BLUESTORE CONFIG REFERENCE

DEVICES

BlueStore manages either one, two, or (in certain cases) three storage devices.

In the simplest case, BlueStore consumes a single (primary) storage device. The storage device is normally partitioned into two parts:

1. A small partition is formatted with XFS and contains basic metadata for the OSD. This *data directory* includes information about the OSD (its identifier, which cluster it belongs to, and its private keyring).
2. The rest of the device is normally a large partition occupying the rest of the device that is managed directly by BlueStore contains all of the actual data. This *primary device* is normally identifed by a `block` symlink in data directory.

It is also possible to deploy BlueStore across two additional devices:

- A *WAL device* can be used for BlueStore's internal journal or write-ahead log. It is identified by the `block.wal` symlink in the data directory. It is only useful to use a WAL device if the device is faster than the primary device (e.g., when it is on an SSD and the primary device is an HDD).
- A *DB device* can be used for storing BlueStore's internal metadata. BlueStore (or rather, the embedded RocksDB) will put as much metadata as it can on the DB device to improve performance. If the DB device fills up, metadata will spill back onto the primary device (where it would have been otherwise). Again, it is only helpful to provision a DB device if it is faster than the primary device.

If there is only a small amount of fast storage available (e.g., less than a gigabyte), we recommend using it as a WAL device. If there is more, provisioning a DB device makes more sense. The BlueStore journal will always be placed on the fastest device available, so using a DB device will provide the same benefit that the WAL device would while *also* allowing additional metadata to be stored there (if it will fix).

A single-device BlueStore OSD can be provisioned with:

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

To specify a WAL device and/or DB device,

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

Note: –data can be a Logical Volume using the vg/lv notation. Other devices can be existing logical volumes or GPT partitions

CACHE SIZE

The amount of memory consumed by each OSD for BlueStore's cache is determined by the `bluestore_cache_size` configuration option. If that config option is not set (i.e., remains at 0), there is a different default value that is used depending on whether an HDD or SSD is used for the primary device (set by the `bluestore_cache_size_ssd` and `bluestore_cache_size_hdd` config options).

BlueStore and the rest of the Ceph OSD does the best it can currently to stick to the budgeted memory. Note that on top of the configured cache size, there is also memory consumed by the OSD itself, and generally some overhead due to memory fragmentation and other allocator overhead.

The configured cache memory budget can be used in a few different ways:

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

Cache memory usage is governed by the following options: `bluestore_cache_meta_ratio`, `bluestore_cache_kv_ratio`, and `bluestore_cache_kv_max`. The fraction of the cache devoted to data is 1.0 minus the meta and kv ratios. The memory devoted to kv metadata (the RocksDB cache) is capped by `bluestore_cache_kv_max` since our testing indicates there are

diminishing returns beyond a certain point.

`bluestore_cache_size`

Description: 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: Unsigned Integer
Required: Yes
Default: `0`

`bluestore_cache_size_hdd`

Description: The default amount of memory BlueStore will use for its cache when backed by an HDD.
Type: Unsigned Integer
Required: Yes
Default: `1 * 1024 * 1024 * 1024` (1 GB)

`bluestore_cache_size_ssd`

Description: The default amount of memory BlueStore will use for its cache when backed by an SSD.
Type: Unsigned Integer
Required: Yes
Default: `3 * 1024 * 1024 * 1024` (3 GB)

`bluestore_cache_meta_ratio`

Description: The ratio of cache devoted to metadata.
Type: Floating point
Required: Yes
Default: `.01`

`bluestore_cache_kv_ratio`

Description: The ratio of cache devoted to key/value data (rocksdb).
Type: Floating point
Required: Yes
Default: `.99`

`bluestore_cache_kv_max`

Description: The maximum amount of cache devoted to key/value data (rocksdb).
Type: Unsigned Integer
Required: Yes
Default: `512 * 1024*1024` (512 MB)

CHECKSUMS

BlueStore checksums all metadata and data written to disk. Metadata checksumming is handled by RocksDB and uses *crc32c*. Data checksumming is done by BlueStore and can make use of *crc32c*, *xxhash32*, or *xxhash64*. The default is *crc32c* and should be suitable for most purposes.

