bcachefs

Kent Overstreet

Nov 02, 2022

OVERVIEW:

CHAPTER

ONE

INTRODUCTION

Bcachefs is a modern, general purpose, copy on write filesystem descended from bcache, a block layer cache.

The internal architecture is very different from most existing filesystems where the inode is central and many data structures hang off of the inode. Instead, bcachefs is architected more like a filesystem on top of a relational database, with tables for the different filesystem data types - extents, inodes, dirents, xattrs, et cetera.

bcachefs supports almost all of the same features as other modern COW filesystems, such as ZFS and btrfs, but in general with a cleaner, simpler, higher performance design.

CHAPTER

TWO

PERFORMANCE

The core of the architecture is a very high performance and very low latency b+ tree, which also is not a conventional b+ tree but more of hybrid, taking concepts from compacting data structures: btree nodes are very large, log structured, and compacted (resorted) as necessary in memory. This means our b+ trees are very shallow compared to other filesystems.

What this means for the end user is that since we require very few seeks or disk reads, filesystem latency is extremely good - especially cache cold filesystem latency, which does not show up in most benchmarks but has a huge impact on real world performance, as well as how fast the system "feels" in normal interactive usage. Latency has been a major focus throughout the codebase - notably, we have assertions that we never hold b+ tree locks while doing IO, and the btree transaction layer makes it easily to aggressively drop and retake locks as needed - one major goal of bcachefs is to be the first general purpose soft realtime filesystem.

Additionally, unlike other COW btrees, btree updates are journalled. This greatly improves our write efficiency on random update workloads, as it means btree writes are only done when we have a large block of updates, or when required by memory reclaim or journal reclaim.

CHAPTER

THREE

BUCKET BASED ALLOCATION

As mentioned bcachefs is descended from bcache, where the ability to efficiently invalidate cached data and reuse disk space was a core design requirement. To make this possible the allocator divides the disk up into buckets, typically 512k to 2M but possibly larger or smaller. Buckets and data pointers have generation numbers: we can reuse a bucket with cached data in it without finding and deleting all the data pointers by incrementing the generation number.

In keeping with the copy-on-write theme of avoiding update in place wherever possible, we never rewrite or overwrite data within a bucket - when we allocate a bucket, we write to it sequentially and then we don't write to it again until the bucket has been invalidated and the generation number incremented.

This means we require a copying garbage collector to deal with internal fragmentation, when patterns of random writes leave us with many buckets that are partially empty (because the data they contained was overwritten) - copy GC evacuates buckets that are mostly empty by writing the data they contain to new buckets. This also means that we need to reserve space on the device for the copy GC reserve when formatting - typically 8% or 12%.

There are some advantages to structuring the allocator this way, besides being able to support cached data:

- By maintaining multiple write points that are writing to different buckets, we're able to easily and naturally segregate unrelated IO from different processes, which helps greatly with fragmentation.
- The fast path of the allocator is essentially a simple bump allocator - the disk space allocation is extremely fast
- Fragmentation is generally a non issue unless copygc has to kick in, and it usually doesn't under typical usage patterns. The allocator and copygc are doing essentially the same things as the flash translation layer in SSDs, but within the filesystem we have much greater visibility into where writes are coming from and how to segregate them, as well as which data is actually live - performance is generally more predictable than with SSDs under similar usage patterns.
- The same algorithms will in the future be used for managing SMR hard drives directly, avoiding the translation layer in the hard drive - doing this work within the filesystem should give much better performance and much more predictable latency.

CHAPTER

FOUR

IO PATH OPTIONS

Most options that control the IO path can be set at either the filesystem level or on individual inodes (files and directories). When set on a directory via the `bcachefs attr` command, they will be automatically applied recursively.

4.1 Checksumming

bcachefs supports both metadata and data checksumming - crc32c by default, but stronger checksums are available as well. Enabling data checksumming incurs some performance overhead - besides the checksum calculation, writes have to be bounced for checksum stability (Linux generally cannot guarantee that the buffer being written is not modified in flight), but reads generally do not have to be bounced.

Checksum granularity in bcachefs is at the level of individual extents, which results in smaller metadata but means we have to read entire extents in order to verify the checksum. By default, checksummed and compressed extents are capped at 64k. For most applications and usage scenarios this is an ideal trade off, but small random `O_DIRECT` reads will incur significant overhead. In the future, checksum granularity will be a per-inode option.

4.2 Encryption

bcachefs supports authenticated (AEAD style) encryption - ChaCha20/Poly1305. When encryption is enabled, the poly1305 MAC replaces the normal data and metadata checksums. This style of encryption is superior to typical block layer or filesystem level encryption (usually AES-XTS), which only operates on blocks and doesn't have a way to store nonces or MACs. In contrast, we store a nonce and cryptographic MAC alongside data pointers - meaning we have a chain of trust up to the superblock (or journal, in the case of unclean shutdowns) and can definitely tell if metadata has been modified, dropped, or replaced with an earlier version - replay attacks are not possible.

