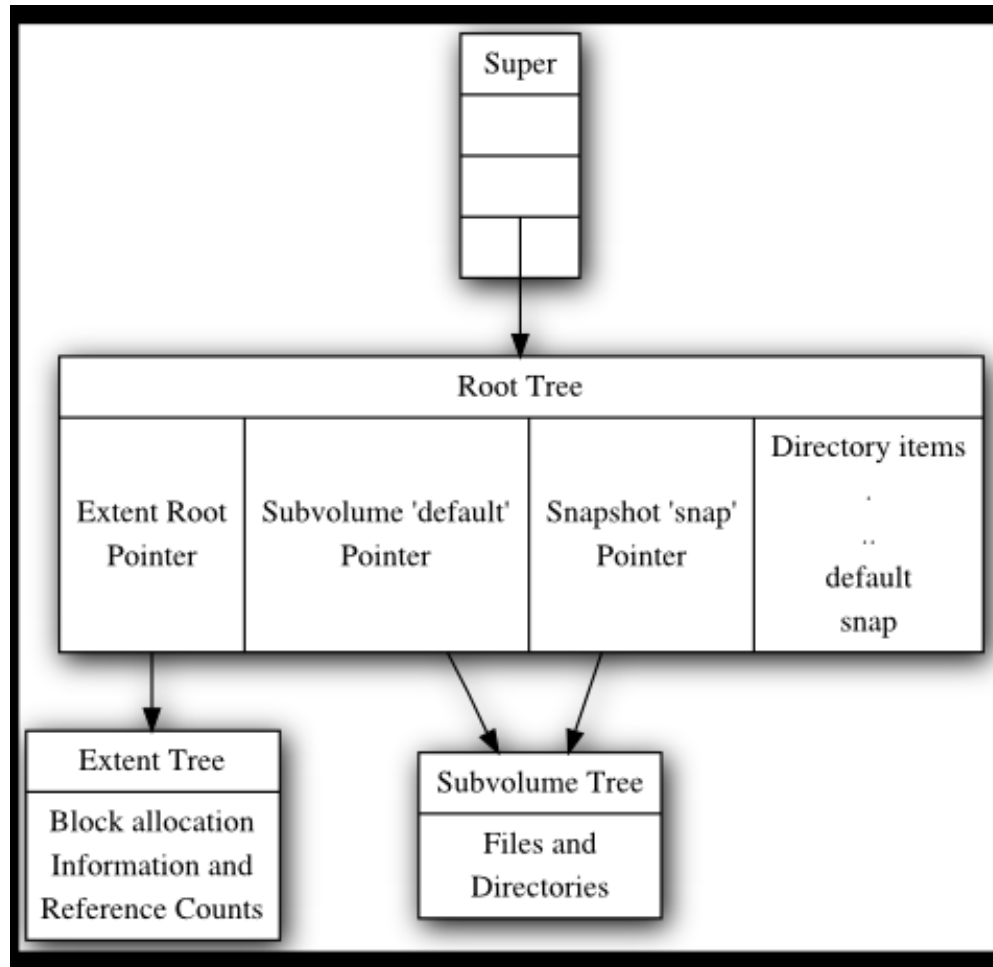
□ Featureful

- Extent based
- Space efficient packing for small files
- Integrity checksumming
- Writable snapshots
- Efficient incremental backups
- Multi-device support (striping, mirroring, RAID)
- Online resize, defrag, scrub/repair
- Transparent compression

btrfs trees

- ❑ Generic copy-on-write tree implementation
- ❑ Reference counting
- ❑ (Almost) everything is a key/value pair
 - ❑ Large data blobs outside of tree
- ❑ Data and metadata segregated on disk
- ❑ Transparent defrag before writeback

