

# ***Btrfs***

Chris Mason

*Oracle*

chris.mason@oracle.com

## **Abstract**

Linux has a wealth of filesystems to choose from, but we are facing a number of challenges with scaling to the large storage subsystems that are becoming common in today's data centers. Filesystems need to scale in their ability to address and manage large storage, and also in their ability to detect, repair and tolerate errors in the data stored on disk.

The main Btrfs features include:

- Extent based file storage ( $2^{64}$  max file size)
- Space efficient packing of small files
- Space efficient indexed directories
- Dynamic inode allocation
- Writable snapshots
- Subvolumes (separate internal filesystem roots)
- Object level mirroring and striping
- Checksums on data and metadata (multiple algorithms available)
- Strong integration with device mapper for multiple device support
- Online filesystem check
- Very fast offline filesystem check
- Efficient incremental backup and FS mirroring

This paper describes the Btrfs implementation and future plans.

## Btrfs Design

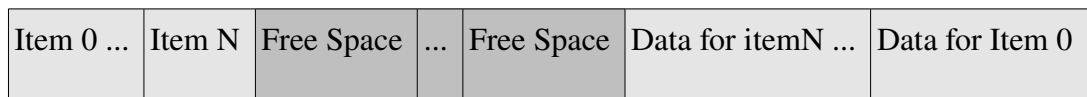
Btrfs is implemented with simple and well known constructs. It should perform well, but the long term goal of maintaining performance as the FS system ages and grows is more important than winning a short lived benchmark. To that end, benchmarks are being used to try and simulate performance over the life of a filesystem.

## Btree Data structures

The Btrfs btree provides a generic facility to store a variety of data types. Internally it only knows about three data structures: keys, items, and a block header:

<pre>struct btrfs_header {     u8 csum[32 bytes];     u8 fsid[16];     __le64 blocknr;     __le64 generation;     __le64 owner;     __le16 nritems;     __le16 flags;     u8 level; }</pre>	<pre>struct btrfs_key {     u64 objectid;     u32 flags;     u64 offset; }</pre>	<pre>struct btrfs_item {     struct btrfs_disk_key key;     __le32 offset;     __le16 size; }</pre>
---	--	---

Upper nodes of the trees contain only [ key, block pointer ] pairs. Tree leaves are broken up into two sections that grow toward each other. Leaves have an array of fixed sized items, and an area where item data is stored. The offset and size fields in the item indicate where in the leaf the item data can be found. Example:



Item data is variably size, and various filesystem data structures are defined as different types of item data. Eight bits of the flags field in struct btrfs\_key indicates the type of data stored in the item.

The block header contains a checksum for the block contents, the uuid of the filesystem that owns the block, the level of the block in the tree, and the block number where this block is supposed to live. These fields allow the contents of the metadata to be verified when the data is read. A future format change will also store a 64 bit sequence number stored in the block and in the node pointer to that block, allowing Btrfs to detect phantom or misplaced writes on the media.

The checksum of the lower node is not stored in the node pointer to simplify the FS writeback code. The sequence number will be known at the time the block is inserted into the btree, but the checksum is only calculated before writing the block to disk. Using the sequence number will allow Btrfs to detect phantom writes without having to find and update the upper node each time the lower node checksum is updated.

The generation field corresponds to the transaction id that allocated the block, which enables easy incremental backups and is used by the copy on write transaction subsystem.

## Filesystem Data Structures

Each object in the filesystem has an objectid, which is allocated dynamically on creation. A free objectid is simply a hole in the key space of the filesystem btree; objectids that don't already exist in the tree. The objectid makes up the most significant bits of the key, allowing all of the items for a given filesystem object to be logically grouped together in the btree.

The offset field of the key stores indicates the byte offset for a particular item in the object. For file extents, this would be the byte offset of the start of the extent in the file. The flags field stores the item type information, and has extra room for expanded use.

## Inodes

Inodes are stored in struct `btrfs_inode_items` at offset zero in the key, and have a type value of one. Inode items are always the lowest valued key for a given object, and they store the traditional stat data for files and directories. The inode structure is relatively small, and will not contain embedded file data or extended attribute data. These things are stored in other item types.

