bcachefs: Principles of Operation

Kent Overstreet (and others)

March 7, 2026

Contents

1 Introduction and overview

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.

The entire filesystem is built on two primitives: btrees and buckets. All filesystem state is a key-value pair in a btree. A handful of btrees store core filesystem data (extents, inodes, dirents, xattrs, subvolumes), while others handle allocation tracking, reverse mappings, background maintenance, and crash recovery. There are no separate inode tables, bitmap allocators, or per-inode extent trees. On the storage side, all disk space is divided into buckets with generation numbers, described in the allocation section below. The btrees map logical filesystem objects to physical bucket locations, and backpointers provide the reverse mapping. Everything else in bcachefs is built on this transactional key-value store over generational bucket storage.

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.

1.1 Performance overview

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. Other COW filesystems use the Linux page cache for metadata, which limits them to 4K btree nodes; bcachefs manages its own btree node cache, so we can use 128K–256K nodes and tune reclaim independently of the page cache. At petabyte scale with spinning disks, deep btrees with 4K nodes mean more seeks per lookup and level-2 nodes too large to stay resident under page cache pressure.

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 easy 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.

1.2 Transactions

All btree operations go through a transaction layer (`btree_trans`) that provides atomic multi-key updates with optimistic concurrency control. Transactions collect updates in memory, then commit them atomically: first acquiring a journal reservation, then taking write locks on affected btree nodes in sorted order, applying updates, and releasing locks after the journal has accepted the change (but before the journal write is persisted to disk). This means btree locks are held only for in-memory operations, never across IO.

Btree nodes use six-locks (shared/intent/exclusive): the intent state allows an operation that modifies multiple nodes to hold intent locks for its duration, upgrading to write only for each individual in-memory update, avoiding the need to hold write locks across entire split or merge operations.

Rather than blocking on lock contention, when a transaction encounters a potential deadlock it drops all locks and restarts from the beginning. Deadlock detection is integrated into the lock acquisition path: before sleeping on a contended lock, the transaction checks for lock cycles and aborts if one is found. This restart-based approach avoids deadlocks entirely and keeps worst-case latency bounded, at the cost of requiring all transaction code to be idempotent and prepared for restarts at any point. This idempotency requirement has a profound consequence for reliability: since every operation can be safely interrupted and restarted, the filesystem is inherently resilient to interruption at any point - including during recovery itself. A crash during journal replay or an on-disk format upgrade simply results in the operation continuing from a safe point on the next mount. Recovery passes generally restart from the beginning and re-check everything; only passes that maintain persistent work queues can resume mid-pass.

Sequence numbers on btree nodes enable efficient relocking after a restart: if a node's sequence number hasn't changed, the lock can be reacquired without re-traversing the btree.

Transactions also support triggers: callbacks that fire on key insertion or deletion. Transactional triggers run before commit and may generate additional updates (e.g. updating backpointers when an extent changes). Atomic triggers run with the journal lock held and are used for accounting updates that must be exactly synchronized with the commit.

For operations that cannot be made atomic within a single transaction (such as truncate or file insert/collapse, which may need to modify an unbounded number of extents), bcachefs uses logged operations: the intent is written to the `logged_ops` btree before starting, and on recovery any incomplete logged operations are replayed to completion.

1.3 Recovery

On unclean shutdown, the journal is replayed to bring the btrees up to date. Journal entries record btree updates in commit order; replay applies them sequentially to reconstruct the state at the last completed commit.

To guarantee ordering of btree updates after a crash, we need to detect when a btree node was flushed to disk but its corresponding journal entry was not; otherwise, recovery could see later updates without the earlier updates they depended on.

During btree node reads, bcachefs detects this situation: bsets (sorted key arrays within a node) whose journal sequence number is newer than the last successfully written journal entry indicate the btree node was written to disk but the corresponding journal entry was not. These sequences are blacklisted in the superblock (`bch_sb_field_journal_seq_blacklist`), and bsets referencing them are ignored until the btree node is next rewritten. After an unclean shutdown, a padding of 64 sequence numbers past the last journal entry read is also blacklisted, to cover in-flight btree writes. Blacklisted sequences are never reused—the 64-bit counter simply skips past them. Blacklist entries are garbage collected once no btree node on disk references them.

1.3.1 Recovery passes

Beyond journal replay, bcachefs has approximately 48 recovery passes with an explicit dependency graph. Not all passes run on every mount - which passes run depends on context:

Every mount A default set of passes is scheduled: reading accounting, bucket generations, and snapshots first, checking snapshot consistency, then resuming logged operations and cleaning up dead inodes.

Unclean shutdown Adds journal replay. No extra repair passes are needed because the filesystem is always consistent after journal replay (absent bugs or data corruption).

Version upgrade/downgrade Each version transition defines specific passes to run for format migration.

Fsck A defined subset of validation passes runs: checking inodes, extents, dirents, xattrs, link counts, and directory structure. Note that `check_backpointers_to_extents` (the reverse direction check) is too expensive to run during normal fsck.

Recovery is split into two phases around the transition to read-write mode. Early passes run in-memory only, accumulating delayed metadata writes: reading btrees into memory, verifying btree topology, and reconstructing allocation state. Once the allocator and journal are initialized, the filesystem transitions to read-write and journal replay runs - followed by the heavier validation and repair passes that need to write.

The filesystem can schedule new recovery passes at any time, including during other recovery passes or normal operation, as soon as it notices something needs repair. In early recovery, this can rewind the recovery sequence to run earlier passes again. Most validation passes (those marked `PASS_ONLINE`) can also run after the filesystem is fully mounted, either triggered explicitly or scheduled automatically when an inconsistency is detected at runtime.

