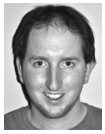
Btrfs

The Swiss Army Knife of Storage

JOSEF BACIK



Josef is the lead developer on Btrfs for Red Hat. He cut his teeth on the clustered file system GFS2, moving on to help maintain ext3 for Red Hat until Btrfs was publicly announced in 2007.
josef@redhat.com

Btrfs is a new file system for Linux that has been under development for four years now and is based on Ohad Rodeh's copy-on-write B-tree. Its aim is to bring more efficient storage management and better data integrity features to Linux. It has been designed to offer advanced features such as built-in RAID support, snapshotting, compression, and encryption. Btrfs also checksums all metadata and will checksum data with the option to turn off data checksumming. In this article I explain a bit about how Btrfs is designed and how you can use these new capabilities to your advantage.

Historically, storage management on Linux has been disjointed. You have the traditional mdadm RAID or the newer dmraid if you wish to set up software RAID. There is LVM for setting up basic storage management capable of having separate volumes from a common storage pool as well as the ability to logically group disks into one storage pool. Then on top of your storage pools you have your file system, usually an ext variant or XFS. The drawback of this approach is that it can get complicated when you want to change your setup. For example, if you want to add another disk to your logical volume in order to add more space to your home file system, you first must initialize and add that disk to your LVM volume group, then extend your logical volume, and only then extend your file system. Btrfs aims to fix this by handling all of this work for you. You simply run the command

```
# btrfs device add <device> <file system>
```

and you are done.

The same thing can be said about snapshotting. With LVM you must have free space in your volume group to create an overflow logical volume which will hold any of the changes to the source logical volume, and if this becomes full the volume becomes disabled. With Btrfs you are still limited to the free space in the file system, but you do not have to plan ahead and leave enough space in your file system in order to do snapshots. A simple df will tell you whether you have enough space to handle the changes to the source volume. Btrfs simply creates a new root and copies the source root information into the new root, allowing snapshot creation on Btrfs to take essentially the same time no matter how large the source volume.

Btrfs's B-tree

Btrfs breaks its metadata up into several B-trees. A B-tree is made up of nodes and leaves and has one or more levels. Information is stored in the tree and organized

by a key. Nodes contain the smallest key and the disk location of the node or leaf in the next level down. Leaves contain the actual data of the tree (see Figure 1).

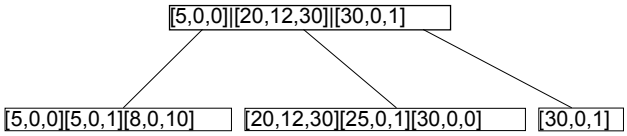


Figure 1: An example of a B-tree with four nodes and two levels

The top level, referred to as the root, acts just like a node. The entries in each node will tell you the first key in the node or leaf below it. In this example each key has three values, which is specific to Btrfs. We break the key up into objectID, the most important part of the key; the type, the second most important; and then the offset, which is the least important. So, as you can see in the above example, we have [30,0,0], which is smaller than [30,0,1]. This is important because for things such as files, we set the objectID to the inode number, and then any inode-specific information can also have the inode number as its objectID, allowing us to specify a different type and offset. Then any metadata related to the inode will be stored close to the actual inode information.

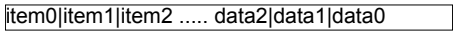


Figure 2: A leaf containing keys and data

The leaf's items contain the key and the size and offset of the data within the leaf (Figure 2). The items grow from the front of the leaf toward the end, and the data grows from the end of the leaf toward the front. Items can have an arbitrary data size, so you could potentially have one item in a leaf and have the rest of the leaf taken up with data.

This is a great advantage to Btrfs when it comes to dealing with small files. All Linux file systems address storage in arbitrary block sizes (e.g., 4 kilobytes). That has traditionally meant that if you create a file that is less than 4 kilobytes you will be wasting the leftover space. With Btrfs we can stash these smaller files directly into our B-tree leaves, so you will have the inode information and the data in the same block, which gives Btrfs a tremendous performance advantage when creating and reading small files.