Encryption can only be specified for the entire filesystem, not per file or directory - this is because metadata blocks do not belong to a particular file. All metadata except for the superblock is encrypted.

In the future we'll probably add AES-GCM for platforms that have hardware acceleration for AES, but in the meantime software implementations of ChaCha20 are also quite fast on most platforms.

`scrypt` is used for the key derivation function - for converting the user supplied passphrase to an encryption key.

To format a filesystem with encryption, use

```
bcachefs format --encrypted /dev/sda1
```

You will be prompted for a passphrase. Then, to use an encrypted filesystem use the command

```
bcachefs unlock /dev/sda1
```

You will be prompted for the passphrase and the encryption key will be added to your in-kernel keyring; mount, fsck and other commands will then work as usual.

The passphrase on an existing encrypted filesystem can be changed with the `bcachefs set-passphrase` command. To permanently unlock an encrypted filesystem, use the `bcachefs remove-passphrase` command - this can be useful when dumping filesystem metadata for debugging by the developers.

There is a `wide_macs` option which controls the size of the cryptographic MACs stored on disk. By default, only 80 bits are stored, which should be sufficient security for most applications. With the `wide_macs` option enabled we store the full 128 bit MAC, at the cost of making extents 8 bytes bigger.

4.3 Compression

bcachefs supports gzip, lz4 and zstd compression. As with data checksumming, we compress entire extents, not individual disk blocks - this gives us better compression ratios than other filesystems, at the cost of reduced small random read performance.

Data can also be compressed or recompressed with a different algorithm in the background by the rebalance thread, if the `background_compression` option is set.

CHAPTER

FIVE

MULTIPLE DEVICES

bcachefs is a multi-device filesystem. Devices need not be the same size: by default, the allocator will stripe across all available devices but biasing in favor of the devices with more free space, so that all devices in the filesystem fill up at the same rate. Devices need not have the same performance characteristics: we track device IO latency and direct reads to the device that is currently fastest.

5.1 Replication

bcachefs supports standard RAID1/10 style redundancy with the `data_replicas` and `metadata_replicas` options. Layout is not fixed as with RAID10: a given extent can be replicated across any set of devices; the `bcachefs fs usage` command shows how data is replicated within a filesystem.

5.2 Erasure coding

bcachefs also supports Reed-Solomon erasure coding - the same algorithm used by most RAID5/6 implementations) When enabled with the `ec` option, the desired redundancy is taken from the `data_replicas` option - erasure coding of metadata is not supported.

Erasure coding works significantly differently from both conventional RAID implementations and other filesystems with similar features. In conventional RAID, the "write hole" is a significant problem - doing a small write within a stripe requires the P and Q (recovery) blocks to be updated as well, and since those writes cannot be done atomically there is a window where the P and Q blocks are inconsistent - meaning that if the system crashes and recovers with a drive missing, reconstruct reads for unrelated data within that stripe will be corrupted.

ZFS avoids this by fragmenting individual writes so that every write becomes a new stripe - this works, but the fragmentation has a negative effect on performance: metadata becomes bigger, and both read and write requests are excessively fragmented. Btrfs's erasure coding implementation is more conventional, and still subject to the write hole problem.

bcachefs's erasure coding takes advantage of our copy on write nature - since updating stripes in place is a problem, we simply don't do that. And since excessively small stripes is a problem for fragmentation, we don't erasure code individual extents, we erasure code entire buckets - taking advantage of bucket based allocation and copying garbage collection.

When erasure coding is enabled, writes are initially replicated, but one of the replicas is allocated from a bucket that is queued up to be part of a new stripe. When we finish filling up the new stripe, we write out the P and Q buckets and then drop the extra replicas for all the data within that stripe - the effect is similar to full data journalling, and it means that after erasure coding is done the layout of our data on disk is ideal.

Since disks have write caches that are only flushed when we issue a cache flush command - which we only do on journal commit - if we can tweak the allocator so that the buckets used for the extra replicas are reused (and then overwritten

again) immediately, this full data journalling should have negligible overhead - this optimization is not implemented yet, however.

5.3 Device labels and targets

By default, writes are striped across all devices in a filesystem, but they may be directed to a specific device or set of devices with the various target options. The allocator only prefers to allocate from devices matching the specified target; if those devices are full, it will fall back to allocating from any device in the filesystem.

Target options may refer to a device directly, e.g. `foreground_target=/dev/sda1`, or they may refer to a device label. A device label is a path delimited by periods - e.g. ssd.ssd1 (and labels need not be unique). This gives us ways of referring to multiple devices in target options: If we specify ssd in a target option, that will refer to all devices with the label ssd or labels that start with ssd. (e.g. ssd.ssd1, ssd.ssd2).