For catastrophic damage, heavier passes exist that reconstruct entire btrees from scratch. These are triggered automatically when normal recovery detects structural damage: `scan_for_btree_nodes` is scheduled when `check_topology` finds missing or unreachable btree nodes - it physically scans the raw device for valid btree node headers and recovers what it can. `reconstruct_snapshots` is scheduled when the snapshots btree itself is lost, or when references to missing snapshots cannot be resolved by deletion alone and recovering those snapshots is feasible. `check_allocations` rebuilds all allocation accounting from the extents and backpointers btrees. These passes are not part of normal recovery and may take considerable time on large filesystems.

When a pass is scheduled at runtime (post-mount), its persistence depends on severity. Passes scheduled without the `nopersistent` flag are written to the superblock's `recovery_passes_required` field and survive reboots: the pass will run automatically at the next mount even if the filesystem is cleanly unmounted. Ephemeral scheduling (used when a prior pass must complete first before the repair makes sense) is remembered only until the current recovery sequence completes and does not touch the superblock. Once a pass runs successfully, it clears its own bit from `recovery_passes_required`.

Some passes are scheduled with a ratelimit flag, used for errors that may be detected in large numbers during a single I/O operation (for example, EC accounting inconsistencies during a stripe write). A ratelimited pass is not rescheduled unless the same error is detected again via a non-ratelimited request, preventing repair-loop flooding. The ratelimiting state is in-memory only and does not affect persistence.

The complete list of recovery passes, in execution order:

`scan_for_btree_nodes` Scan all devices for btree nodes by magic number, deduplicate replicas, and build node scan table for topology repair

`check_topology` Verify btree roots exist (reconstructing from node scan if missing), then recursively validate parent-child links and min/max key boundaries

`accounting_read` Read accounting keys from btree and journal into memory, merging deltas and initializing per-device usage counters *(always)*

`alloc_read` Populate in-memory bucket generation cache from bucket_gens btree (or alloc btree on older filesystems) *(always)*

`stripes_read` Reserved for erasure-coding stripe initialization; currently a no-op

`initialize_subvolumes` Create root snapshot tree, root snapshot node, and root subvolume for a new filesystem

`snapshots_read` Populate in-memory snapshot table with ancestry bitmaps and depth info by iterating snapshot btree in reverse order *(always)* `check_allocations` Full GC pass: walk all btrees marking referenced buckets, then compare against alloc btree to repair data_type, sector counts, and stripe refs *(fsck, alloc)*

`journal_replay` Replay pending journal keys into btrees, accounting keys first; sorted-order bulk insert with per-key fallback for journal deadlocks *(always)*

`merge_btree_nodes` Merge adjacent underfull btree nodes to reclaim wasted space *(online)*

`check_alloc_info` Cross-check alloc btree against freespace, need_discard, and bucket_gens btrees; repair missing or incorrect entries in each *(fsck, online, alloc)*

`check_lrus` Verify LRU btree entries match alloc key timestamps for cached-data and fragmentation LRUs; delete stale entries *(fsck, online, alloc)*

`check_btree_backpointers` Verify every backpointer entry references a valid alloc key; remove backpointers for nonexistent buckets *(fsck, online, alloc)*

`check_backpointers_to_extents` Verify each backpointer matches an actual extent or btree pointer at the claimed location; remove stale entries *(online)*

`check_extents_to_backpointers` Find buckets with missing backpointers by scanning alloc btree, then regenerate them from extent and btree pointer data *(fsck, online, alloc)*

`check_alloc_to_lru_refs` Ensure cached buckets have correct cached-data LRU entries and fragmentable buckets have correct fragmentation LRU entries *(fsck, online, alloc)*

`bucket_gens_init` Populate bucket_gens btree from alloc btree generation numbers; one-time migration

`reconstruct_snapshots` Scan snapshot-bearing btrees to find snapshot IDs in use, then reconstruct missing snapshot nodes and tree entries

`delete_dead_interior_snapshots` Collapse interior snapshot nodes with no remaining keys by re-parenting their single live child

`check_snapshot_trees` Validate snapshot_tree entries: root snapshot reference, back-link consistency, master_subvol points to a real non-snapshot subvolume *(fsck, online)*

`check_snapshots` Validate snapshot btree in reverse order: parent/child bidirectional links, tree_id, depth, subvol flag, and skiplist pointers *(always, fsck, online, nodefer)* `check_subvols` Validate subvolume entries: snapshot exists, root inode has correct bi_subvol, fs_path_parent is valid; delete unlinked subvols *(fsck, online)*

`check_subvol_children` Walk subvolume_children btree and remove entries not matching a real subvolume with correct fs_path_parent *(fsck, online)*

`delete_dead_snapshots` Delete snapshot data keys across all snapshot-bearing btrees, then remove snapshot nodes and mark empty interior nodes *(fsck, online)*

`fs_upgrade_for_subvolumes` One-time migration: set bi_subvol on root inode for pre-subvolumes filesystems

`check_inodes` Validate inode fields (mode, flags, i_size, bi_subvol), delete orphaned unlinked inodes, repair invalid backpointers *(fsck)*

`check_extents` Validate extent keys: owning inode exists, snapshot valid, no overlaps, i_size and i_sectors consistent *(fsck)*

`check_indirect_extents` Validate reflink indirect extents; drop stale device pointers whose generation no longer matches *(fsck, online)*

`check_dirents` Validate directory entries: target inode exists in correct snapshot, d_type matches inode mode, hash values correct *(fsck)*

`check_xattrs` Validate xattr entries: owning inode exists in valid snapshot, hash correct; delete orphans *(fsck)*

`check_root` Ensure root subvolume and root directory inode exist; create them if missing *(fsck, online)*

`check_unreachable_inodes` Find inodes with no directory entry (unset bi_dir backpointer); reattach to lost+found *(fsck)*

