Notice

This document is for a development version of Ceph.

Report a Documentation Bug

BLUESTORE CONFIGURATION REFERENCE

DEVICES

BlueStore manages either one, two, or in certain cases three storage devices. These *devices* are "devices" in the Linux/Unix sense. This means that they are assets listed under `/dev` or `/devices`. Each of these devices may be an entire storage drive, or a partition of a storage drive, or a logical volume. BlueStore does not create or mount a conventional file system on devices that it uses; BlueStore reads and writes to the devices directly in a "raw" fashion.

In the simplest case, BlueStore consumes all of a single storage device. This device is known as the *primary device*. The primary device is identified by the `block` symlink in the data directory.

The data directory is a `tmpfs` mount. When this data directory is booted or activated by `ceph-volume`, it is populated with metadata files and links that hold information about the OSD: for example, the OSD's identifier, the name of the cluster that the OSD belongs to, and the OSD's private keyring.

In more complicated cases, BlueStore is deployed across one or two additional devices:

- A *write-ahead log (WAL) device* (identified as `block.wal` in the data directory) can be used to separate out BlueStore's internal journal or write-ahead log. Using a WAL device is advantageous only if the WAL device is faster than the primary device (for example, if the WAL device is an SSD and the primary device is an HDD).
- A *DB device* (identified as `block.db` in the data directory) can be used to store BlueStore's internal metadata. BlueStore (or more precisely, the embedded RocksDB) will put as much metadata as it can on the DB device in order to improve performance. If the DB device becomes full, metadata will spill back onto the primary device (where it would have been located in the absence of the DB device). Again, it is advantageous to provision a DB device only if it is faster than the primary device.

recommend using the available space as a WAL device. But if more fast storage is available, it makes more sense to provision a DB device. Because the BlueStore journal is always placed on the fastest device available, using a DB device provides the same benefit that using a WAL device would, while *also* allowing additional metadata to be stored off the primary device (provided that it fits). DB devices make this possible because whenever a DB device is specified but an explicit WAL device is not, the WAL will be implicitly colocated with the DB on the faster device.

To provision a single-device (colocated) BlueStore OSD, run the following command:

```
$ ceph-volume lvm prepare --bluestore --data <device>
```

To specify a WAL device or DB device, run the following command:

```
$ ceph-volume lvm prepare --bluestore --data <device> --block.wal <wal-device> --block.db
<db-device>
```

Note

The option `--data` can take as its argument any of the following devices: logical volumes specified using *vg/lv* notation, existing logical volumes, and GPT partitions.

PROVISIONING STRATEGIES

BlueStore differs from Filestore in that there are several ways to deploy a BlueStore OSD. However, the overall deployment strategy for BlueStore can be clarified by examining just these two common arrangements:

BLOCK (DATA) ONLY

If all devices are of the same type (for example, they are all HDDs), and if there are no fast devices available for the storage of metadata, then it makes sense to specify the block device only and to leave `block.db` and `block.wal` unseparated. The lvm command for a single `/dev/sda` device is as follows:

```
$ ceph-volume lvm create --bluestore --data /dev/sda
```

for a logical volume named `ceph-vg/block-lv` is as follows:

```
$ ceph-volume lvm create --bluestore --data ceph-vg/block-lv
```

BLOCK AND BLOCK.DB

If you have a mix of fast and slow devices (for example, SSD or HDD), then we recommend placing `block.db` on the faster device while `block` (that is, the data) is stored on the slower device (that is, the rotational drive).

You must create these volume groups and these logical volumes manually. The `ceph-volume` tool is currently unable to do so [create them?] automatically.