Four target options exist. These options all may be set at the filesystem level (at format time, at mount time, or at runtime via sysfs), or on a particular file or directory:

`foreground_target`: normal foreground data writes, and metadata if `metadata_target` is not set

`metadata_target`: btree writes

`background_target`: If set, user data (not metadata) will be moved to this target in the background

`promote_target`: If set, a cached copy will be added to this target on read, if none exists

5.4 Caching

When an extent has multiple copies on different devices, some of those copies may be marked as cached. Buckets containing only cached data are discarded as needed by the allocator in LRU order.

When data is moved from one device to another according to the `background_target` option, the original copy is left in place but marked as cached. With the `promote_target` option, the original copy is left unchanged and the new copy on the `promote_target` device is marked as cached.

To do writeback caching, set `foreground_target` and `promote_target` to the cache device, and `background_target` to the backing device. To do writearound caching, set `foreground_target` to the backing device and `promote_target` to the cache device.

5.5 Durability

Some devices may be considered to be more reliable than others. For example, we might have a filesystem composed of a hardware RAID array and several NVME flash devices, to be used as cache. We can set replicas=2 so that losing any of the NVME flash devices will not cause us to lose data, and then additionally we can set durability=2 for the hardware RAID device to tell bcachefs that we don't need extra replicas for data on that device - data on that device will count as two replicas, not just one.

The durability option can also be used for writethrough caching: by setting durability=0 for a device, it can be used as a cache and only as a cache - bcachefs won't consider copies on that device to count towards the number of replicas we're supposed to keep.

CHAPTER

SIX

REFLINK

bcachefs supports reflink, similarly to other filesystems with the same feature. cp –reflink will create a copy that shares the underlying storage. Reading from that file will become slightly slower - the extent pointing to that data is moved to the reflink btree (with a refcount added) and in the extents btree we leave a key that points to the indirect extent in the reflink btree, meaning that we now have to do two btree lookups to read from that data instead of just one.

CHAPTER

SEVEN

INLINE DATA EXTENTS

bcachefs supports inline data extents, controlled by the `inline_data` option (on by default). When the end of a file is being written and is smaller than half of the filesystem blocksize, it will be written as an inline data extent. Inline data extents can also be reflinked (moved to the reflink btree with a refcount added): as a todo item we also intend to support compressed inline data extents.

CHAPTER

EIGHT

SUBVOLUMES AND SNAPSHOTS

bcachefs supports subvolumes and snapshots with a similar userspace interface as btrfs. A new subvolume may be created empty, or it may be created as a snapshot of another subvolume. Snapshots are writeable and may be snapshotted again, creating a tree of snapshots.

Snapshots are very cheap to create: they're not based on cloning of COW btrees as with btrfs, but instead are based on versioning of individual keys in the btrees. Many thousands or millions of snapshots can be created, with the only limitation being disk space.

The following subcommands exist for managing subvolumes and snapshots:

- `bcachefs subvolume create`: Create a new, empty subvolume
- `bcachefs subvolume destroy`: Delete an existing subvolume or snapshot
- `bcachefs subvolume snapshot`: Create a snapshot of an existing subvolume

A subvolume can also be deleting with a normal rmdir after deleting all the contents, as with `rm -rf`. Still to be implemented: read-only snapshots, recursive snapshot creation, and a method for recursively listing subvolumes.

CHAPTER

NINE

QUOTAS

bcachefs supports conventional user/group/project quotas. Quotas do not currently apply to snapshot subvolumes, because if a file changes ownership in the snapshot it would be ambiguous as to what quota data within that file should be charged to.

When a directory has a project ID set it is inherited automatically by descendants on creation and rename. When renaming a directory would cause the project ID to change we return -EXDEV so that the move is done file by file, so that the project ID is propagated correctly to descendants - thus, project quotas can be used as subdirectory quotas.

CHAPTER

TEN

FORMATTING

To format a new bcachefs filesystem use the subcommand `bcachefs format`, or `mkfs.bcachefs`. All persistent filesystem-wide options can be specified at format time. For an example of a multi device filesystem with compression, encryption, replication and writeback caching:

```
bcachefs format --compression=lz4                \
                --encrypted                      \
                --replicas=2                     \
                --label=ssd.ssd1 /dev/sda        \
                --label=ssd.ssd2 /dev/sdb        \
                --label=hdd.hdd1 /dev/sdc        \
                --label=hdd.hdd2 /dev/sdd        \
                --label=hdd.hdd3 /dev/sde        \
                --label=hdd.hdd4 /dev/sdf        \
                --foreground_target=ssd          \
                --promote_target=ssd             \
                --background_target=hdd
```

CHAPTER

ELEVEN

MOUNTING

To mount a multi device filesystem, there are two options. You can specify all component devices, separated by hyphens, e.g.

```
mount -t bcachefs /dev/sda:/dev/sdb:/dev/sdc /mnt
```

Or, use the mount.bcachefs tool to mount by filesystem UUID. Still todo: improve the mount.bcachefs tool to support mounting by filesystem label.