`check_subvolume_structure` Follow each subvolume's fs_path_parent chain to root, verify no cycles or dead ends; reattach disconnected subvolumes *(fsck, online)*

`check_directory_structure` DFS from each directory following parent pointers, detect cycles, renumber bi_depth, reattach disconnected directories *(fsck, online)*

`check_nlinks` Two-pass nlink verification: collect hardlinked inodes, count directory references, fix bi_nlink mismatches *(fsck)*

`check_reconcile_work` Validate reconcile work/hipri/pending/scan btrees against actual extent data; remove stale entries *(fsck, online)* `resume_logged_ops` Resume incomplete logged operations (fallocate, stripe creation) from logged_ops btree, then delete completed entries *(always)*

`delete_dead_inodes` Scan deleted_inodes btree and fully remove inodes with nlink == 0 that are not open *(always)*

`kill_i_generation_keys` Remove KEY_TYPE_inode_generation keys; this older generation persistence mechanism has been superseded *(online)*

`fix_reflink_p` One-time migration: clear stale front_pad/back_pad fields in KEY_TYPE_reflink_p keys

`set_fs_needs_reconcile` One-time pass: insert full-filesystem reconcile_scan entry to trigger background data verification

`btree_bitmap_gc` Recompute per-device btree_allocated_bitmap by scanning all live btree node pointers *(online)*

1.4 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. The generation number mechanism was originally created for invalidating cached data, but it turned out to be valuable well beyond that: it makes bootstrapping allocation information during journal replay much more tractable, and it makes it possible to repair certain kinds of corruption that would be unrecoverable otherwise.

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 - 8% by default (configurable from 5% to 21%). A fragmentation LRU btree tracks bucket fill levels so that copygc can find the most fragmented buckets directly instead of scanning the entire alloc btree.

The allocator originally used in-memory bucket arrays and dedicated scanning threads to maintain free lists, discard lists, and eviction heaps. These were replaced by btree-based data structures: a freespace btree, a discard btree, and an LRU btree. The scanning threads are gone entirely; the code that replaces them is transactional btree code, much of it trigger-based, which is far easier to debug and reason about. The old threads were also prone to excessive CPU usage when the filesystem was nearly full; the btree approach eliminated those corner cases.

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 are intended to be used for managing SMR (shingled magnetic recording) and zoned hard drives directly, avoiding the translation layer in the drive. Buckets map naturally to zones: when we allocate a bucket, we write to it once in an append-only fashion, then never write to it again until we discard and reuse the whole thing — exactly the semantics a zoned device requires.

1.5 Backpointers

Every sector range on disk that contains data or metadata has a corresponding backpointer: a reverse reference from the physical location back to the logical btree entry that owns it. Backpointers enable efficient answers to the question "what data lives in this bucket?" without scanning the entire extents btree. This is essential for copygc, device evacuation, scrub, and the reconcile subsystem.

See Backpointers in subsystem details for on-disk structure, maintenance, and consistency validation.

2 Feature overview

2.1 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 set-file-option` command, they will be automatically applied recursively.

Checksumming, compression, and encryption all operate at the granularity of entire extents (up to `encoded_extent_max`, default 256 KB). An extent that has been through any of these transformations is called an *encoded extent*. Because the checksum covers the full encoded extent, any read within it — even a single sector — must read, verify, and (if compressed) decompress the whole thing. This is the fundamental performance tradeoff shared by all three features: they improve data integrity and storage efficiency at the cost of read amplification for small random IO. For workloads dominated by small random reads (databases, VMs), reducing `encoded_extent_max` or disabling these features may be appropriate.

2.1.1 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.

2.1.2 Encryption

bcachefs supports authenticated (AEAD) encryption using ChaCha20/Poly1305. Unlike typical block layer or filesystem encryption (usually AES-XTS), which operates on fixed blocks with no room for nonces or MACs, bcachefs stores a nonce and cryptographic MAC alongside every data pointer. This creates a chain of trust from the superblock (or journal) down to individual extents: any modification, deletion, reordering, or rollback of metadata is detectable.

Encryption is all-or-nothing at the filesystem level: all data and metadata except the superblock is encrypted, and all data and metadata is authenticated. Per-file or per-directory encryption is not supported because btree nodes are shared structures that do not belong to any individual file. Encryption can only be enabled at format time; it cannot be added to an existing filesystem.

Critical exception: Data written with the `nocow` option is stored **unencrypted**, even on an encrypted filesystem (see the Nocow section below).

The key hierarchy uses a passphrase (never stored on disk) to derive a passphrase key via scrypt, which decrypts a random master key stored in the superblock. The master key encrypts all data and metadata. Changing the passphrase re-encrypts only the master key, not filesystem data. The KDF runs in userspace, so alternative key sources (hardware tokens, key files) can be integrated without kernel changes. There is currently no key rotation, key escrow, or multi-passphrase support. Losing the passphrase means permanent data loss.

At mount time, the kernel retrieves the passphrase key from the Linux kernel keyring (`bcachefs:<UUID>`). Note that keys added to a session keyring are not visible from other sessions — use `KEY_SPEC_USER_KEYRING` for cross-session visibility. Similarly, `sudo mount` uses root's keyring, not the calling user's.

By default, MACs are truncated to 80 bits, which is sufficient for most threat models. The `wide_macs` option stores the full 128-bit MAC and is recommended when the storage device itself is untrusted (e.g. USB drives, network storage). Metadata always uses full-width MACs.

2.1.2.1 Nonce reuse with external snapshots

If the underlying storage is snapshotted externally (LVM, ZFS zvol, VM snapshot) and the snapshot is mounted read-write, both instances share the same master key and will reuse nonces, breaking encryption security. bcachefs's own snapshots do not have this problem.