## Files

Small files that occupy less than one filesystem block may be packed into the btree inside the extent item. In this case the key offset is the byte offset of the data in the file, and the size field of struct `btrfs_item` indicates how much data is stored. There may be more than one of these per file.

Larger files are stored in extents. struct `btrfs_file_extent_item` records a generation number for the extent and a [ disk block, disk num blocks ] pair to record the area of disk corresponding to the file. Extents also store the logical offset and the number of blocks used by this extent record into the extent on disk. This allows Btrfs to satisfy a rewrite into the middle of an extent without having to read the old file data first. For example, writing 1MB into the middle of a existing 128MB extent may result in three extent records:

[ old extent: bytes 0-64MB ], [ new extent 1MB ], [ old extent: bytes 65MB – 128MB ]

File data checksums are stored in the btree in a struct `btrfs_csum_item`. The offset of the key corresponds to the first byte offset in the file of the data checksummed. A single item may store a number of checksums. struct `btrfs_csum_items` are only used for file extents. File data inline in the btree is covered by the checksum at the start of the btree block.

If there are no struct `btrfs_csum_items` present, any data in the file extent is considered uninitialized and all zeros are returned on reads, which allows for file preallocation. The admin will be able to choose what happens when file checksums don't match, either -EIO can be returned or zero filled data.

## Directories

Directories are indexed in two different ways. For filename lookup, there is an index comprised of keys:

Directory Objectid	BTRFS_DIR_ITEM_KEY	64 bit filename hash
--------------------	--------------------	----------------------

The default directory hash used is TEA hash, although other hashes may be added later on. A flags field in the directory inode will indicate which hash is used for a given directory.

The second directory index is used by `readdir` to return data in inode number order. This more closely resembles the order of blocks on disk and generally provides better performance for reading data in bulk (backups, copies, etc). Also, it allows fast checking that a given inode is linked into a directory when verifying inode link counts. This index uses an additional set of keys:

Directory Objectid	BTRFS_DIR_INDEX_KEY	Inode Objectid
--------------------	---------------------	----------------

## Reference Counted Extents

Reference counting is the basis for the snapshotting subsystems. For every extent allocated to a btree or a file, Btrfs records the number of references in a struct `btrfs_extent_item`. The trees that hold these items also serve as the allocation map for blocks that are in use on the filesystem. Some trees are not reference counted and are only protected by a copy on write logging. However, the same type of extent items are used for all allocated blocks on the disk.

## Extent Block Groups

Extent block groups allow allocator optimizations by breaking the disk up into chunks of 256MB or more. For each chunk, they record information about the number of blocks available. Files and directories will have a preferred block group which they try first for allocations.

Block groups have a flag that indicate if they are preferred for data or metadata allocations, and at `mkfs` time the disk is broken up into alternating metadata (33% of the disk) and data groups (66% of the disk). As the disk fills, a group's preference may change back and forth, but Btrfs always tries to avoid intermixing data and metadata extents in the same group. This substantially improves `fsck` throughput, and reduces seeks during writeback while the FS is mounted. It does slightly increase the seeks while reading.

## Extent Trees and DM integration

The Btrfs extent trees are intended to divide up the available storage into a number of flexible allocation policies. Each extent tree owns a section of the underlying disk, and they can be assigned to a collection of (or a single) tree roots, directories or inodes. Policies will direct how a given allocation is spread across the extent trees available, allowing the admin to direct which parts of the filesystem are striped, mirrored or confined to a given device.

Btrfs will try to tie in with DM in order to easily manage large pools of storage. The basic idea is to have at least one extent tree per spindle, and then allow the admin to assign those extent trees to subvolumes, directories or files.

## Snapshots and Subvolumes

Subvolumes are basically a named btree that holds files and directories. They have inodes inside the tree of tree roots and can have non-root owners and groups. Subvolumes can be given a quota of blocks, and once this quota is reached no new writes are allowed. All of the blocks and file extents inside of subvolumes are reference counted to allow snapshotting. Up to  $2^{64}$  subvolumes may be created on the FS.

Snapshots are identical to subvolumes, but their root block is initially shared with another subvolume. When the snapshot is taken, the reference count on the root block is increased, and the copy on write transaction system ensures changes made in either the snapshot or the source subvolume are private to that root. Snapshots are writable, and they can be snapshotted again any number of times. If read only snapshots are desired, their block quota is set to one at creation time.