btrfs trees



btrfs copy-on-write trees

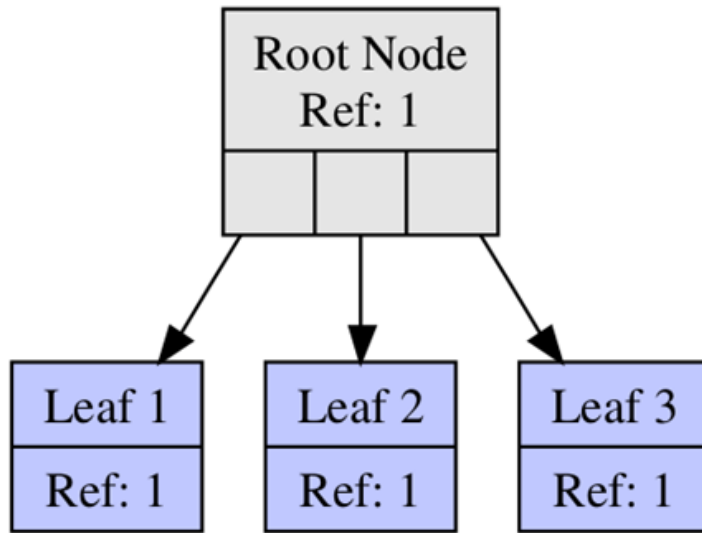


Figure: Before COW

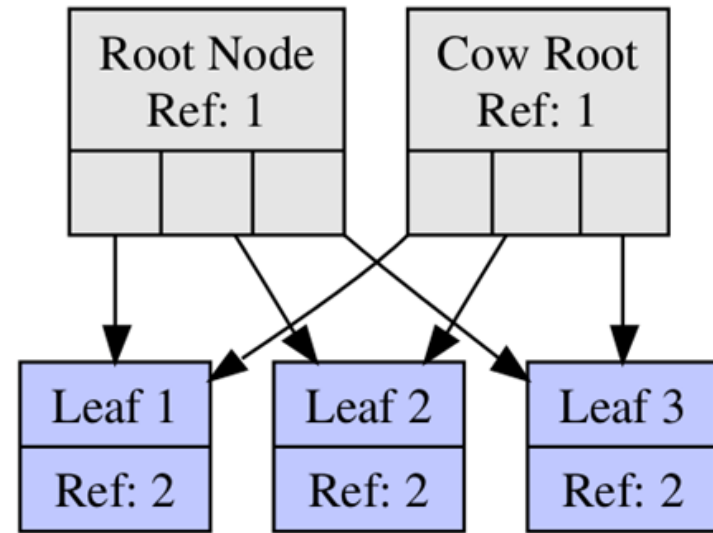


Figure: After COW

btrfs transaction commit

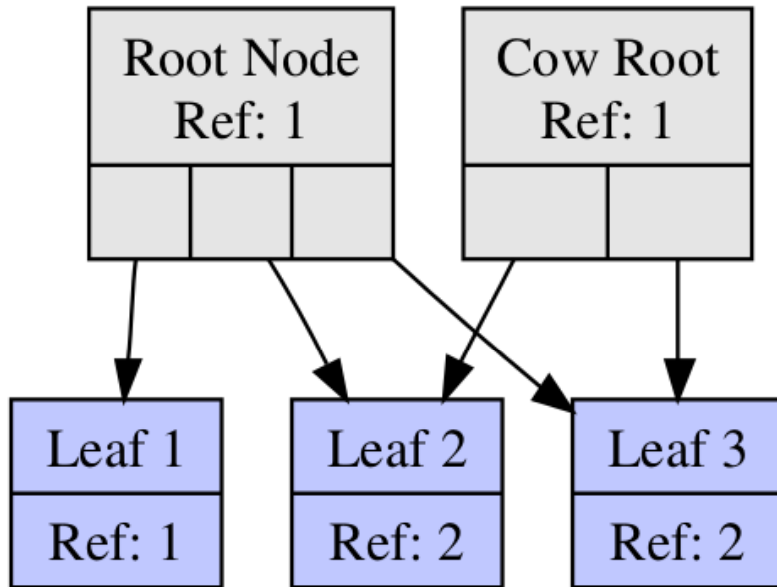


Figure: Modification after COW

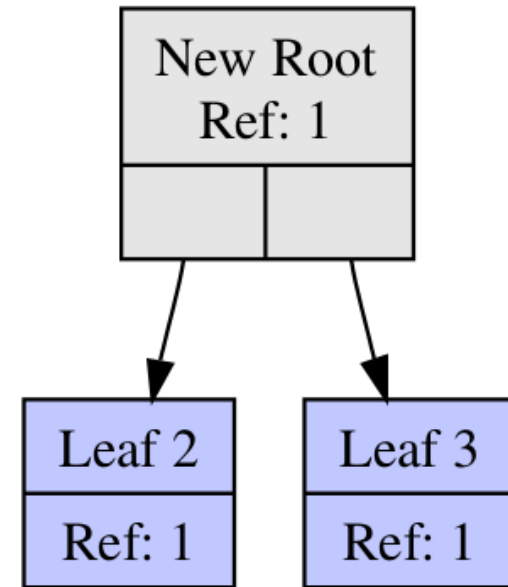


Figure: After Deletion

Ceph basics

- ❑ Scalable file system, block device, object store
 - ❑ Objects, not blocks
 - ❑ 100s to 1000s of storage bricks (OSDs)
 - ❑ Self-healing, self-managing, replicated, no SPOFs, etc.
- ❑ OSDs are smart
 - ❑ Peer to peer, loosely coordinated, strong consistency
 - ❑ Manage replication, recovery, data migration
 - ❑ Carefully manage consistency of locally stored objects

Consistency and safety

- ❑ All objects are replicated
 - ❑ Writes/updates apply to all replicas before ack
 - ❑ Nodes may fail at any time
 - ❑ They frequently recover
- ❑ We keep our local store in a consistent state
 - ❑ Know what we have
 - ❑ Know how that compares to what others have
 - ❑ So we can re-sync quickly
- ❑ Versioning and logs!