No special handling is needed for recovering from unclean shutdown. Journal replay happens automatically, and diagnostic messages in the dmesg log will indicate whether recovery was from clean or unclean shutdown.

The `-o degraded` option will allow a filesystem to be mounted without all the the devices, but will fail if data would be missing. The `-o very_degraded` can be used to attempt mounting when data would be missing.

Also relevant is the `-o nochanges` option. It disallows any and all writes to the underlying devices, pinning dirty data in memory as necessary if for example journal replay was necessary - think of it as a "super read-only" mode. It can be used for data recovery, and for testing version upgrades.

The `-o verbose` enables additional log output during the mount process.

CHAPTER

TWELVE

CHECKING FILESYSTEM INTEGRITY

It is possible to run fsck either in userspace with the `bcachefs fsck` subcommand (also available as `fsck.bcachefs`, or in the kernel while mounting by specifying the `-o fsck` mount option. In either case the exact same fsck implementation is being run, only the environment is different. Running fsck in the kernel at mount time has the advantage of somewhat better performance, while running in userspace has the ability to be stopped with ctrl-c and can prompt the user for fixing errors. To fix errors while running fsck in the kernel, use the `-o fix_errors` option.

The `-n` option passed to fsck implies the `-o nochanges` option; `bcachefs fsck -ny` can be used to test filesystem repair in dry-run mode.

CHAPTER

THIRTEEN

STATUS OF DATA

The `bcachefs fs usage` may be used to display filesystem usage broken out in various ways. Data usage is broken out by type: superblock, journal, btree, data, cached data, and parity, and by which sets of devices extents are replicated across. We also give per-device usage which includes fragmentation due to partially used buckets.

CHAPTER

FOURTEEN

JOURNAL

The journal has a number of tunables that affect filesystem performance. Journal commits are fairly expensive operations as they require issuing FLUSH and FUA operations to the underlying devices. By default, we issue a journal flush one second after a filesystem update has been done; this is controlled with the `journal_flush_delay` option, which takes a parameter in milliseconds.

Filesystem sync and fsync operations issue journal flushes; this can be disabled with the `journal_flush_disabled` option - the `journal_flush_delay` option will still apply, and in the event of a system crash we will never lose more than (by default) one second of work. This option may be useful on a personal workstation or laptop, and perhaps less appropriate on a server.

The journal reclaim thread runs in the background, kicking off btree node writes and btree key cache flushes to free up space in the journal. Even in the absence of space pressure it will run slowly in the background: this is controlled by the `journal_reclaim_delay` parameter, with a default of 100 milliseconds.

The journal should be sized sufficiently that bursts of activity do not fill up the journal too quickly; also, a larger journal mean that we can queue up larger btree writes. The `bcachefs device resize-journal` can be used for resizing the journal on disk on a particular device - it can be used on a mounted or unmounted filesystem.

In the future, we should implement a method to see how much space is currently utilized in the journal.

CHAPTER

FIFTEEN

DEVICE MANAGEMENT

15.1 Filesystem resize

A filesystem can be resized on a particular device with the `bcachefs device resize` subcommand. Currently only growing is supported, not shrinking.

15.2 Device add/removal

The following subcommands exist for adding and removing devices from a mounted filesystem:

- `bcachefs device add`: Formats and adds a new device to an existing filesystem.
- `bcachefs device remove`: Permenantly removes a device from an existing filesystem.
- `bcachefs device online`: Connects a device to a running filesystem that was mounted without it (i.e. in degraded mode)
- `bcachefs device offline`: Disconnects a device from a mounted filesystem without removing it.
- `bcachefs device evacuate`: Migrates data off of a particular device to prepare for removal, setting it read-only if necessary.
- `bcachefs device set-state`: Changes the state of a member device: one of rw (readwrite), ro (readonly), failed, or spare.

  A failed device is considered to have 0 durability, and replicas on that device won't be counted towards the number of replicas an extent should have by rereplicate - however, bcachefs will still attempt to read from devices marked as failed.

The `bcachefs device remove`, `bcachefs device offline` and `bcachefs device set-state` commands take force options for when they would leave the filesystem degraded or with data missing. Todo: regularize and improve those options.

CHAPTER

SIXTEEN

DATA MANAGEMENT

16.1 Data rereplicate

The `bcachefs data rereplicate` command may be used to scan for extents that have insufficient replicas and write additional replicas, e.g. after a device has been removed from a filesystem or after replication has been enabled or increased.

16.2 Rebalance

To be implemented: a command for moving data between devices to equalize usage on each device. Not normally required because the allocator attempts to equalize usage across devices as it stripes, but can be necessary in certain scenarios - i.e. when a two-device filesystem with replication enabled that is very full has a third device added.

