NAME

btrfs-balance − balance block groups on a btrfs filesystem

SYNOPSIS

btrfs balance *<subcommand> <args>*

DESCRIPTION

The primary purpose of the balance feature is to spread block groups across all devices so they match constraints defined by the respective profiles. See **mkfs.btrfs**(8) section *PROFILES* for more details. The scope of the balancing process can be further tuned by use of filters that can select the block groups to process. Balance works only on a mounted filesystem.

The balance operation is cancellable by the user. The on−disk state of the filesystem is always consistent so an unexpected interruption (eg. system crash, reboot) does not corrupt the filesystem. The progress of the balance operation is temporarily stored as an internal state and will be resumed upon mount, unless the mount option *skip_balance* is specified.

Warning

running balance without filters will take a lot of time as it basically rewrites the entire filesystem and needs to update all block pointers.

The filters can be used to perform following actions:

- convert block group profiles (filter *convert*)
- make block group usage more compact (filter *usage*)
- perform actions only on a given device (filters *devid*, *drange*)

The filters can be applied to a combination of block group types (data, metadata, system). Note that changing *system* needs the force option.

Note

the balance operation needs enough work space, ie. space that is completely unused in the filesystem, otherwise this may lead to ENOSPC reports. See the section *ENOSPC* for more details.

COMPATIBILITY

Note

The balance subcommand also exists under the **btrfs filesystem** namespace. This still works for backward compatibility but is deprecated and should not be used any more.

Note

A short syntax **btrfs balance** *<path>* works due to backward compatibility but is deprecated and should not be used any more. Use **btrfs balance start** command instead.

PERFORMANCE IMPLICATIONS

Balancing operations are very IO intensive and can also be quite CPU intensive, impacting other ongoing filesystem operations. Typically large amounts of data are copied from one location to another, with corresponding metadata updates.

Depending upon the block group layout, it can also be seek heavy. Performance on rotational devices is noticeably worse compared to SSDs or fast arrays.

SUBCOMMAND

cancel *<path>*

cancels a running or paused balance, the command will block and wait until the current blockgroup being processed completes

pause *<path>*

pause running balance operation, this will store the state of the balance progress and used filters to the filesystem

resume *<path>*
: resume interrupted balance, the balance status must be stored on the filesystem from previous run, eg. after it was forcibly interrupted and mounted again with *skip_balance*

start [options] *<path>*
: start the balance operation according to the specified filters, no filters will rewrite the entire filesystem. The process runs in the foreground.

Note

the balance command without filters will basically rewrite everything in the filesystem. The run time is potentially very long, depending on the filesystem size. To prevent starting a full balance by accident, the user is warned and has a few seconds to cancel the operation before it starts. The warning and delay can be skipped with *−−full−balance* option.

Please note that the filters must be written together with the *−d*, *−m* and *−s* options, because they're optional and bare *−d* etc also work and mean no filters.

Options

−d[*<filters>*]
: act on data block groups, see **FILTERS** section for details about *filters*

−m[*<filters>*]
: act on metadata chunks, see **FILTERS** section for details about *filters*

−s[*<filters>*]
: act on system chunks (requires *−f*), see **FILTERS** section for details about *filters*.

−v
: be verbose and print balance filter arguments

−f
: force a reduction of metadata integrity, eg. when going from *raid1* to *single*

−−background|−−bg
: run the balance operation asynchronously in the background, uses **fork**(2) to start the process that calls the kernel ioctl

status [−v] *<path>*
: Show status of running or paused balance.

If *−v* option is given, output will be verbose.

FILTERS

From kernel 3.3 onwards, btrfs balance can limit its action to a subset of the whole filesystem, and can be used to change the replication configuration (e.g. moving data from single to RAID1). This functionality is accessed through the *−d*, *−m* or *−s* options to btrfs balance start, which filter on data, metadata and system blocks respectively.

A filter has the following structure: *type*[=*params*][,*type*=...]

The available types are:

profiles=*<profiles>*
: Balances only block groups with the given profiles. Parameters are a list of profile names separated by "|" (pipe).

usage=*<percent>*, **usage=***<range>*
: Balances only block groups with usage under the given percentage. The value of 0 is allowed and will clean up completely unused block groups, this should not require any new work space allocated. You may want to use *usage=0* in case balance is returning ENOSPC and your filesystem is not too full.