The following is a basic list of the B-trees you get with a newly created file system:

- Tree root B-tree: This keeps the location of the roots of all of the various B-trees.
- Chunk B-tree: This keeps track of which chunks of the devices are allocated and to what type.
- Extent B-tree: This tree keeps track of all of the extents allocated for the system and their reference counts.
- Checksum B-tree: This tree stores the checksums of all of the data extents in the file system.
- File system B-tree: This holds the actual filesystem information, the file, and directory information.

We have all of these various B-trees to allow us a great deal of flexibility. For example, instead of having to come up with ways to stash extent reference count

information alongside file information, we simply store the two different sets of data in different trees. This makes everything easier for the developers and gives us a nice set of rules for offline error recovery. If we know how each tree should look generally, it makes it trivial to build tools to put things back together if some sort of catastrophic failure occurs.

Snapshotting

Btrfs's snapshotting is simple to use and understand. The snapshots will show up as normal directories under the snapshotted directory, and you can cd into it and walk around like in a normal directory. By default, all snapshots are writeable in Btrfs, but you can create read-only snapshots if you so choose. Read-only snapshots are great if you are just going to take a snapshot for a backup and then delete it once the backup completes. Writeable snapshots are handy because you can do things such as snapshot your file system before performing a system update; if the update breaks your system, you can reboot into the snapshot and use it like your normal file system.

When you create a new Btrfs file system, the root directory is a subvolume. Snapshots can only be taken of subvolumes, because a subvolume is the representation of the root of a completely different filesystem tree, and you can only snapshot a filesystem tree. The simplest way to think of this would be to create a subvolume for /home, so you could snapshot / and /home independently of each other. So you could run the following command to create a subvolume:

```
btrfs subvolume create /home
```

And then at some point down the road when you need to snapshot /home for a backup, you simply run

```
btrfs subvolume snapshot /home/ /home-snap
```

Once you are done with your backup, you can delete the snapshot with the command

```
btrfs subvolume delete /home-snap/
```

The hard work of unlinking the snapshot tree is done in the background, so you may notice I/O happening on a seemingly idle box; this is just Btrfs cleaning up the old snapshot. If you have a lot of snapshots or don't remember which directories you created as subvolumes, you can run the command

```
# btrfs subvolume list /mnt/btrfs-test/
ID 267 top level 5 path home
ID 268 top level 5 path snap-home
ID 270 top level 5 path home/josef
```

This doesn't differentiate between a snapshot and a normal subvolume, so you should probably name your snapshots consistently so that later on you can tell which is which.

Future Proofing

Btrfs uses 64 bits wherever possible to handle the various identifiers within the B-trees. This means that Btrfs can handle up to 2^{64} inodes, minus a couple of hundred for special items. This is a per filesystem tree limit, so you can create multiple subvolumes within the same file system and get even more inodes. Since

you can have a total of 2^{64} subvolumes, you could potentially have 2^{128} inodes in one file system, minus a negligible amount for reserved objects. This is scalability far above what could previously be achieved with a Linux file system.

The use of 64 bits also applies to how Btrfs addresses its disk space, enabling it to address up to 8 exabytes of storage. This makes Btrfs very future proof; it will be useful for many years to come as our storage capacities increase.

Directories

Directories and files look the same on disk in Btrfs, which is in keeping with the UNIX way of doing things. The ext file system variants have to pre-allocate their inode space when making the file system, so you are limited to the number of files you can create once you create the file system. With Btrfs we add a couple of items to the B-tree when you create a new file, which limits you only by the amount of metadata space you have in your file system.