The following procedure illustrates the manual creation of volume groups and logical volumes. For this example, we shall assume four rotational drives (`sda`, `sdb`, `sdc`, and `sdd`) and one (fast) SSD (`sdx`). First, to create the volume groups, run the following commands:

```
$ vgcreate ceph-block-0 /dev/sda
$ vgcreate ceph-block-1 /dev/sdb
$ vgcreate ceph-block-2 /dev/sdc
$ vgcreate ceph-block-3 /dev/sdd
```

Next, to create the logical volumes for `block`, run the following commands:

```
$ lvcreate -l 100%FREE -n block-0 ceph-block-0
$ lvcreate -l 100%FREE -n block-1 ceph-block-1
$ lvcreate -l 100%FREE -n block-2 ceph-block-2
$ lvcreate -l 100%FREE -n block-3 ceph-block-3
```

Because there are four HDDs, there will be four OSDs. Supposing that there is a 200GB SSD in `/dev/sdx`, we can create four 50GB logical volumes by running the following commands:

```
$ vgcreate ceph-db-0 /dev/sdx
$ lvcreate -L 50GB -n db-0 ceph-db-0
$ lvcreate -L 50GB -n db-1 ceph-db-0
$ lvcreate -L 50GB -n db-2 ceph-db-0
$ lvcreate -L 50GB -n db-3 ceph-db-0
```


Finally, to create the four OSDs, run the following commands:

```
$ ceph-volume lvm create --bluestore --data ceph-block-1/block-1 --block.db ceph-db-0/db-1
$ ceph-volume lvm create --bluestore --data ceph-block-2/block-2 --block.db ceph-db-0/db-2
$ ceph-volume lvm create --bluestore --data ceph-block-3/block-3 --block.db ceph-db-0/db-3
```

After this procedure is finished, there should be four OSDs, `block` should be on the four HDDs, and each HDD should have a 50GB logical volume (specifically, a DB device) on the shared SSD.

SIZING

When deploying hybrid HDD and SSD OSDs, it is important to provision a large enough `block.db` logical volume for BlueStore.

We recommend when offloading WAL+DB to a faster device that the size of `block.db` be at least 2.5% of the size of the larger but slower `block` device.

When running a release older than Squid, RocksDB compression is not enabled, and larger offload shares were recommended. For RGW workloads, we recommended that the `block.db` be at least 4% of the `block` size, because RGW makes heavy use of `block.db` to store metadata (in particular, omap keys). For example, if the `block` size is 1TB, then `block.db` would have a size of at least 40GB. For RBD workloads, however, `block.db` usually needs no more than 1% to 2% of the `block` size.

In older releases, internal level sizes are such that the DB can fully utilize only those specific partition / logical volume sizes that correspond to sums of L0, L0+L1, L1+L2, and so on--that is, given default settings, sizes of roughly 3GB, 30GB, 300GB, and so on. Most deployments do not substantially benefit from sizing that accommodates L3 and higher, though DB compaction can be facilitated by doubling these figures to 6GB, 60GB, and 600GB. OSDs created before Pacific will benefit from using `ceph-bluestore-tool` to convert RocksDB to use sharding.

Improvements in Nautilus 14.2.12, Octopus 15.2.6, and subsequent releases allow for better utilization of arbitrarily-sized DB devices. Moreover, the Pacific release brings experimental dynamic-level support. Because of these advances, users of older releases might want to plan ahead by provisioning larger DB devices today so that the benefits of scale can be realized when upgrades are made in the future.

When *not* using a mix of fast and slow devices, there is no requirement to create separate logical volumes for `block.db` or `block.wal`. BlueStore will automatically colocate these within the space of `block`.

BlueStore can be configured to automatically resize its caches, provided that certain conditions are met: TCMalloc must be configured as the memory allocator and the `bluestore_cache_autotune` configuration option must be enabled (note that it is currently enabled by default). When automatic cache sizing is in effect, BlueStore attempts to keep OSD heap memory usage under a certain target size (as determined by `osd_memory_target`). This approach makes use of a best-effort algorithm and caches do not shrink smaller than the size defined by the value of `osd_memory_cache_min`. Cache ratios are selected in accordance with a hierarchy of priorities. But if priority information is not available, the values specified in the `bluestore_cache_meta_ratio` and `bluestore_cache_kv_ratio` options are used as fallback cache ratios.