16.3 Scrub

To be implemented: a command for reading all data within a filesystem and ensuring that checksums are valid, fixing bitrot when a valid copy can be found.

CHAPTER

SEVENTEEN

OPTIONS

Most bcachefs options can be set filesystem wide, and a significant subset can also be set on inodes (files and directories), overriding the global defaults. Filesystem wide options may be set when formatting, when mounting, or at runtime via `/sys/fs/bcachefs/<uuid>/options/`. When set at runtime via sysfs the persistent options in the superblock are updated as well; when options are passed as mount parameters the persistent options are unmodified.

17.1 File and directory options

<say something here about how attrs must be set via bcachefs attr command>

Options set on inodes (files and directories) are automatically inherited by their descendants, and inodes also record whether a given option was explicitly set or inherited from their parent. When renaming a directory would cause inherited attributes to change we fail the rename with -EXDEV, causing userspace to do the rename file by file so that inherited attributes stay consistent.

Inode options are available as extended attributes. The options that have been explicitly set are available under the `bcachefs` namespace, and the effective options (explicitly set and inherited options) are available under the `bcachefs_effective` namespace. Examples of listing options with the getfattr command:

```
$ getfattr -d -m '^bcachefs\.' filename
$ getfattr -d -m '^bcachefs_effective\.' filename
```

Options may be set via the extended attribute interface, but it is preferable to use the `bcachefs setattr` command as it will correctly propagate options recursively.

17.2 Full option list

`block_size` **format**

Filesystem block size (default 4k)

`btree_node_size` **format**
Btree node size, default 256k
`errors` **format,mount,rutime**
Action to take on filesystem error
`metadata_replicas` **format,mount,runtime**

Number of replicas for metadata (journal and btree)
`data_replicas` **format,mount,runtime,inode**
Number of replicas for user data
`replicas` **format**
Alias for both metadata_replicas and data_replicas
`metadata_checksum` **format,mount,runtime**
Checksum type for metadata writes
`data_checksum` **format,mount,runtime,inode**
Checksum type for data writes
`compression` **format,mount,runtime,inode**
Compression type
`background_compression` **format,mount,runtime,inode**
Background compression type
`str_hash` **format,mount,runtime,inode**
Hash function for string hash tables (directories and xattrs)
`metadata_target` **format,mount,runtime,inode**
Preferred target for metadata writes
`foreground_target` **format,mount,runtime,inode**
Preferred target for foreground writes
`background_target` **format,mount,runtime,inode**
Target for data to be moved to in the background
`promote_target` **format,mount,runtime,inode**
Target for data to be copied to on read
`erasure_code` **format,mount,runtime,inode**
Enable erasure coding
`inodes_32bit` **format,mount,runtime**
Restrict new inode numbers to 32 bits
`shard_inode_numbers` **format,mount,runtime**
Use CPU id for high bits of new inode numbers.
`wide_macs` **format,mount,runtime**
Store full 128 bit cryptographic MACs (default 80)
`inline_data` **format,mount,runtime**
Enable inline data extents (default on)
`journal_flush_delay` **format,mount,runtime**
Delay in milliseconds before automatic journal commit (default 1000)
`journal_flush_disabled`**format,mount,runtime**

Disables journal flush on sync/fsync. `journal_flush_delay` remains in effect, thus with the default setting not more than 1 second of work will be lost.

`journal_reclaim_delay`**format,mount,runtime**
Delay in milliseconds before automatic journal reclaim
`acl` **format,mount**
Enable POSIX ACLs
`usrquota` **format,mount**

Enable user quotas
`grpquota` **format,mount**
Enable group quotas
`prjquota` **format,mount**
Enable project quotas
`degraded` **mount**
Allow mounting with data degraded
`very_degraded` **mount**
Allow mounting with data missing
`verbose` **mount**
Extra debugging info during mount/recovery
`fsck` **mount**
Run fsck during mount
`fix_errors` **mount**
Fix errors without asking during fsck
`ratelimit_errors` **mount**
Ratelimit error messages during fsck
`read_only` **mount**
Mount in read only mode
`nochanges` **mount**
Issue no writes, even for journal replay
`norecovery` **mount**
Don't replay the journal (not recommended)
`noexcl` **mount**
Don't open devices in exclusive mode
`version_upgrade` **mount**
Upgrade on disk format to latest version
`discard` **device**
Enable discard/TRIM support

17.3 Error actions

The `errors` option is used for inconsistencies that indicate some sort of a bug. Valid error actions are:

`continue`
: Log the error but continue normal operation

`ro`
: Emergency read only, immediately halting any changes to the filesystem on disk

`panic`
: Immediately halt the entire machine, printing a backtrace on the system console

17.4 Checksum types

Valid checksum types are:

`none` `crc32c`

(default)

`crc64`

17.5 Compression types

Valid compression types are:

none
(default)

`lz4` `gzip` `zstd`

17.6 String hash types

Valid hash types for string hash tables are:

`crc32c` `crc64` `siphash`

(default)

CHAPTER

EIGHTEEN

DEBUGGING TOOLS

18.1 Sysfs interface

Mounted filesystems are available in sysfs at `/sys/fs/bcachefs/<uuid>/` with various options, performance counters and internal debugging aids.