If you have ever created thousands of files in a directory on an ext file system and then deleted the files, you may have noticed that doing an ls on the directory would take much longer than you'd expect given that there may only be a few files in the directory. You may have even had to run this command:

```
e2fsck -D /dev/sda1
```

to re-optimize your directories in ext. This is due to a flaw in how the directory indexes are stored in ext: they cannot be shrunk. So once you add thousands of files and the internal directory index tree grows to a large size, it will not shrink back down as you remove files. This is not the case with Btrfs. In Btrfs we store a file index next to the directory inode within the file system B-tree. The B-tree will grow and shrink as necessary, so if you create a billion files in a directory and then remove all of them, an ls will take only as long as if you had just created the directory.

Btrfs also has an index for each file that is based on the name of the file. This is handy because instead of having to search through the containing directory's file index for a match, we simply hash the name of the file and search the B-tree for this hash value. This item is stored next to the inode item of the file, so looking up the name will usually read in the same block that contains all of the important information you need. Again, this limits the amount of I/O that needs to be done to accomplish basic tasks.

Space Allocation

Like many other modern file systems, Btrfs uses delayed allocation to allow for better disk allocation. This means that Btrfs will only allocate space on the disk when the system decides it needs to get rid of dirty pages, so you end up with much larger allocations being made and much larger chunks of sequential data, which makes reading the data back faster.

Btrfs allocates space on its disks by allocating chunks, usually in 1 gigabyte chunks for data and 256 megabyte chunks for metadata. A chunk will have a specific profile associated with it: for example, it can be allocated for either data or metadata and then also have a RAID profile component. Once a chunk is allocated for either data or metadata, that space can only be used for one or the other. This allows Btrfs to have different allocation profiles for metadata and data.

For example, say you have a four-disk setup and you want to mirror your metadata but stripe your data. You can make your file system and specify RAID1 for metadata and RAID0 for data. Then whenever you write your metadata it will be mirrored across all of your disks, but when you write your data it will only be striped across the disks.

This split of metadata and data can be confusing to some users. A user may see that she has 10 gigabytes of data on her 16 gigabyte file system but only has 2 gigabytes free. In order to help clear up the confusion, Btrfs has its own df command which will show exactly how the space on the file system is used. Here is an example output from a full 7 gigabyte file system:

```
# btrfs filesystem df /mnt/btrfs-test/
Data: total=6.74GB, used=6.74GB
System: total=4.00MB, used=4.00KB
Metadata: total=264.00MB, used=121.34MB
```

This only shows allocated chunks and their usage amount. So with the above file system, if I add a disk with

```
# btrfs device add /dev/sdc /mnt/btrfs-test/
```

and then re-run `btrfs filesystem df`, I will see basically the same thing:

```
# btrfs filesystem df /mnt/btrfs-test/
Data: total=6.74GB, used=6.74GB
System: total=4.00MB, used=4.00KB
Metadata: total=264.00MB, used=121.35MB
```

This is because the new disk I added has not been allocated for either data or metadata. So I can use another command, `btrfs filesystem show`, and see the following:

```
# btrfs filesystem show /dev/sdb
Label: none uuid: 5eb80e04-26b9-4bb2-bd0f-a90a94464d6b
        Total devices 2 FS bytes used 6.86GB
        devid 1 size 7.00GB used 7.00GB path /dev/sdb
        devid 2 size 2.73TB used 0.00 path /dev/sdc
```

The size value is the size of the disk, and the used value is the size of the chunks allocated on that disk. So the new disk is 2.73 TB but hasn't had any chunks allocated from the disk, potentially allowing 2.73 TB of free space for allocation. You will see this reflected in the normal df command:

```
# df -h
Filesystem   Size   Used   Avail  Use %   Mounted on
/dev/sdb     2.8T   6.9G   2.8T   1%      /mnt/btrfs-test
```

Once you add a device, it is generally a good idea to run a balance on the file system with the command:

```
# btrfs filesystem balance /mnt/btrfs-test
```

This command, which can be run at any time, is used to redistribute space and reclaim any wasted space. If you add a disk, running balance will make sure everything is spread evenly across the disks.