Full data checksumming does increase the amount of metadata that BlueStore must store and manage. When possible, e.g., when clients hint that data is written and read sequentially, BlueStore will checksum larger blocks, but in many cases it must store a checksum value (usually 4 bytes) for every 4 kilobyte block of data.

It is possible to use a smaller checksum value by truncating the checksum to two or one byte, reducing the metadata overhead. The trade-off is that the probability that a random error will not be detected is higher with a smaller checksum, going from about one in four billion with a 32-bit (4 byte) checksum to one in 65,536 for a 16-bit (2 byte) checksum or one in 256 for an 8-bit (1 byte) checksum. The smaller checksum values can be used by selecting *crc32c_16* or *crc32c_8* as the checksum algorithm.

The *checksum algorithm* can be set either via a per-pool `csum_type` property or the global config option. For example,

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

`bluestore_csum_type`

Description: The default checksum algorithm to use.
Type: String
Required: Yes
Valid Settings: `none`, `crc32c`, `crc32c_16`, `crc32c_8`, `xxhash32`, `xxhash64`
Default: `crc32c`

INLINE COMPRESSION

BlueStore supports inline compression using *snappy*, *zlib*, or *lz4*. Please note that the *lz4* compression plugin is not distributed in the official release.

Whether data in BlueStore is compressed is determined by a combination of the *compression mode* and any hints associated with a write operation. The modes are:

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

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

Note that regardless of the mode, if the size of the data chunk is not reduced sufficiently it will not be used and the original (uncompressed) data will be stored. For example, if the `bluestore compression required ratio` is set to `.7` then the compressed data must be 70% of the size of the original (or smaller).

The *compression mode*, *compression algorithm*, *compression required ratio*, *min blob size*, and *max blob size* can be set either via a per-pool property or a global config option. Pool properties can be set with:

```
ceph osd pool set <pool-name> compression_algorithm <algorithm>
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`

Description: 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: String
Required: No
Valid Settings: `lz4`, `snappy`, `zlib`, `zstd`
Default: `snappy`

`bluestore compression mode`

Description: 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: String
Required: No
Valid Settings: `none`, `passive`, `aggressive`, `force`
Default: `none`

`bluestore compression required ratio`

Description: The ratio of the size of the data chunk after compression relative to the original size must be at least this small in order to store the compressed version.
Type: Floating point
Required: No

Default: .875

`bluestore compression min blob size`

Description: Chunks smaller than this are never compressed. The per-pool property `compression_min_blob_size` overrides this setting.
Type: Unsigned Integer
Required: No
Default: 0

`bluestore compression min blob size hdd`

Description: Default value of `bluestore compression min blob size` for rotational media.
Type: Unsigned Integer
Required: No
Default: 128K

`bluestore compression min blob size ssd`

Description: Default value of `bluestore compression min blob size` for non-rotational (solid state) media.
Type: Unsigned Integer
Required: No
Default: 8K

`bluestore compression max blob size`

Description: Chunks larger than this are broken into smaller blobs sizing `bluestore compression max blob size` before being compressed. The per-pool property `compression_max_blob_size` overrides this setting.
Type: Unsigned Integer
Required: No
Default: 0

`bluestore compression max blob size hdd`

Description: Default value of `bluestore compression max blob size` for rotational media.
Type: Unsigned Integer
Required: No
Default: 512K

`bluestore compression max blob size ssd`

Description: Default value of `bluestore compression max blob size` for non-rotational (solid state) media.
Type: Unsigned Integer
Required: No
Default: 64K

SPDK USAGE

If you want to use SPDK driver for NVME SSD, you need to specify NVMe serial number here with “spdk:” prefix for `bluestore_block_path`.

For example, users can find the serial number with:

```
$ lspci -vvv -d 8086:0953 | grep "Device Serial Number"
```

and then set:

```
bluestore block path = spdk:...
```

If you want to run multiple SPDK instances per node, you must specify the amount of dpdk memory size in MB each instance will use, to make sure each instance uses its own dpdk memory

In most cases, we only need one device to serve as data, db, db wal purposes. We need to make sure configurations below to make sure all IOs issued under SPDK.:

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

Otherwise, the current implementation will setup symbol file to kernel filesystem location and uses kernel driver to issue DB/WAL IO.