18.1.1 Options

Filesystem options may be viewed and changed via
`/sys/fs/bcachefs/<uuid>/options/`, and settings changed via sysfs will be persistently changed in the superblock as well.

18.1.2 Time stats

bcachefs tracks the latency and frequency of various operations and events, with quantiles for latency/duration in the `/sys/fs/bcachefs/<uuid>/time_stats/` directory.

`blocked_allocate`
Tracks when allocating a bucket must wait because none are immediately available, meaning the copygc thread is not keeping up with evacuating mostly empty buckets or the allocator thread is not keeping up with invalidating and discarding buckets.

`blocked_allocate_open_bucket`
Tracks when allocating a bucket must wait because all of our handles for pinning open buckets are in use (we statically allocate 1024).

`blocked_journal`
Tracks when getting a journal reservation must wait, either because journal reclaim isn't keeping up with reclaiming space in the journal, or because journal writes are taking too long to complete and we already have too many in flight.

`btree_gc`
Tracks when the btree_gc code must walk the btree at runtime - for recalculating the oldest outstanding generation number of every bucket in the btree.

`btree_lock_contended_read`

`btree_lock_contended_intent`

`btree_lock_contended_write`
Track when taking a read, intent or write lock on a btree node must block.

`btree_node_mem_alloc`
Tracks the total time to allocate memory in the btree node cache for a new btree node.

`btree_node_split`
Tracks btree node splits - when a btree node becomes full and is split into two new nodes

`btree_node_compact`
Tracks btree node compactions - when a btree node becomes full and needs to be compacted on disk.

`btree_node_merge`
Tracks when two adjacent btree nodes are merged.

`btree_node_sort`
Tracks sorting and resorting entire btree nodes in memory, either after reading them in from disk or for compacting prior to creating a new sorted array of keys.

`btree_node_read`
Tracks reading in btree nodes from disk.

`btree_interior_update_foreground`
Tracks foreground time for btree updates that change btree topology - i.e. btree node splits, compactions and merges; the duration measured roughly corresponds to lock held time.

`btree_interior_update_total`
Tracks time to completion for topology changing btree updates; first they have a foreground part that updates btree nodes in memory, then after the new nodes are written there is a transaction phase that records an update to an interior node or a new btree root as well as changes to the alloc btree.

`data_read`
Tracks the core read path - looking up a request in the extents (and possibly also reflink) btree, allocating bounce buffers if necessary, issuing reads, checksumming, decompressing, decrypting, and delivering completions.

`data_write`
Tracks the core write path - allocating space on disk for a new write, allocating bounce buffers if necessary, compressing, encrypting, checksumming, issuing writes, and updating the extents btree to point to the new data.

`data_promote`
Tracks promote operations, which happen when a read operation writes an additional cached copy of an extent to `promote_target`. This is done asynchronously from the original read.

`journal_flush_write`
Tracks writing of flush journal entries to disk, which first issue cache flush operations to the underlying devices then issue the journal writes as FUA writes. Time is tracked starting from after all journal reservations have released their references or the completion of the previous journal write.

`journal_noflush_write`
Tracks writing of non-flush journal entries to disk, which do not issue cache flushes or FUA writes.

`journal_flush_seq`
Tracks time to flush a journal sequence number to disk by filesystem sync and fsync operations, as well as the allocator prior to reusing buckets when none that do not need flushing are available.

18.1.3 Internals

`btree_cache`
Shows information on the btree node cache: number of cached nodes, number of dirty nodes, and whether the cannibalize lock (for reclaiming cached nodes to allocate new nodes) is held.

`dirty_btree_nodes`
Prints information related to the interior btree node update machinery, which is responsible for ensuring dependent btree node writes are ordered correctly.

For each dirty btree node, prints:

- Whether the `need_write` flag is set
- The level of the btree node
- The number of sectors written
- Whether writing this node is blocked, waiting for other nodes to be written
- Whether it is waiting on a btree_update to complete and make it reachable on-disk

`btree_key_cache`
Prints infromation on the btree key cache: number of freed keys (which must wait for a sRCU barrier to complete before being freed), number of cached keys, and number of dirty keys.

`btree_transactions`
Lists each running btree transactions that has locks held, listing which nodes they have locked and what type of lock, what node (if any) the process is blocked attempting to lock, and where the btree transaction was invoked from.