The argument may be a single value or a range. The single value *N* means *at most N percent used*, equivalent to *..N* range syntax. Kernels prior to 4.4 accept only the single value format. The minimum range boundary is inclusive, maximum is exclusive.

devid=<*id*>
Balances only block groups which have at least one chunk on the given device. To list devices with ids use **btrfs filesystem show**.

drange=<*range*>
Balance only block groups which overlap with the given byte range on any device. Use in conjunction with *devid* to filter on a specific device. The parameter is a range specified as *start..end*.

vrange=<*range*>
Balance only block groups which overlap with the given byte range in the filesystem's internal virtual address space. This is the address space that most reports from btrfs in the kernel log use. The parameter is a range specified as *start..end*.

convert=<*profile*>
Convert each selected block group to the given profile name identified by parameters.

Note
starting with kernel 4.5, the *data* chunks can be converted to/from the *DUP* profile on a single device.

Note
starting with kernel 4.6, all profiles can be converted to/from *DUP* on multi−device filesystems.

limit=<*number*>, **limit=**<*range*>
Process only given number of chunks, after all filters are applied. This can be used to specifically target a chunk in connection with other filters (*drange*, *vrange*) or just simply limit the amount of work done by a single balance run.

The argument may be a single value or a range. The single value *N* means *at most N chunks*, equivalent to *..N* range syntax. Kernels prior to 4.4 accept only the single value format. The range minimum and maximum are inclusive.

stripes=<*range*>
Balance only block groups which have the given number of stripes. The parameter is a range specified as *start..end*. Makes sense for block group profiles that utilize striping, ie. RAID0/10/5/6. The range minimum and maximum are inclusive.

soft
Takes no parameters. Only has meaning when converting between profiles. When doing convert from one profile to another and soft mode is on, chunks that already have the target profile are left untouched. This is useful e.g. when half of the filesystem was converted earlier but got cancelled.

The soft mode switch is (like every other filter) per−type. For example, this means that we can convert metadata chunks the "hard" way while converting data chunks selectively with soft switch.

Profile names, used in *profiles* and *convert* are one of: *raid0*, *raid1*, *raid10*, *raid5*, *raid6*, *dup*, *single*. The mixed data/metadata profiles can be converted in the same way, but it's conversion between mixed and non−mixed is not implemented. For the constraints of the profiles please refer to **mkfs.btrfs**(8), section *PROFILES*.

ENOSPC

The way balance operates, it usually needs to temporarily create a new block group and move the old data there, before the old block group can be removed. For that it needs the work space, otherwise it fails for ENOSPC reasons. This is not the same ENOSPC as if the free space is exhausted. This refers to the space on the level of block groups, which are bigger parts of the filesystem that contain many file extents.

The free work space can be calculated from the output of the **btrfs filesystem show** command:

```
  Label: 'BTRFS'  uuid: 8a9d72cd−ead3−469d−b371−9c7203276265
          Total devices 2 FS bytes used 77.03GiB
          devid    1 size 53.90GiB used 51.90GiB path /dev/sdc2
          devid    2 size 53.90GiB used 51.90GiB path /dev/sde1
```

size − used = free work space 53.90GiB − 51.90GiB = 2.00GiB

An example of a filter that does not require workspace is *usage=0*. This will scan through all unused block groups of a given type and will reclaim the space. After that it might be possible to run other filters.

CONVERSIONS ON MULTIPLE DEVICES

Conversion to profiles based on striping (RAID0, RAID5/6) require the work space on each device. An interrupted balance may leave partially filled block groups that consume the work space.

EXAMPLES

A more comprehensive example when going from one to multiple devices, and back, can be found in section *TYPICAL USECASES* of **btrfs−device**(8).

MAKING BLOCK GROUP LAYOUT MORE COMPACT

The layout of block groups is not normally visible; most tools report only summarized numbers of free or used space, but there are still some hints provided.