Checksumming

Since Btrfs does checksum all of its data, it uses several worker threads to offload this work. When writing big chunks of data, the work will be split up among all of the processors on the system to calculate the checksums of the chunks. The same happens for reading: on completion of the read, the pages are handed off to worker threads which calculate and verify the checksums of the data so that the work is spread out, and this makes checksumming a much smaller performance hit than normal. For metadata checksumming, the checksum is calculated at write time as well, but is stored at the front of the metadata block.

Checksumming is great because it keeps the file system from crashing the box by reading bogus data, and also allows users to know that they need to be looking for a new hard drive or new memory. If you have a RAID profile that gives you multiple copies of the same data or metadata, such as RAID1 or RAID10, Btrfs will automatically try one of the other mirrors so that it can find a valid block. If it does find a valid block, everything will continue on as normal and the application will be none the wiser. The checksum mismatch will be logged so the user or administrator can be aware of the problem. If there are no other mirrors or all of the other mirrors are corrupt as well, Btrfs will return an error and the application will deal with it accordingly.

Compression

Btrfs currently supports two compression methods, zlib and lzo, with lzo being the default. You simply mount the file system with

```
mount -o compress
```

and any new writes will be compressed. Sometimes small writes will not compress well and will actually require more space compressed than uncompressed. Btrfs will notice this sort of behavior and turn off compression on the file in an effort to give the user the best possible space usage while using compression. Sometimes this is not what the user wants, however, so it can be changed by using the mount option:

```
mount -o compress-force
```

This option will force Btrfs to always compress the data no matter how it looks when compressed. Generally speaking, Btrfs does a good job balancing what should and shouldn't be compressed. The benefit of this compression infrastructure is that it is well abstracted, which makes adding support for new compression algorithms relatively easy, and hopefully it will be used to add encryption support in the future.

Solid State Drives

Solid state drives are changing how we think about storage, and Btrfs is no exception. Btrfs will automatically detect if it is on an SSD and will appropriately adjust how it allocates space. On spinning disks it is important to get good data locality, that is, to store related data as close together as possible to reduce seeking. With SSDs this is not as much of an issue, since the seek penalty is almost nothing. So instead of Btrfs wasting CPU cycles trying to get good data locality on an SSD, it will simply keep track of the last used free space for an allocation and start its

search there. Btrfs also supports TRIM, but this is turned off by default until more vendors can handle it reliably and quickly.

Filesystem Consistency

Traditional Linux file systems have used journals to ensure metadata consistency after crashes or power failures. In the case of ext this means all metadata is written twice, once to the journal and then to its final destination. In the case of XFS this usually means that a small record of what has changed is written to the journal, and eventually the changed block is written to disk. If the machine crashes or experiences a power failure, these journals have to be read on mount and re-run onto the file system to make sure nothing was lost. With Btrfs everything is copied on write. That means whenever we modify a block, we allocate a new location on disk for it, make our modification, write it to the new location, and then free the old location. You either get the change or you don't, so you don't have to log the change or replay anything the next time you mount the file system after a failure—the file system will always be consistent.

Journaled file systems can only ensure metadata consistency by writing to the journal. If your application uses fsync() to ensure data integrity, any other metadata changes that have happened recently must also be written to the journal. If you have other threads on the system modifying lots of metadata, you will have inconsistent fsync() times on a journaled file system. On Btrfs there's a special B-tree called a tree log that we use for fsync(). Any time you call fsync() on a file, Btrfs will go through and find all of the metadata that is required for that given file, copy it to the tree log, and write out the tree log. Because other threads that are modifying the metadata on the system will not affect the application doing fsync(), the application should see consistent fsync() performance. The only exception is if there are multiple applications doing fsync() on the same file system. They will all be logged to the same tree log, but this is the same as on a journaled file system.

Performance

Btrfs strives to have great performance in all workloads, but some workloads work better than others. One area where Btrfs has problems is with random overwrite workloads (i.e., writing to a file and then writing over a part of that file, and doing this often and randomly). Because of the copy-on-write design, this will lead to bad fragmentation and could result in slow cold cache reading. There is work to fix this shortcoming, and you can mount with the option autodefrag and Btrfs will notice this behavior and attempt to defragment the file in the background.