Mitigation: Never mount an external snapshot of an encrypted volume read-write — keep external snapshots read-only (`-o nochanges`). Alternatively, place LUKS between the snapshot layer and bcachefs (e.g. LVM → LUKS → bcachefs).

See Encryption in subsystem details for key hierarchy internals, keyring integration, and MAC storage details.

2.1.3 Compression

bcachefs supports gzip, lz4 and zstd compression, operating at extent granularity as described above. This gives better compression ratios than filesystems that compress at the 4K page level: compression algorithms find more redundancy when fed more data at once, and rounding compressed output back up to block alignment at small granularity loses much of the compression ratio. Per-extent granularity also simplifies the IO path: everything works in terms of extents, and there is nothing smaller than an extent to write code to handle.

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

2.1.4 Nocow

The `nocow` option enables in-place writes: data is overwritten directly rather than being written to a new location (copy-on-write). This bypasses the normal encoded-extent pipeline — nocow data is not checksummed, not compressed, and not encrypted, even on an encrypted filesystem.

Nocow mode is useful for workloads that require stable disk offsets or that cannot tolerate the write amplification of COW (databases, VM images). It can be set per-file or per-directory.

Snapshots and reflink still trigger COW semantics even for nocow files — when a nocow file has shared extents (via snapshot or reflink), writes to those extents will COW as usual. Once the shared reference is gone, subsequent writes resume in-place.

Because nocow data has no checksums, it cannot be verified by scrub, cannot be self-healed from a replica on checksum mismatch, and is not protected against bitrot. On encrypted filesystems, nocow data is stored in plaintext. These tradeoffs should be weighed carefully; for most workloads, the default COW behavior with checksumming is the better choice.

2.2 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.

2.2.1 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.

Replica placement is flexible: each write picks devices from those available for the requested data type, respecting target group constraints and balancing across devices by free space. When a read detects a checksum error on one replica, it transparently falls back to another replica and repairs the corrupted copy by rewriting it. If the desired replica count is reduced (or increased), the reconcile subsystem adjusts existing data in the background.

2.2.2 Erasure coding

bcachefs supports Reed-Solomon erasure coding, the same algorithm used by most RAID5/6 implementations. Erasure coding is a per-inode option (`erasure_code`), so it can be enabled on specific directories via `set-file-option` without affecting the rest of the filesystem. The desired redundancy is taken from the `data_replicas` option: `data_replicas=2` yields one parity block (RAID-5 style), `data_replicas=3` yields two (RAID-6 style), and `data_replicas=1` disables erasure coding entirely. Parity is limited to at most two blocks. Erasure coding of metadata is not supported.

In conventional RAID, the "write hole" is a significant problem: a small write within a stripe requires updating the P and Q (parity) blocks, and since those writes cannot be done atomically, a crash can leave parity inconsistent, corrupting reconstruct reads for unrelated data in the same stripe. ZFS avoids this by making every write a new stripe, but the resulting fragmentation hurts performance: reads of fragmented data are bounded by the latency of the slowest fragment, driving median latency toward tail latency.

bcachefs takes advantage of copy-on-write to avoid both problems. Since updating stripes in place is the root cause, we simply never do it. And since per-extent stripes would cause the same fragmentation as ZFS, we erasure code entire buckets, taking advantage of bucket-based allocation and copying garbage collection.

See Erasure coding in subsystem details for write path mechanics, stripe layout, fragmentation handling, and on-disk representation.

2.2.3 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

2.2.4 Caching and durability

Devices can be configured for tiered storage using the target options and per-device durability settings. See Device management in subsystem details for the full treatment of caching modes (writeback, writearound), durability counting, and how they interact with replication.

2.2.5 Splitbrain detection

When assembling a multi-device filesystem, bcachefs detects devices that have diverged—been mounted and modified independently. Every superblock records a sequence number (`seq`) and write timestamp (`write_time`); devices with the same sequence but different write times have diverged.

Since version 1.4, superblocks also track the last known sequence of every other member device—a *vector clock*. A vector clock is a distributed-systems technique where each participant maintains a vector of counters, one per participant. Each device's superblock records not just its own sequence number, but the last sequence number it observed for every other device. When devices are reassembled, bcachefs compares each device's actual sequence against what the other devices expected it to be: if device A's actual sequence exceeds what device B recorded for it, then A was modified independently while B was offline. This catches divergence even in cases where simple timestamp comparison would not (e.g. clock skew, or writes that happen to produce the same sequence but different content).

The two common causes of splitbrain are:

- **Partial mount of replicated filesystem**: Mounting a subset of devices from a replicated filesystem (`replicas >= 2`) with `-o degraded=very` allows writes to proceed despite missing devices. If the "missing" device is later reconnected, it now has stale metadata while the mounted devices have advanced.
- **External snapshots**: LVM, SAN, or VM snapshots create block devices with valid bcachefs superblocks sharing the same filesystem UUID. If both original and snapshot are visible during assembly, or if the snapshot is mounted read-write and later reassembled with the original, the devices have diverged.

When divergence is detected, bcachefs excludes the stale device. Once devices have genuinely diverged with writes on both sides, recovery is impractical - resolving differences key-by-key across all btrees is not feasible. The `-o no_splitbrain_check` option includes divergent devices anyway, but is only safe when no actual modifications occurred on the excluded device.

2.3 Reflink

bcachefs supports reflink (`cp -reflink`, `FICLONE` ioctl), creating copies that share underlying storage. The original extent is moved to a separate reflink btree with a reference count; reads through a reflinked extent require two btree lookups instead of one. Writing new data over a reflinked range works through normal COW — the refcount is decremented, and when it reaches zero the shared extent and its disk space are freed.