`btree_updates`
Lists outstanding interior btree updates: the mode (nothing updated yet, or updated a btree node, or wrote a new btree root, or was reparented by another btree update), whether its new btree nodes have finished writing, its embedded closure's refcount (while nonzero, the btree update is still waiting), and the pinned journal sequence number.

`journal_debug`
Prints a variety of internal journal state.

`journal_pins` Lists items pinning journal entries, preventing them from being reclaimed.

`new_stripes`
Lists new erasure-coded stripes being created.

`stripes_heap`
Lists erasure-coded stripes that are available to be reused.

`open_buckets`
Lists buckets currently being written to, along with data type and refcount.

`io_timers_read`

`io_timers_write`
Lists outstanding IO timers - timers that wait on total reads or writes to the filesystem.

`trigger_journal_flush`
Echoing to this file triggers a journal commit.

`trigger_gc`
Echoing to this file causes the GC code to recalculate each bucket's oldest_gen field.

`prune_cache`
Echoing to this file prunes the btree node cache.

`read_realloc_races`
This counts events where the read path reads an extent and discovers the bucket that was read from has been reused while the IO was in flight, causing the read to be retried.

`extent_migrate_done`
This counts extents moved by the core move path, used by copygc and rebalance.

`extent_migrate_raced`
This counts extents that the move path attempted to move but no longer existed when doing the final btree update.

18.1.4 Unit and performance tests

Echoing into `/sys/fs/bcachefs/<uuid>/perf_test` runs various low level btree tests, some intended as unit tests and others as performance tests. The syntax is

```
echo <test_name> <nr_iterations> <nr_threads> > perf_test
```

When complete, the elapsed time will be printed in the dmesg log. The full list of tests that can be run can be found near the bottom of `fs/bcachefs/tests.c`.

18.2 Debugfs interface

The contents of every btree, as well as various internal per-btree-node information, are available under `/sys/kernel/debug/bcachefs/<uuid>/`.

For every btree, we have the following files:

btree_name
Entire btree contents, one key per line

btree_name`-formats`
Information about each btree node: the size of the packed bkey format, how full each btree node is, number of packed and unpacked keys, and number of nodes and failed nodes in the in-memory search trees.

btree_name`-bfloat-failed`
For each sorted set of keys in a btree node, we construct a binary search tree in eytzinger layout with compressed keys. Sometimes we aren't able to construct a correct compressed search key, which results in slower lookups; this file lists the keys that resulted in these failed nodes.

18.3 Listing and dumping filesystem metadata

18.3.1 bcachefs show-super

This subcommand is used for examining and printing bcachefs superblocks. It takes two optional parameters:

`-l`: Print superblock layout, which records the amount of space reserved for the superblock and the locations of the backup superblocks.

`-f, -fields=(fields)`: List of superblock sections to print, `all` to print all sections.

18.3.2 bcachefs list

This subcommand gives access to the same functionality as the debugfs interface, listing btree nodes and contents, but for offline filesystems.

18.3.3 bcachefs list_journal

This subcommand lists the contents of the journal, which primarily records btree updates ordered by when they occured.

18.3.4 bcachefs dump

This subcommand can dump all metadata in a filesystem (including multi device filesystems) as qcow2 images: when encountering issues that `fsck` can not recover from and need attention from the developers, this makes it possible to send the developers only the required metadata. Encrypted filesystems must first be unlocked with `bcachefs remove-passphrase`.

CHAPTER

NINETEEN

IOCTL INTERFACE

This section documents bcachefs-specific ioctls:

`BCH_IOCTL_QUERY_UUID`
Returs the UUID of the filesystem: used to find the sysfs directory given a path to a mounted filesystem.

`BCH_IOCTL_FS_USAGE`
Queries filesystem usage, returning global counters and a list of counters by `bch_replicas` entry.

`BCH_IOCTL_DEV_USAGE`
Queries usage for a particular device, as bucket and sector counts broken out by data type.

`BCH_IOCTL_READ_SUPER`
Returns the filesystem superblock, and optionally the superblock for a particular device given that device's index.

`BCH_IOCTL_DISK_ADD`
Given a path to a device, adds it to a mounted and running filesystem. The device must already have a bcachefs superblock; options and parameters are read from the new device's superblock and added to the member info section of the existing filesystem's superblock.

`BCH_IOCTL_DISK_REMOVE`
Given a path to a device or a device index, attempts to remove it from a mounted and running filesystem. This operation requires walking the btree to remove all references to this device, and may fail if data would become degraded or lost, unless appropriate force flags are set.

`BCH_IOCTL_DISK_ONLINE`
Given a path to a device that is a member of a running filesystem (in degraded mode), brings it back online.

`BCH_IOCTL_DISK_OFFLINE`
Given a path or device index of a device in a multi device filesystem, attempts to close it without removing it, so that the device may be re-added later and the contents will still be available.