`bluestore_cache_autotune`

Automatically tune the space ratios assigned to various BlueStore caches while respecting minimum values.

type: `bool`
runtime updatable: `true`
default: `true`
see also: `bluestore_cache_size`, `bluestore_cache_meta_ratio`

`osd_memory_target`

When TCMalloc is available and cache autotuning is enabled, try to keep this many bytes mapped in memory. Note: This may not exactly match the RSS memory usage of the process. While the total amount of heap memory mapped by the process should usually be close to this target, there is no guarantee that the kernel will actually reclaim memory that has been unmapped. During initial development, it was found that some kernels result in the OSD's RSS memory exceeding the mapped memory by up to 20%. It is hypothesised however, that the kernel generally may be more aggressive about reclaiming unmapped memory when there is a high amount of memory pressure. Your mileage may vary.

type: `size`
runtime updatable: `true`
default: `4Gi`
min: `896_M`
see also: `bluestore_cache_autotune`, `osd_memory_cache_mi`
`osd_memory_base`, `osd_memory_target_autotune`

The number of seconds to wait between rebalances when cache autotune is enabled. *bluestore_cache_autotune_interval* sets the speed at which Ceph recomputes the allocation ratios of various caches. Note: Setting this interval too small can result in high CPU usage and lower performance.

type: `float`
runtime updatable: `true`
default: `5.0`
see also: `bluestore_cache_autotune`

osd_memory_base

When TCMalloc and cache autotuning are enabled, estimate the minimum amount of memory in bytes the OSD will need. This is used to help the autotuner estimate the expected aggregate memory consumption of the caches.

type: `size`
runtime updatable: `true`
default: `768Mi`
see also: `bluestore_cache_autotune`

osd_memory_expected_fragmentation

When TCMalloc and cache autotuning are enabled, estimate the percentage of memory fragmentation. This is used to help the autotuner estimate the expected aggregate memory consumption of the caches.

type: `float`
runtime updatable: `true`
default: `0.15`
allowed range: `[0, 1]`
see also: `bluestore_cache_autotune`

osd_memory_cache_min

used for caches. Note: Setting this value too low can result in significant cache thrashing.

type:	`size`
runtime updatable:	`true`
default:	`128Mi`
min:	`128_M`
see also:	`bluestore_cache_autotune`

`osd_memory_cache_resize_interval`

When TCMalloc and cache autotuning are enabled, wait this many seconds between resizing caches. This setting changes the total amount of memory available for BlueStore to use for caching. Note that setting this interval too small can result in memory allocator thrashing and lower performance.

type:	`float`
runtime updatable:	`true`
default:	`1.0`
see also:	`bluestore_cache_autotune`

MANUAL CACHE SIZING

The amount of memory consumed by each OSD to be used for its BlueStore cache is determined by the `bluestore_cache_size` configuration option. If that option has not been specified (that is, if it remains at 0), then Ceph uses a different configuration option to determine the default memory budget: `bluestore_cache_size_hdd` if the primary device is an HDD, or `bluestore_cache_size_ssd` if the primary device is an SSD.

BlueStore and the other subsystems within the OSD make every effort to work within this memory budget. Note that in addition to the configured cache size, there is also memory consumed by the OSD itself. There is additional utilization due to memory fragmentation and other allocator overhead.

The configured cache-memory budget can be used to store the following types of things:

- Key/Value metadata (that is, RocksDB's internal cache)
- BlueStore metadata
- BlueStore data (that is, recently read or recently written object data)

`bluestore_cache_kv_ratio`. The fraction of the cache that is reserved for data is governed by both the effective BlueStore cache size (which depends on the relevant `bluestore_cache_size[_ssd|_hdd]` option and the device class of the primary device) and the "meta" and "kv" ratios. This data fraction can be calculated with the following formula: `<effective_cache_size> * (1 - bluestore_cache_meta_ratio - bluestore_cache_kv_ratio)`.