Reflink is currently a one-way transformation: once an extent becomes indirect, it stays indirect even when only one reference remains. This means a file that was once reflinked permanently pays the double-lookup cost on reads.

IO option propagation (compression, checksums, replicas) for shared extents is controlled by a security boundary: only the source file's options can propagate to the shared data, preventing a reflink copy from degrading another user's data protection settings.

See Reflink in subsystem details for the full on-disk representation, lifecycle mechanics, and snapshot interaction.

2.4 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 by default and may be snapshotted again, creating a tree of snapshots; they can also be created as read-only with the `-read-only` (or `-r`) flag.

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 delete`: Delete an existing subvolume or snapshot
- `bcachefs subvolume snapshot`: Create a snapshot of an existing subvolume

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

See Subvolumes and snapshots in subsystem details for the full architecture, key visibility model, and snapshot lifecycle.

2.5 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. Quota accounting resolves every inode through the master subvolume only, charging sectors and inode counts to the uid/gid/project recorded there. Writes to snapshot subvolumes bypass quota enforcement entirely. If there is no master subvolume for a snapshot tree, quota accounting for that tree is skipped.

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.

2.6 32-bit inodes

The `inodes_32bit` option restricts new inode numbers to 32 bits. This is needed for legacy NFS compatibility (NFSv2 and some older NFSv3 implementations use 32-bit inode numbers), for 32-bit userspace that may truncate 64-bit inode numbers, and for Wine/Proton/Steam which run 32-bit Windows applications.

This option can be set per-directory, affecting only files created within that directory tree.

Note that `inodes_32bit` silently disables `shard_inode_numbers_bits`: there are too few bits in the 32-bit space to make CPU-based sharding practical.

2.7 Casefolding

bcachefs supports case-insensitive filenames via the Linux kernel's Unicode subsystem (`CONFIG_UNICODE`). The `casefold` option is a per-directory property that is inherited by newly created subdirectories.

Important: Casefolding can only be enabled on an empty directory. Existing entries cannot be converted because they would need to be rehashed with case-folded names. On disk, casefolded directories store both the original filename and its case-folded form; the case-folded form is used for lookups while the original is preserved for display.

The Unicode version used for case folding is hardcoded to Unicode 12.1.0. The kernel will fail to mount (or use an older version correctly) if built with different Unicode tables.

2.8 Migration

`bcachefs migrate` converts an existing filesystem (ext4, XFS, etc.) to bcachefs in place, without reformatting or copying data to a separate device. The original data is preserved and incorporated into the new bcachefs filesystem. After migration, `bcachefs migrate-superblock` writes a standard superblock so the filesystem can be mounted normally.

2.9 Images

`bcachefs image create` builds a bcachefs filesystem image from a directory tree, suitable for embedding in OS images, containers, or VM disks. `bcachefs image update` incrementally updates an existing image to reflect changes in the source directory, minimizing the amount of data rewritten.

3 Management

3.1 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
```

3.2 Mounting

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

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

Or, use the mount.bcachefs tool to mount by filesystem UUID or label:

```
mount -t bcachefs UUID=<uuid> /mnt
mount -t bcachefs LABEL=<label> /mnt
```

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 devices, but will fail if data would be missing. The `-o degraded=very` option can be used to attempt mounting when data would be missing.

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

3.3 Monitoring

Three commands provide a comprehensive view of filesystem health and performance:

`bcachefs fs usage` Displays filesystem space usage broken out by data type (superblock, journal, btree, data, cached data, parity), by which sets of devices extents are replicated across, and per-device usage including fragmentation from partially-used buckets. This is the primary tool for understanding where space is going and how data is distributed across devices.

`bcachefs fs top` Real-time display of btree operations by process, similar to `top(1)` for CPU usage. Shows which processes are generating btree reads, writes, and transaction restarts. This is the primary tool for diagnosing metadata performance problems—if the filesystem feels slow, `fs top` will show what's generating the load. Every counter displayed by `fs top` has a corresponding tracepoint, so once you identify which operation is hot, you can drill down into individual events with `perf trace` or `bpftrace`.

`bcachefs fs timestats` Real-time display of operation latency statistics: min, max, mean, standard deviation, and EWMA for both duration and inter-arrival time. Covers all major IO and internal operations (data reads, data writes, btree reads, btree splits, journal flushes, copygc, etc.). This is the primary tool for understanding latency—if an operation is slow, timestats will show which stage of the pipeline is responsible.

For deeper investigation, sysfs (`/sys/fs/bcachefs/<uuid>/`) provides per-option tunables, persistent counters, and internal state; debugfs (`/sys/kernel/debug/bcachefs/<uuid>/`) provides btree contents, transaction state, lock contention, and journal pin diagnostics. See §7 for details.

3.4 Journal tuning

The journal has tunables that affect filesystem performance; see Journal in subsystem details for the full architecture. The key tunables:

`journal_flush_delay` Milliseconds before auto-flushing (default 1000). Sync and fsync force immediate flushes.

`journal_flush_disabled` Disables sync/fsync flushes. The `journal_flush_delay` timer still applies, so at most one second of work is lost on crash. Useful on workstations, less appropriate on servers.

`journal_reclaim_delay` Background reclaim interval (default 100 ms).

The journal should be sized so that bursts of activity do not fill it; a larger journal also allows larger batched btree writes. Use `bcachefs device resize-journal` to grow the journal online. Current utilization is visible via `/sys/fs/bcachefs/<uuid>/internal/journal_debug`.

3.5 Device management

Devices can be added, removed, resized, and taken online/offline at any time without unmounting. The `bcachefs device` subcommands handle all device lifecycle operations; see Device management in subsystem details for the full treatment of device states, durability, caching, degraded mode, and error tracking.

3.6 Data management

3.6.1 Reconcile

The reconcile subsystem (formerly "rebalance") runs in the background to ensure all data and metadata matches configured IO path options: replication count (`data_replicas`, `metadata_replicas`), compression, checksum type, erasure coding, and device targets (`foreground_target`, `metadata_target`, etc.). When options change or devices are added or removed, reconcile propagates these changes to existing data and metadata by moving, rewriting, or recompressing extents and btree nodes.

Reconcile uses 6 dedicated btrees for work tracking: `reconcile_work` and `reconcile_hipri` for normal and high-priority logical work, `reconcile_pending` for work that failed due to insufficient space or devices, `reconcile_scan` for option change propagation via scan cookies, and physical variants (`reconcile_work_phys`, `reconcile_hipri_phys`) that track work by device LBA for efficient processing on rotational devices.

Work enters the system when filesystem or inode options change (triggering a scan), or when extent triggers detect a mismatch between current data placement and desired options. The reconcile thread processes work in priority order: high-priority metadata first (under-replicated or evacuating), then high-priority data, then normal metadata (moving stray metadata to `metadata_target`), then normal data, then pending retries (only after device configuration changes). The `reconcile_pending` btree solves the long-standing "rebalance spinning" problem, where the old rebalance thread would burn CPU retrying moves that could not complete (e.g. because the target was full). A practical consequence: you can create a single-device filesystem with `replicas=2`, and all data will be marked as degraded with 1x availability; when a second device is added, reconcile automatically replicates everything to 2x without further user intervention.

The `bcachefs reconcile status` command shows current reconcile progress, and `bcachefs reconcile wait` blocks until reconcile completes for specified work types. The `reconcile_enabled` and `reconcile_on_ac_only` options control when reconcile runs.

3.6.2 Scrub

Scrub reads all data on a running filesystem and verifies checksums, detecting silent data corruption (bitrot). When a checksum mismatch is found and a valid redundant copy exists (from replication or erasure coding), the corrupted copy is automatically repaired by rewriting it from the good copy. Scrub walks data in physical (LBA) order using backpointers, which is efficient for rotational devices and avoids the random access pattern that would result from walking the logical extent tree.

Scrub can be run on a specific device or on all devices in the filesystem. Progress is reported via sysfs and can be monitored with `bcachefs data scrub`. Affected file paths are currently logged to the kernel log only.

4 Repair and recovery

bcachefs is designed to detect, report, and repair problems at every level: inline during normal IO, in the background via scheduled recovery passes, and through explicit administrator-driven tools. This section covers all of these mechanisms.

4.1 Online self-healing

Bcachefs classifies metadata errors into three tiers based on severity and whether a safe automatic repair is known:

Autofix errors (FSCK_AUTOFIX) Inconsistencies with a well-understood, always-safe repair that can be applied immediately inline during normal operation, without interrupting I/O or requiring a recovery pass. An example is a missing journal entry replica marker: the repair is simply to write the correct marker, with no data at risk.

Fixable errors (FSCK_CAN_FIX) Inconsistencies that have a known repair procedure but require a dedicated recovery pass to execute safely. Under `errors=fix_safe` (the default since version 1.19), detecting such an error schedules the relevant recovery pass to run online in the background. The pass runs concurrently with normal filesystem operation without taking the filesystem offline, provided it carries the `PASS_ONLINE` flag. Errors detected during recovery that would rewind the recovery sequence are handled specially: recovery restarts from the affected pass rather than aborting.

Inconsistent errors Structural corruption for which no safe automatic repair is known, or where continuing operation risks further data loss. These trigger emergency read-only regardless of the `errors` setting (except `errors=continue`, which logs the error and proceeds). The filesystem logs a message directing the administrator to run fsck. If a repair procedure is subsequently developed and deemed safe, the error can be reclassified to `FSCK_CAN_FIX` in a future kernel release.

The `errors` mount option controls the policy for the second and third tiers; see §6.4. The default `fix_safe` enables background self-repair for fixable errors while taking the conservative read-only path for inconsistent ones.