## Btree Roots

Each Btrfs filesystem consists of a number of tree roots. A freshly formatted filesystem will have roots for:

- The tree of tree roots
- The tree of allocated extents
- The default subvolume tree

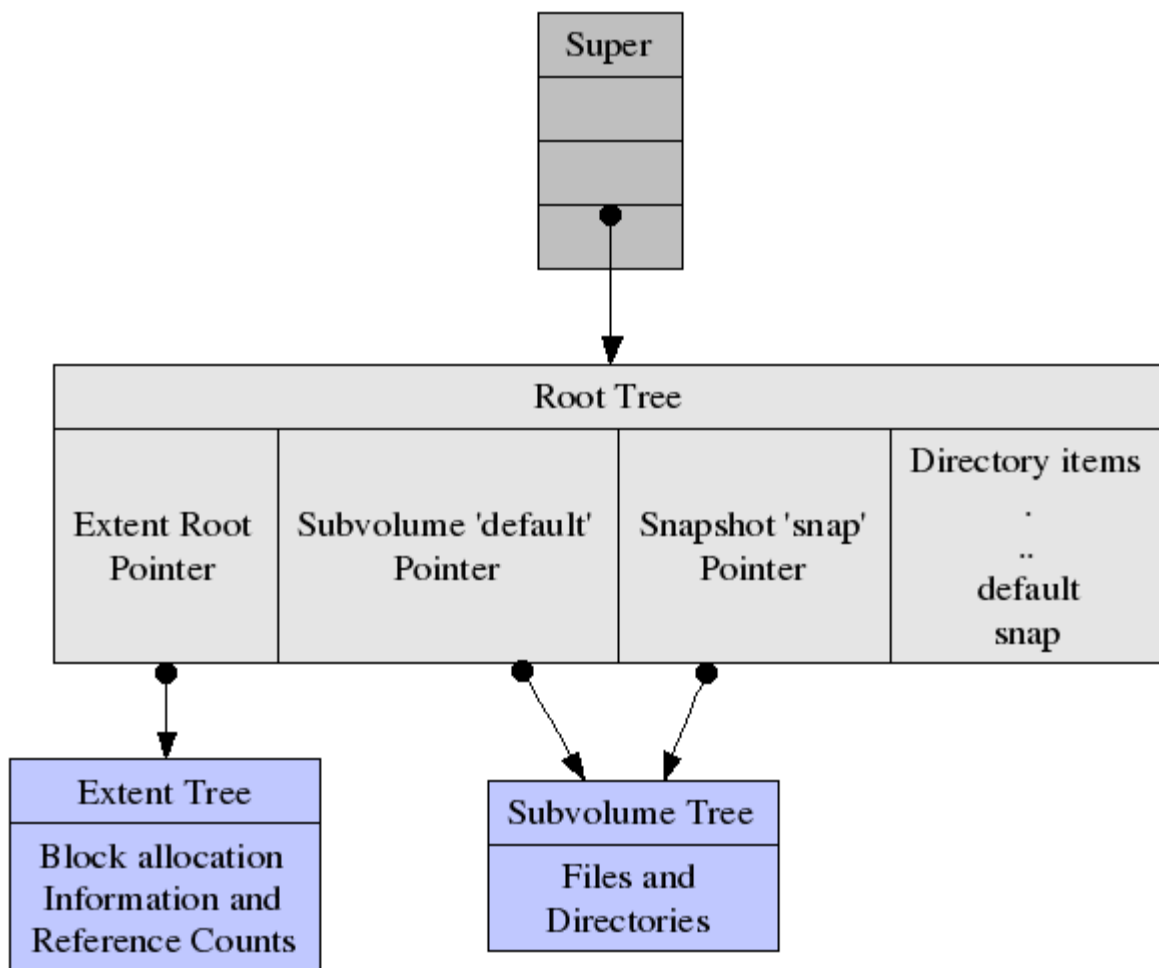
The tree of tree roots records the root block for the extent tree and the root blocks and names for each subvolume and snapshot tree. As transactions commit, the root block pointers are updated in this tree to reference the new roots created by the transaction, and then the new root block of this tree is recorded in the FS super block.