BCH_IOCTL_DISK_SET_STATE

Given a path or device index of a device in a multi device filesystem, attempts to set its state to one of read-write, read-only, failed or spare. Takes flags to force if the filesystem would become degraded.

BCH_IOCTL_DISK_GET_IDX

BCH_IOCTL_DISK_RESIZE

BCH_IOCTL_DISK_RESIZE_JOURNAL

BCH_IOCTL_DATA

Starts a data job, which walks all data and/or metadata in a filesystem performing, performaing some operation on each btree node and extent. Returns a file descriptor which can be read from to get the current status of the job, and closing the file descriptor (i.e. on process exit stops the data job.

BCH_IOCTL_SUBVOLUME_CREATE

BCH_IOCTL_SUBVOLUME_DESTROY

BCHFS_IOC_REINHERIT_ATTRS

CHAPTER

TWENTY

ON DISK FORMAT

20.1 Superblock

The superblock is the first thing to be read when accessing a bcachefs filesystem. It is located 4kb from the start of the device, with redundant copies elsewhere - typically one immediately after the first superblock, and one at the end of the device.

The `bch_sb_layout` records the amount of space reserved for the superblock as well as the locations of all the superblocks. It is included with every superblock, and additionally written 3584 bytes from the start of the device (512 bytes before the first superblock).

Most of the superblock is identical across each device. The exceptions are the `dev_idx` field, and the journal section which gives the location of the journal.

The main section of the superblock contains UUIDs, version numbers, number of devices within the filesystem and device index, block size, filesystem creation time, and various options and settings. The superblock also has a number of variable length sections:

`BCH_SB_FIELD_journal`
List of buckets used for the journal on this device.

`BCH_SB_FIELD_members`
List of member devices, as well as per-device options and settings, including bucket size, number of buckets and time when last mounted.

`BCH_SB_FIELD_crypt`
Contains the main chacha20 encryption key, encrypted by the user's passphrase, as well as key derivation function settings.

`BCH_SB_FIELD_replicas`
Contains a list of replica entries, which are lists of devices that have extents replicated across them.

`BCH_SB_FIELD_quota`
Contains timelimit and warnlimit fields for each quota type (user, group and project) and counter (space, inodes).

`BCH_SB_FIELD_disk_groups`

Formerly referred to as disk groups (and still is throughout the code); this section contains device label strings and records the tree structure of label paths, allowing a label once parsed to be referred to by integer ID by the target options.

`BCH_SB_FIELD_clean`

When the filesystem is clean, this section contains a list of journal entries that are normally written with each journal write (`struct jset`): btree roots, as well as filesystem usage and read/write counters (total amount of data read/written to this filesystem). This allows reading the journal to be skipped after clean shutdowns.

20.2 Journal

Every journal write (`struct jset`) contains a list of entries: `struct jset_entry`. Below are listed the various journal entry types.

`BCH_JSET_ENTRY_btree_key`

This entry type is used to record every btree update that happens. It contains one or more btree keys (`struct bkey`), and the `btree_id` and `level` fields of `jset_entry` record the btree ID and level the key belongs to.

`BCH_JSET_ENTRY_btree_root`

This entry type is used for pointers btree roots. In the current implementation, every journal write still records every btree root, although that is subject to change. A btree root is a bkey of type `KEY_TYPE_btree_ptr_v2`, and the btree_id and level fields of `jset_entry` record the btree ID and depth.

`BCH_JSET_ENTRY_clock`

Records IO time, not wall clock time - i.e. the amount of reads and writes, in 512 byte sectors since the filesystem was created.

`BCH_JSET_ENTRY_usage`

Used for certain persistent counters: number of inodes, current maximum key version, and sectors of persistent reservations.

`BCH_JSET_ENTRY_data_usage`

Stores replica entries with a usage counter, in sectors.

`BCH_JSET_ENTRY_dev_usage`

Stores usage counters for each device: sectors used and buckets used, broken out by each data type.

20.3 Btrees

20.4 Btree keys

KEY_TYPE_deleted

KEY_TYPE_whiteout

KEY_TYPE_error

KEY_TYPE_cookie

KEY_TYPE_hash_whiteout

KEY_TYPE_btree_ptr

KEY_TYPE_extent

KEY_TYPE_reservation

KEY_TYPE_inode

KEY_TYPE_inode_generation

KEY_TYPE_dirent

KEY_TYPE_xattr

KEY_TYPE_alloc

KEY_TYPE_quota

KEY_TYPE_stripe

KEY_TYPE_reflink_p

KEY_TYPE_reflink_v

KEY_TYPE_inline_data

KEY_TYPE_btree_ptr_v2

KEY_TYPE_indirect_inline_data

KEY_TYPE_alloc_v2

KEY_TYPE_subvolume

KEY_TYPE_snapshot

KEY_TYPE_inode_v2

KEY_TYPE_alloc_v3