Atomicity

- ❑ Objects have version metadata
 - ❑ “Placement groups” have update logs
 - ❑ And writes are ordered
- ❑ Want atomic data+metadata commits
 - ❑ Object content
 - ❑ Version metadata
 - ❑ Log entry
- ❑ fsync(2) on extN isn't good enough

Transaction hooks

- ❑ Btrfs groups many operations into a single commit/transaction
- ❑ We added `ioctl(2)` to start/end transactions

```
#define BTRFS_IOC_TRANS_START _IO(BTRFS_IOCTL_MAGIC, 6)
#define BTRFS_IOC_TRANS_END   _IO(BTRFS_IOCTL_MAGIC, 7)
```

 - ❑ START pins the current transaction; END releases
 - ❑ User process can bracket sets of operations and know they will commit atomically
- ❑ Protects against node failures

Transaction hooks

- ❑ What about software errors?
 - ❑ By default, END is implied when the *fd* is closed
 - ❑ Software crash means partial transaction can reach disk
 - ❑ Mount option would disable implied END and intentionally wedge the machine
- ❑ No rollback

Compound operations

- ❑ Various interfaces proposed for compound kernel operations
 - ❑ Syslets – Info Molnar, ~2007
 - ❑ Btrfs usertrans ioctl – Me, ~2009
- ❑ Describe multiple operations via single syscall
 - ❑ Varying degrees of generality, flexibility
 - ❑ No worry about process completing transaction
- ❑ Need to ensure the operation will succeed
 - ❑ ENOSPC, EIO, EFAULT, bad transaction, etc.

Snapshots

- ❑ Granularity of durability is a btrfs transaction
 - ❑ e.g. a snapshot
- ❑ Explicitly manage btrfs commits from userspace
 - ❑ Do whatever operations we'd like
 - ❑ Quiesce writes
 - ❑ Take a snapshot of the btrfs subvolume
 - ❑ Repeat
- ❑ On failure/restart, roll back to last snapshot

Commit process

- ❑ Normal commit sequence
 - ❑ Block start of new transactions
 - ❑ Flush/perform delayed allocations, writeback
 - ❑ Make btree state consistent
 - ❑ Allow new transactions
 - ❑ Flush new trees
 - ❑ Update superblock(s) to point to new tree roots
- ❑ Want to minimize idle window

Async snapshot interface

□ New async snapshot ioctls

```
#define BTRFS_SUBVOL_CREATE_ASYNC    (1ULL << 0)
```

```
struct btrfs_ioctl_vol_args_v2 {  
    __s64 fd;  
    __u64 transid;  
    __u64 flags;  
    __u64 unused[4];  
    char name[BTRFS_SUBVOL_NAME_MAX + 1];  
};
```

```
#define BTRFS_IOC_SNAP_CREATE_V2 _IOW(BTRFS_IOCTL_MAGIC, 23, \  
    struct btrfs_ioctl_vol_args_v2)  
#define BTRFS_IOC_WAIT_SYNC _IOW(BTRFS_IOCTL_MAGIC, 22, __u64)
```

- ❑ Full btrfs commits have high latency
 - ❑ Tree flush, delayed allocation, superblock updates
 - ❑ Even fsync(2) has most of that
 - ❑ Poor IO pattern
- ❑ Ceph OSDs have independent journal
 - ❑ Separate device or file
 - ❑ Keep write latency low
 - ❑ Exploit SSDs, NVRAM, etc.

Journal mode

- ❑ Write-ahead
 - ❑ Any fs
 - ❑ Operations must be idempotent
- ❑ Parallel
 - ❑ Journal relative to a consistency point
 - ❑ Btrfs only
- ❑ Mask commit latency
- ❑ Optimizations
 - ❑ Clone large writes from journal to fs

Other useful bits

- ❑ Clone range
 - ❑ loctl to clone ranges of bytes between files
- ❑ Btrfs checksums
 - ❑ Will be used to validate Ceph intra-node scrub
- ❑ Multidevice support
 - ❑ Replication/stripping today, parity tomorrow
- ❑ Compression

- ❑ ENOSPC
- ❑ Slightly different commit path
 - ❑ Every commit is a snapshot commit
- ❑ Ceph replication masks some of it
- ❑ Improving test coverage

Questions

- ❑ <http://btrfs.wiki.kernel.org/>
- ❑ <http://ceph.newdream.net/>