The tree of tree roots acts as a directory of all the other trees on the filesystem, and it has directory items recording the names of all snapshots and subvolumes in the FS. Each snapshot or subvolume has an objectid in the tree of tree roots, and at least one corresponding struct `btrfs_root_item`. Directory items in the tree map names of snapshots and subvolumes to these root items. Because the root item key is updated with every transaction commit, the directory items reference a generation number of `(u64)-1`, which tells the lookup code

to find the most recent root available.

The extent trees are used to manage allocated space on the devices. The space available can be divided between a number of extent trees to reduce lock contention and give different allocation policies to different block ranges.

The diagram below depicts a collection of tree roots. The super block points to the root tree, and the root tree points to the extent trees and subvolumes. The root tree also has a directory to map subvolume names to struct btrfs\_root\_items in the root tree. This filesystem has one subvolume named 'default' (created by mkfs), and one snapshot of 'default' named 'snap' (created by the admin some time later). In this example, 'default' has not changed since the snapshot was created and so both point tree to the same root block on disk.



## Copy on Write Logging

Data and metadata in Btrfs are protected with copy on write logging (COW). Once the transaction that allocated the space on disk has committed, any new writes to that logical address in the file or btree will go to a newly allocated block, and block pointers in the btrees and super blocks will be updated to reflect the new location.

Some of the Btrfs trees do not use reference counting for their allocated space. This includes the root tree, and the extent trees. As blocks are replaced in these trees, the old block is freed in the extent tree. These

blocks are not reused for other purposes until the transaction that freed them commits.

All subvolume (and snapshot) trees are reference counted. When a COW operation is performed on a btree node, the reference count of all the blocks it points to is increased by one. For leaves, the reference counts of any file extents in the leaf are increased by one. When the transaction commits, a new root pointer is inserted in the root tree for each new subvolume root. The key used has the form:

Subvolume inode number	BTRFS_ROOT_ITEM_KEY	Transaction ID
------------------------	---------------------	----------------

The updated btree blocks are all flushed to disk, and then the super block is updated to point to the new root tree. Once the super block has been properly written to disk, the transaction is considered complete. At this time the root tree has two pointers for each subvolume changed during the transaction. One item points to the new tree and one points to the tree that existed at the start of the last transaction.

Any time after the commit finishes, the older subvolume root items may be removed. The reference count on the subvolume root block is lowered by one. If the reference count reaches zero, the block is freed and the reference count on any nodes the root points to is lowered by one. If a tree node or leaf can be freed, it is traversed to free the nodes or extents below it in the tree in a depth first fashion.

The traversal and freeing of the tree may be done in pieces by inserting a progress record in the root tree. The progress record indicates the last key and level touched by the traversal so the current transaction can commit and the traversal can resume in the next transaction. If the system crashes before the traversal completes, the progress record is used to safely delete the root on the next mount.

Ohad Rodeh describes (with pictures!) a similar reference counted snapshot algorithm here:

[http://www.cs.huji.ac.il/~orodeh/papers/LinuxFS\\_Workshop.pdf](http://www.cs.huji.ac.il/~orodeh/papers/LinuxFS_Workshop.pdf)

## **Btrfsck**

The filesystem checking utility is a crucial tool, but it can be a major bottleneck in getting systems back online after something has gone wrong. Btrfs aims to be tolerant of invalid metadata, and will avoid using metadata it determines to be incorrect. The disk format allows Btrfs to deal with most corruptions at run time, without crashing the system and without requiring offline filesystem checking.

An offline btrfsck is being developed, in part to help verify the filesystem during testing, and as an emergency tool to make sure the filesystem is safe for mounting. The existing tool only verifies the extent allocation maps, making sure that reference counts are correct and that all extents are accounted for. If the extent maps are correct, there is no risk of incorrectly writing over existing data or metadata as blocks are allocated for new use.

btrfsck is able to read metadata in roughly disk order. As it scans the btrees on disk, it collects the locations of nodes and leaves and pulls them from the disk in large sequential batches. For the most part, btrfsck is bound by the sequential read throughput of the storage, and it is able to take advantage of multi-spindle arrays. The price paid for the extra speed is more ram. Btrfsck uses about 3x more ram than ext2fsck.