Btrfs also has quite a bit of latency associated with doing direct I/O to files. This, coupled with the copy-on-write nature of the file system, means that any enterprise database workload is likely to be slower on Btrfs than on XFS or ext4.

Btrfs tries to provide the most consistent performance possible as the file system fills up. For example, Figure 3 (next page) shows a workload where 16 threads are creating 512-kilobyte files across 512 subdirectories on a 7 gigabyte disk with ext4, Btrfs, and XFS.

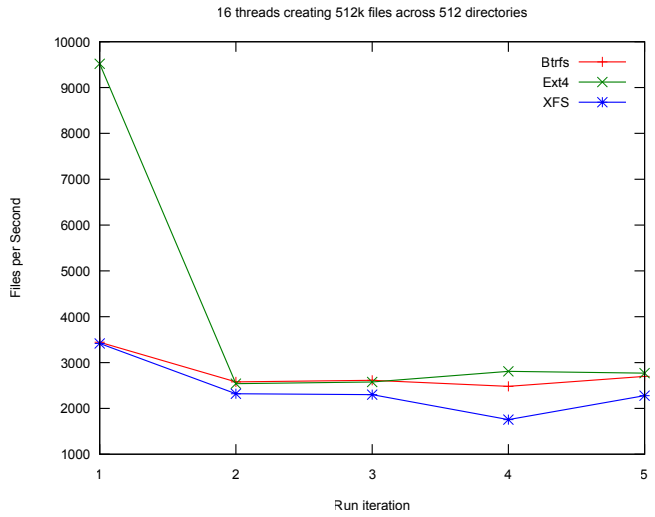


Figure 3: Btrfs, ext4, and XFS performance comparison

With small files Btrfs can really shine, since it will inline the data into the metadata. So you get a graph that looks like Figure 4.

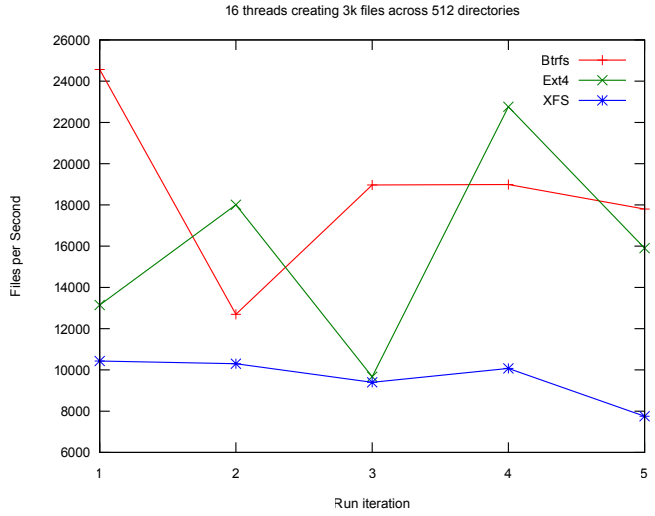


Figure 4: Btrfs, ext4, and XFS performance comparison using small files

Streaming writes onto Btrfs should be close to disk speeds. For example, writing 20 gigabytes directly to my local disk gives me a speed of 148 MB/s, and then writing to the same disk with Btrfs on top gives me 145 MB/s. Btrfs does very well at saturating the link to the disk.

Since Btrfs is still under heavy development, much of the effort is focused on finishing features and fixing stability issues. Performance is very much an important part of development, but, unlike XFS and ext4, Btrfs has not had years of widespread use to hammer out the kinks and optimize performance. In most common

workloads, Btrfs should perform much like its counterparts, but there is still a lot of work that needs to be done before it is a fair comparison.

Acknowledgments

Thanks to Chris Mason and Rik Farrow for checking the accuracy of this article and reading all of my drafts.