Read path self-healing: When a read detects a checksum mismatch and a valid replica exists, the corrupted copy is automatically repaired by rewriting it from the good copy. This happens transparently on every read without administrator intervention. If erasure coding is available, data can be reconstructed from parity even with no good replica.

Error accounting: Every error detection event is recorded as a persistent count in the superblock's errors section. These counts survive across reboots and accumulate over the filesystem's lifetime, providing an audit trail of what has been detected and repaired. The counts are visible in `bcachefs show-super` output and via the sysfs `errors` file.

4.2 Fsck

The `bcachefs fsck` subcommand (also available as `fsck.bcachefs`) can run fsck in userspace (`-K`), in the kernel at mount time (`-k`), or online on a mounted filesystem, selecting the mode automatically or via command line options. The in-kernel offline mode is useful when an older bcachefs-tools binary cannot handle the filesystem version. In all cases the exact same fsck implementation runs—the recovery passes described in the introduction. Running in userspace allows stopping with ctrl-c and prompting for fixes; `bcachefs fsck -y` auto-fixes, `-n` runs read-only.

Online fsck runs the validation passes marked as safe for concurrent use (`PASS_ONLINE`) on a mounted, read-write filesystem. Combined with `errors=fix_safe`, this means most metadata inconsistencies are detected and repaired automatically without administrator intervention.

4.2.1 Safe inspection with nochanges

The `-o nochanges` mount option disallows any writes to the underlying devices, pinning dirty data in memory if journal replay is necessary. This is the primary tool for safe inspection of a damaged filesystem: it allows recovery to run and metadata to be read without risking further damage.

`bcachefs fsck -n` implies `-o nochanges`. To test a repair before committing: `-o ro,nochanges,fsck,fix_errors` runs the full repair in memory without writing to disk, allowing the filesystem to be mounted and its contents verified before repeating without `nochanges`.

4.3 Journal rewind

The `journal_rewind` mount option accepts a journal sequence number and rolls back filesystem metadata to that point, discarding subsequent updates and fixing up inconsistencies afterwards. This is a disaster recovery tool for reverting damage from a bug in recovery that irreversibly corrupted or deleted metadata, or (combined with `-o nochanges`) for attempting to recover recently deleted files.

Journal rewind drops all leaf-level btree updates to non-allocator btrees at or above the specified sequence number, then automatically triggers a full fsck to rebuild allocation metadata.

4.3.1 Usage

Journal rewind is used in conjunction with `bcachefs list_journal` to identify the sequence number to rewind to. The `journal_transaction_names` option (on by default) is required: it controls whether `BCH_JSET_ENTRY_overwrite` entries (the old values being replaced) are stored in the journal, which is what rewind replays to undo changes. It also records which function created each journal entry, making it easier to identify which transactions caused a problem.

4.3.2 Limitations

Since rewind operates on metadata only, data in buckets that have been reused (generation incremented and overwritten with new data) is unrecoverable, and `nocow` data cannot be rolled back at all. The journal records allocator updates, so `list_journal -b alloc` can identify when specific buckets were recycled. The rewind window is bounded by the journal size: only sequences between `last_seq` (oldest entry still on disk) and the current sequence are available, so a larger journal provides a deeper rewind history.

4.4 Disaster recovery

bcachefs is designed to recover from catastrophic metadata damage—not just the routine inconsistencies that fsck handles, but situations where entire btrees are lost or corrupted beyond normal repair. The key insight is that the filesystem's data is recoverable as long as the leaf nodes of the extents and dirents btrees survive, because those two btrees contain the essential mappings: which data belongs to which file, and which files exist in which directories. Everything else can be reconstructed from them.

In practical terms: if every btree except extents and dirents is destroyed, recovery can still place every file's data in its correct directory with its correct name. File sizes may be slightly off if inode metadata is lost (inodes record the precise file size, while extents only record block-aligned ranges), and permissions, ownership, and timestamps will be lost with the inodes, but the actual data—the bytes the user cares about—will be intact and in the right place. This is a much stronger recovery guarantee than most filesystems provide.

4.4.1 Reconstruction passes

When normal recovery detects structural damage, it automatically schedules heavier reconstruction passes:

`scan_for_btree_nodes` + `check_topology` These two passes work together. When `check_topology` finds missing or unreachable btree nodes, it schedules `scan_for_btree_nodes`, which physically scans every device for valid btree node headers (identified by magic number), deduplicates replicas, and builds a table of recovered nodes. Then `check_topology` runs again, using the scan results to reconstruct missing interior nodes and reconnect orphaned subtrees. Together they can recover a btree even when the root node and multiple levels of interior nodes are lost—as long as the leaf nodes are intact. This reads every sector of every device and may take considerable time on large filesystems.

`reconstruct_snapshots` Scheduled when the snapshots btree itself is lost, or when references to missing snapshots cannot be resolved by deletion alone. Scans all snapshot-aware btrees to find which snapshot IDs are actually in use, then reconstructs the snapshot tree structure from that evidence.

`check_allocations` Full allocation rebuild: walks all btrees marking which buckets are referenced, then compares against the alloc btree to repair data types, sector counts, and stripe references. This is the pass that makes it possible to lose the entire alloc btree and recover—the allocation state is fully derivable from the extent and backpointer btrees.

These passes are not part of normal recovery and are only triggered when damage is detected. They run automatically—no special flags or manual intervention is required. After reconstruction, the filesystem schedules a full fsck to validate the rebuilt metadata.

4.4.2 Superblock recovery

The superblock itself has multiple layers of redundancy. Up to 61 backup copies can be stored on each device, and every device in the filesystem has a complete copy of the superblock. The `bcachefs recover-super` command can reconstruct a completely overwritten primary superblock from backup copies on the same device or from any other device in the filesystem.

4.4.3 Diagnostic tools

When investigating damage before attempting repair:

`bcachefs fsck -n` Runs all recovery and repair passes without writing to disk (`-o nochanges`). Shows what would be repaired.

`bcachefs list` Dumps btree contents for offline inspection. Useful for examining what data survived in specific btrees.

`bcachefs list_journal` Lists journal contents with sequence numbers, transaction names, and per-btree filtering. Essential for understanding what happened before a crash and for identifying journal rewind targets.

`bcachefs dump` Exports all metadata as qcow2 images for offline analysis, without including user data. For encrypted filesystems, the passphrase must first be removed with `bcachefs remove-passphrase`.

5 Command reference

The `bcachefs` tool provides filesystem management through grouped subcommands. Run `bcachefs` with no arguments for a command summary, or `bcachefs <command> -help` for detailed usage.