`bluestore_cache_size`

The amount of memory BlueStore will use for its cache. If zero, `bluestore_cache_size_hdd` or `bluestore_cache_size_ssd` will be used instead.

type: `size`
runtime updatable: `true`
default: `0B`

`bluestore_cache_size_hdd`

The default amount of memory BlueStore will use for its cache when backed by an HDD.

type: `size`
runtime updatable: `true`
default: `1Gi`
see also: `bluestore_cache_size`

`bluestore_cache_size_ssd`

The default amount of memory BlueStore will use for its cache when backed by an SSD.

type: `size`
runtime updatable: `true`
default: `3Gi`
see also: `bluestore_cache_size`

`bluestore_cache_meta_ratio`

Ratio of BlueStore cache to devote to metadata

type: `float`
runtime updatable: `true`
default: `0.45`
see also: `bluestore_cache_size`

Ratio of BlueStore cache to devote to key/value database (RocksDB)

type: `float`
runtime updatable: `true`
default: `0.45`
see also: `bluestore_cache_size`

CHECKSUMS

BlueStore checksums all metadata and all data written to disk. Metadata checksumming is handled by RocksDB and uses the *crc32c* algorithm. By contrast, data checksumming is handled by BlueStore and can use either *crc32c*, *xxhash32*, or *xxhash64*. Nonetheless, *crc32c* is the default checksum algorithm and it is suitable for most purposes.

Full data checksumming increases the amount of metadata that BlueStore must store and manage. Whenever possible (for example, when clients hint that data is written and read sequentially), BlueStore will checksum larger blocks. In many cases, however, it must store a checksum value (usually 4 bytes) for every 4 KB block of data.

It is possible to obtain a smaller checksum value by truncating the checksum to one or two bytes and reducing the metadata overhead. A drawback of this approach is that it increases the probability of a random error going undetected: about one in four billion given a 32-bit (4 byte) checksum, 1 in 65,536 given a 16-bit (2 byte) checksum, and 1 in 256 given an 8-bit (1 byte) checksum. To use the smaller checksum values, select *crc32c_16* or *crc32c_8* as the checksum algorithm.

The *checksum algorithm* can be specified either via a per-pool `csum_type` configuration option or via the global configuration option. For example:

```
$ ceph osd pool set <pool-name> csum_type <algorithm>
```

`bluestore_csum_type`

type: `str`
runtime updatable: `true`
default: `crc32c`
valid choices:

- `none`
- `crc32c`
- `crc32c_16`
- `crc32c_8`
- `xxhash32`
- `xxhash64`

INLINE COMPRESSION

BlueStore supports inline compression using *snappy*, *zlib*, *lz4*, or *zstd*.

Whether data in BlueStore is compressed is determined by two factors: (1) the *compression mode* and (2) any client hints associated with a write operation. The compression modes are as follows:

- **none**: Never compress data.
- **passive**: Do not compress data unless the write operation has a *compressible* hint set.
- **aggressive**: Do compress data unless the write operation has an *incompressible* hint set.
- **force**: Try to compress data no matter what.

For more information about the *compressible* and *incompressible* I/O hints, see `rados_set_alloc_hint()`.

Note that data in BlueStore will be compressed only if the data chunk will be sufficiently reduced in size (as determined by the `bluestore compression required ratio` setting). No matter which compression modes have been used, if the data chunk is too big, then it will be discarded and the original (uncompressed) data will be stored instead. For example, if `bluestore compression required ratio` is set to `.7`, then data compression will take place only if the size of the compressed data is no more than 70% of the size of the original data.

The *compression mode*, *compression algorithm*, *compression required ratio*, *min blob size*, and *max blob size* settings can be specified either via a per-pool property or via a global config option. To specify pool properties, run the following commands:

```
$ ceph osd pool set <pool-name> compression_mode <mode>
$ ceph osd pool set <pool-name> compression_required_ratio <ratio>
$ ceph osd pool set <pool-name> compression_min_blob_size <size>
$ ceph osd pool set <pool-name> compression_max_blob_size <size>
```

bluestore_compression_algorithm

The default compressor to use (if any) if the per-pool property `compression_algorithm` is not set. Note that `zstd` is *not* recommended for BlueStore due to high CPU overhead when compressing small amounts of data.

type: `str`
runtime updatable: `true`
default: `snappy`
valid choices:

- <empty string>
- `snappy`
- `zlib`
- `zstd`
- `lz4`

bluestore_compression_mode

The default policy for using compression if the per-pool property `compression_mode` is not set. `none` means never use compression. `passive` means use compression when `clients hint` that data is compressible. `aggressive` means use compression unless clients hint that data is not compressible. `force` means use compression under all circumstances even if the clients hint that the data is not compressible.

type: `str`
runtime updatable: `true`
default: `none`
valid choices:

- `none`
- `passive`
- `aggressive`
- `force`

bluestore_compression_required_ratio

be at least this small in order to store the compressed version.