Let’s use the following real life example and start with the output:

```
$ btrfs filesystem df /path
Data, single: total=75.81GiB, used=64.44GiB
System, RAID1: total=32.00MiB, used=20.00KiB
Metadata, RAID1: total=15.87GiB, used=8.84GiB
GlobalReserve, single: total=512.00MiB, used=0.00B
```

Roughly calculating for data, *75G − 64G = 11G*, the used/total ratio is about *85%*. How can we can interpret that:

- chunks are filled by 85% on average, ie. the *usage* filter with anything smaller than 85 will likely not affect anything
- in a more realistic scenario, the space is distributed unevenly, we can assume there are completely used chunks and the remaining are partially filled

Compacting the layout could be used on both. In the former case it would spread data of a given chunk to the others and removing it. Here we can estimate that roughly 850 MiB of data have to be moved (85% of a 1 GiB chunk).

In the latter case, targeting the partially used chunks will have to move less data and thus will be faster. A typical filter command would look like:

```
# btrfs balance start −dusage=50 /path
Done, had to relocate 2 out of 97 chunks

$ btrfs filesystem df /path
Data, single: total=74.03GiB, used=64.43GiB
System, RAID1: total=32.00MiB, used=20.00KiB
Metadata, RAID1: total=15.87GiB, used=8.84GiB
GlobalReserve, single: total=512.00MiB, used=0.00B
```

As you can see, the *total* amount of data is decreased by just 1 GiB, which is an expected result. Let’s see what will happen when we increase the estimated usage filter.

```
# btrfs balance start −dusage=85 /path
Done, had to relocate 13 out of 95 chunks

$ btrfs filesystem df /path
Data, single: total=68.03GiB, used=64.43GiB
System, RAID1: total=32.00MiB, used=20.00KiB
Metadata, RAID1: total=15.87GiB, used=8.85GiB
GlobalReserve, single: total=512.00MiB, used=0.00B
```

Now the used/total ratio is about 94% and we moved about *74G − 68G = 6G* of data to the remaining blockgroups, ie. the 6GiB are now free of filesystem structures, and can be reused for new data or metadata block groups.

We can do a similar exercise with the metadata block groups, but this should not typically be necessary, unless the used/total ratio is really off. Here the ratio is roughly 50% but the difference as an absolute number is "a few gigabytes", which can be considered normal for a workload with snapshots or reflinks updated frequently.

```
# btrfs balance start −musage=50 /path
Done, had to relocate 4 out of 89 chunks

$ btrfs filesystem df /path
Data, single: total=68.03GiB, used=64.43GiB
System, RAID1: total=32.00MiB, used=20.00KiB
Metadata, RAID1: total=14.87GiB, used=8.85GiB
GlobalReserve, single: total=512.00MiB, used=0.00B
```

Just 1 GiB decrease, which possibly means there are block groups with good utilization. Making the metadata layout more compact would in turn require updating more metadata structures, ie. lots of IO. As running out of metadata space is a more severe problem, it's not necessary to keep the utilization ratio too high. For the purpose of this example, let's see the effects of further compaction:

```
# btrfs balance start −musage=70 /path
Done, had to relocate 13 out of 88 chunks

$ btrfs filesystem df .
Data, single: total=68.03GiB, used=64.43GiB
System, RAID1: total=32.00MiB, used=20.00KiB
Metadata, RAID1: total=11.97GiB, used=8.83GiB
GlobalReserve, single: total=512.00MiB, used=0.00B
```

GETTING RID OF COMPLETELY UNUSED BLOCK GROUPS

Normally the balance operation needs a work space, to temporarily move the data before the old block groups gets removed. If there's no work space, it ends with *no space left*.

There's a special case when the block groups are completely unused, possibly left after removing lots of files or deleting snapshots. Removing empty block groups is automatic since 3.18. The same can be achieved manually with a notable exception that this operation does not require the work space. Thus it can be used to reclaim unused block groups to make it available.

```
# btrfs balance start −dusage=0 /path
```

This should lead to decrease in the *total* numbers in the **btrfs filesystem df** output.

EXIT STATUS

Unless indicated otherwise below, all **btrfs balance** subcommands return a zero exit status if they succeed, and non zero in case of failure.

The **pause**, **cancel**, and **resume** subcommands exit with a status of **2** if they fail because a balance operation was not running.

The **status** subcommand exits with a status of **0** if a balance operation is not running, **1** if the command−line usage is incorrect or a balance operation is still running, and **2** on other errors.

AVAILABILITY

btrfs is part of btrfs−progs. Please refer to the btrfs wiki **http://btrfs.wiki.kernel.org** for further details.

SEE ALSO

mkfs.btrfs(8), **btrfs−device**(8)