- **type:** `float`
- **runtime updatable:** `true`
- **default:** `0.875`

`bluestore_compression_min_blob_size`

Chunks smaller than this are never compressed. The per-pool property `compression_min_blob_size` overrides this setting.

- **type:** `size`
- **runtime updatable:** `true`
- **default:** `0B`

`bluestore_compression_min_blob_size_hdd`

Default value of `bluestore compression min blob size` for rotational media.

- **type:** `size`
- **runtime updatable:** `true`
- **default:** `64Ki`
- **see also:** `bluestore_compression_min_blob_size`

`bluestore_compression_min_blob_size_ssd`

Default value of `bluestore compression min blob size` for non- rotational (solid state) media.

- **type:** `size`
- **runtime updatable:** `true`
- **default:** `64Ki`
- **see also:** `bluestore_compression_min_blob_size`

`bluestore_compression_max_blob_size`

Chunks larger than this value are broken into smaller blobs of at most `bluestore_compression_max_blob_size` bytes before being compressed. The per-pool property `compression_max_blob_size` overrides this setting.

- **type:** `size`
- **runtime updatable:** `true`
- **default:** `0B`

Default value of `bluestore compression max blob size` for rotational media.

type: `size`
runtime updatable: `true`
default: `64Ki`
see also: `bluestore_compression_max_blob_size`

`bluestore_compression_max_blob_size_ssd`

Default value of `bluestore compression max blob size` for non- rotational (SSD, NVMe) media.

type: `size`
runtime updatable: `true`
default: `64Ki`
see also: `bluestore_compression_max_blob_size`

ROCKSDB SHARDING

BlueStore maintains several types of internal key-value data, all of which are stored in RocksDB. Each data type in BlueStore is assigned a unique prefix. Prior to the Pacific release, all key-value data was stored in a single RocksDB column family: 'default'. In Pacific and later releases, however, BlueStore can divide key-value data into several RocksDB column families. BlueStore achieves better caching and more precise compaction when keys are similar: specifically, when keys have similar access frequency, similar modification frequency, and a similar lifetime. Under such conditions, performance is improved and less disk space is required during compaction (because each column family is smaller and is able to compact independently of the others).

OSDs deployed in Pacific or later releases use RocksDB sharding by default. However, if Ceph has been upgraded to Pacific or a later version from a previous version, sharding is disabled on any OSDs that were created before Pacific.

To enable sharding and apply the Pacific defaults to a specific OSD, stop the OSD and run the following command:

```
# ceph-bluestore-tool \
 --path <data path> \
 --sharding="m(3) p(3,0-12) O(3,0-13)=block_cache={type=binned_lru} L P"
 reshard
```

Enables sharding of BlueStore's RocksDB. When `true`, `bluestore_rocksdb_cfs` is used. Only applied when OSD is doing `--mkfs`.

type: `bool`
runtime updatable: `true`
default: `true`

bluestore_rocksdb_cfs

Definition of BlueStore's RocksDB sharding. The optimal value depends on multiple factors, and modification is inadvisable. This setting is used only when OSD is doing `--mkfs`. Next runs of OSD retrieve sharding from disk.

type: `str`
runtime updatable: `false`
default: `m(3) p(3,0-12) O(3,0-13)=block_cache={type=binned_lru} L=min_write_buffer_number_to_merge=32 P=min_write_buffer_number_to_merge=32`

THROTTLING

bluestore_throttle_bytes

Maximum bytes in flight before we throttle IO submission

type: `size`
runtime updatable: `true`
default: `64Mi`

bluestore_throttle_deferred_bytes

Maximum bytes for deferred writes before we throttle IO submission

type: `size`
runtime updatable: `true`
default: `128Mi`

bluestore_throttle_cost_per_io

type: `size`
runtime updatable: `true`
default: `0B`

`bluestore_throttle_cost_per_io_hdd`

Default bluestore_throttle_cost_per_io for rotational media (HDDs)

type: `uint`
runtime updatable: `true`
default: `670000`
see also: `bluestore_throttle_cost_per_io`

`bluestore_throttle_cost_per_io_ssd`

Default bluestore_throttle_cost_per_io for non-rotation (SSD) media

type: `uint`
runtime updatable: `true`
default: `4000`
see also: `bluestore_throttle_cost_per_io`

SPDK USAGE

To use the SPDK driver for NVMe devices, you must first prepare your system. See SPDK document.

SPDK offers a script that will configure the device automatically. Run this script with root permissions:

```
$ sudo src/spdk/scripts/setup.sh
```

You will need to specify the subject NVMe device's device selector with the "spdk:" prefix for `bluestore_block_path`.

In the following example, you first find the device selector of an Intel NVMe SSD by running the following command:

The form of the device selector is either `DDDD:BB:DD.FF` or `DDDD.BB.DD.FF`.

Next, supposing that `0000:01:00.0` is the device selector found in the output of the `lspci` command, you can specify the device selector by running the following command:

```
bluestore_block_path = "spdk:trtype:PCIe traddr:0000:01:00.0"
```

You may also specify a remote NVMeoF target over the TCP transport, as in the following example:

```
bluestore_block_path = "spdk:trtype:TCP traddr:10.67.110.197 trsvcid:4420 subnqn:nqn.2019-02.io.spdk:cnode1"
```

To run multiple SPDK instances per node, you must make sure each instance uses its own DPDK memory by specifying for each instance the amount of DPDK memory (in MB) that the instance will use.

In most cases, a single device can be used for data, DB, and WAL. We describe this strategy as *colocating* these components. Be sure to enter the below settings to ensure that all I/Os are issued through SPDK:

```
bluestore_block_db_path = ""
bluestore_block_db_size = 0
bluestore_block_wal_path = ""
bluestore_block_wal_size = 0
```

If these settings are not entered, then the current implementation will populate the SPDK map files with kernel file system symbols and will use the kernel driver to issue DB/WAL I/Os.

MINIMUM ALLOCATION SIZE

There is a configured minimum amount of storage that BlueStore allocates on a
storage device. In practice, this is the least amount of capacity that even a tiny R
can consume on each OSD's primary device. The configuration option in question--

`bluestore_min_alloc_size_hdd` or `bluestore_min_alloc_size_ssd`, depending on the OSD's `rotational` attribute. Thus if an OSD is created on an HDD, BlueStore is initialized with the current value of `bluestore_min_alloc_size_hdd`; but with SSD OSDs (including NVMe devices), BlueStore is initialized with the current value of `bluestore_min_alloc_size_ssd`.

In Mimic and earlier releases, the default values were 64KB for rotational media (HDD) and 16KB for non-rotational media (SSD). The Octopus release changed the default value for non-rotational media (SSD) to 4KB, and the Pacific release changed the default value for rotational media (HDD) to 4KB.

These changes were driven by space amplification that was experienced by Ceph RADOS Gateway (RGW) deployments that hosted large numbers of small files (S3/Swift objects).

For example, when an RGW client stores a 1 KB S3 object, that object is written to a single RADOS object. In accordance with the default `min_alloc_size` value, 4 KB of underlying drive space is allocated. This means that roughly 3 KB (that is, 4 KB minus 1 KB) is allocated but never used: this corresponds to 300% overhead or 25% efficiency. Similarly, a 5 KB user object will be stored as two RADOS objects, a 4 KB RADOS object and a 1 KB RADOS object, with the result that 4KB of device capacity is stranded. In this case, however, the overhead percentage is much smaller. Think of this in terms of the remainder from a modulus operation. The overhead *percentage* thus decreases rapidly as object size increases.

There is an additional subtlety that is easily missed: the amplification phenomenon just described takes place for *each* replica. For example, when using the default of three copies of data (3R), a 1 KB S3 object actually strands roughly 9 KB of storage device capacity. If erasure coding (EC) is used instead of replication, the amplification might be even higher: for a `k=4, m=2` pool, our 1 KB S3 object allocates 24 KB (that is, 4 KB multiplied by 6) of device capacity.

When an RGW bucket pool contains many relatively large user objects, the effect of this phenomenon is often negligible. However, with deployments that can expect a significant fraction of relatively small user objects, the effect should be taken into consideration.

The 4KB default value aligns well with conventional HDD and SSD devices. However, certain novel coarse-IU (Indirection Unit) QLC SSDs perform and wear best when `bluestore_min_alloc_size_ssd` is specified at OSD creation to match the device's IU: this might be 8KB, 16KB, or even 64KB. These novel storage drives can achieve read performance that is competitive with that of conventional TLC SSDs and write performance that is faster than that of HDDs, with higher density and lower cost than TLC SSDs.

default value only to appropriate devices, and not to conventional HDD and SSD devices. Error can be avoided through careful ordering of OSD creation, with custom OSD device classes, and especially by the use of central configuration *masks*.

In Quincy and later releases, you can use the `bluestore_use_optimal_io_size_for_min_alloc_size` option to allow automatic discovery of the correct value as each OSD is created. Note that the use of `bcache`, `OpenCAS`, `dmcrypt`, `ATA over Ethernet`, *iSCSI*, or other device-layering and abstraction technologies might confound the determination of correct values. Moreover, OSDs deployed on top of VMware storage have sometimes been found to report a `rotational` attribute that does not match the underlying hardware.

We suggest inspecting such OSDs at startup via logs and admin sockets in order to ensure that their behavior is correct. Be aware that this kind of inspection might not work as expected with older kernels. To check for this issue, examine the presence and value of `/sys/block/<drive>/queue/optimal_io_size`.

Note

When running Reef or a later Ceph release, the `min_alloc_size` baked into each OSD is conveniently reported by `ceph osd metadata`.

To inspect a specific OSD, run the following command:

```
# ceph osd metadata osd.1701 | egrep rotational\|alloc
```

This space amplification might manifest as an unusually high ratio of raw to stored data as reported by `ceph df`. There might also be `%USE` / `VAR` values reported by `ceph osd df` that are unusually high in comparison to other, ostensibly identical, OSDs. Finally, there might be unexpected balancer behavior in pools that use OSDs that have mismatched `min_alloc_size` values.

This BlueStore attribute takes effect *only* at OSD creation; if the attribute is changed later, a specific OSD's behavior will not change unless and until the OSD is destroyed and redeployed with the appropriate option value(s). Upgrading to a later Ceph release will *not* change the value used by OSDs that were deployed under older releases or with other settings.

bluestore_min_alloc_size

copy-on-write operation is triggered (e.g., when writing to something that was recently snapshotted). Similarly, less data is journaled before performing an overwrite (writes smaller than min_alloc_size must first pass through the BlueStore WAL). Larger values of min_alloc_size reduce the amount of metadata required to describe the on-disk layout and reduce overall fragmentation. Setting to 0 directs that the effective value is taken from bluestore_min_alloc_size_hdd or bluestore_min_alloc_size_ssd according to the kernel's rotational attribute for the underlying device. Note that this is baked into each OSD at creation. An OSD must be rebuilt to use a different value.

type: `uint`
runtime updatable: `false`
default: `0`

`bluestore_min_alloc_size_hdd`

Default min_alloc_size value for rotational media

type: `size`
runtime updatable: `false`
default: `4Ki`
see also: `bluestore_min_alloc_size`

`bluestore_min_alloc_size_ssd`

Default min_alloc_size value for non-rotational (solid state) media

type: `size`
runtime updatable: `false`
default: `4Ki`
see also: `bluestore_min_alloc_size`

`bluestore_use_optimal_io_size_for_min_alloc_size`

Discover media optimal IO size and use for min_alloc_size. This is useful when OSDs are created on coarse-IU QLC SSDs or other novel types of underlyinng block device. It is a no-op for conventional media.

type: `bool`
runtime updatable: `false`
default: `false`
see also: `bluestore_min_alloc_size`

Ceph Foundation. If you would like to support this and our other efforts, please consider joining now.