5.1 Superblock commands

`bcachefs format` (alias: `mkfs`) Format a new filesystem

Formats one or more devices as a new bcachefs filesystem. Supports multi-device filesystems with configurable replication, erasure coding, encryption (chacha20/poly1305), and compression. All filesystem options from Section 6 can be set at format time. Per-device options (labels, durability, data_allowed) can be set independently for each device.

`bcachefs show-super` Print superblock information to stdout

Prints superblock information for a bcachefs device. By default shows the core superblock header, member device information, and error counters. Use -f to select specific superblock sections (comma-separated) or -f all to display every section. Use -F for scripting: prints a single named field with no header. Use --layout to display the superblock layout (backup locations and sizes).

`bcachefs recover-super` Attempt to recover overwritten superblock from backups

Attempts to find and restore a bcachefs superblock from backup copies on the device. The device is scanned for valid superblock signatures and the most recently mounted copy is selected. In multi-device filesystems, the superblock can also be recovered from another member device using --src_device.

`bcachefs set-fs-option` Set a filesystem option

Set a filesystem or device option on a running filesystem. Changes are persisted to the superblock. Use -d to target a specific device for device-scoped options. See Section 6 for the full list of available options.

`bcachefs reset-counters` Reset filesystem counters

Reset persistent counters stored in the superblock. Operates on an unmounted device. By default all counters are reset; use --counters to reset specific ones. See Section 7.1.3 for a list of available counters.

`bcachefs strip-alloc` Strip alloc info on a filesystem to be used read-only

Removes allocation metadata and journal from a filesystem intended for read-only use. Alloc info is regenerated on first read-write mount. Currently unsafe on large (multi-TB) filesystems due to memory requirements of alloc info reconstruction; this limitation will be lifted when online check_allocations is complete.

5.2 Images

bcachefs can generate compact, read-only filesystem images from a directory tree in a single command. Images support all filesystem features including compression, and are generally competitive with EROFS in size.

By default, allocation metadata is stripped from the image and regenerated on first read-write mount (requires kernel 6.16+); regeneration is fast enough to be unnoticeable on typical image sizes. Use --keep-alloc to preserve it.

`bcachefs image create` Create a filesystem image

Creates a new filesystem image file from a source directory. The image is written sequentially using a temporary metadata device, then compacted and truncated to its final size.

`bcachefs image update` Update a filesystem image, minimizing changes

> Updates an existing filesystem image to match a source directory with minimal on-disk changes. Only modified data is rewritten, making this suitable for delta transfer workflows (e.g. rsync the image after update).

5.3 Mount

`bcachefs mount` Mount a bcachefs filesystem by its UUID

> Mounts a bcachefs filesystem. Devices are discovered automatically by scanning for the filesystem UUID—unlike btrfs, this is handled entirely in userspace.
>
> If the filesystem is encrypted, the passphrase will be looked up in the kernel keyring first; if not found, the user is prompted interactively (or reads from stdin if not a terminal). Use -k or -f to specify alternative unlock methods.

5.4 Repair

`bcachefs fsck` Check an existing filesystem for errors

> Runs both online and offline filesystem checks. When given a mountpoint or a device that is already mounted, fsck runs online via ioctl. When given an unmounted device, it runs offline. The in-kernel fsck implementation is preferred when the kernel and filesystem versions match; use -k to force kernel fsck or -K to force userspace.
>
> Not all fsck passes are available online yet; offline fsck runs the full set of recovery passes.

`bcachefs recovery-pass` List and manage scheduled recovery passes

> Lists recovery passes scheduled to run at next mount, and allows forcing specific passes to run or clearing scheduled passes. Recovery passes handle everything from btree topology repair to allocation consistency checks; see the recovery passes table for the full list.

5.5 Running filesystem

`bcachefs fs usage` Display detailed filesystem usage

> Displays filesystem space usage broken down by category. Output modes: replicas (data/metadata replication), btree (per-btree space), compression (ratios and savings), rebalance_work (pending reconcile work), devices (per-device breakdown). Use -f to select specific fields, -a for all, -h for human-readable sizes.

`bcachefs fs top` Display runtime performance info

Interactive TUI showing live filesystem performance counters: IO rates, btree operations, journal writes, and more. Counters are grouped by category and displayed as a delta/second. Press 'q' to exit, up/down to scroll. See Section 7.1.3 for counter descriptions.

`bcachefs fs timestats` Display bcachefs time statistics

Interactive TUI showing latency and frequency statistics for filesystem operations: reads, writes, btree node allocation, journal flushes, and more. Displays min/max/mean/stddev with quantile breakdowns. Press 'q' to exit, up/down to scroll, 'c' to clear. See Section 7.1.2 for the full list of tracked operations.

5.6 Devices

`bcachefs device add` Add a new device to an existing filesystem

Formats a new device and adds it to a running filesystem. The device is formatted with matching block size and btree node size, then joined to the filesystem. Filesystem options such as --discard and --durability can be set per-device.

`bcachefs device online` Re-add a device to a running filesystem

`bcachefs device offline` Take a device offline, without removing it

`bcachefs device remove` Remove a device from a filesystem

Removes a device from a filesystem. Without --force, removal will fail if the device still has data that hasn't been evacuated—it is always safe to run without --force. Use --force (-f) to allow data loss, or --force-metadata (-F) to also allow metadata loss.

`bcachefs device evacuate` Migrate data off a device

Sets the device state to evacuating and waits for reconcile to move all data off the device. Multiple devices can be evacuated simultaneously. Progress is displayed as remaining data on the device. Once evacuation completes, the device can be removed.

`bcachefs device set-state` Change the state of a device

Device states: rw (read-write, normal operation), ro (read-only, no new allocations), evacuating (reconcile actively moves data off), spare (not used for any data). State changes interact with reconcile: setting a device to evacuating triggers data migration, and setting it to ro prevents new writes while keeping existing data accessible.

Devices are automatically set to ro if too many IO errors are detected, preventing further damage. Use -o to change the state of an offline device.

`bcachefs device resize` Resize the filesystem on a device

Resizes the filesystem on a device, either growing or shrinking to match a new device size. If no size is given, grows to fill the entire device (useful after expanding the underlying block device or partition). Can be run on a mounted filesystem.

`bcachefs device resize-journal` Resize the journal on a device

Resizes the journal on a device. A larger journal allows more write batching and reduces metadata write amplification, improving runtime performance. However, recovery currently scans the journal linearly, so a larger journal means longer mount times after an unclean shutdown.

5.7 Subvolumes and snapshots

`bcachefs subvolume create` (alias: `new`) Create a new subvolume

Creates a new subvolume at the given path. Subvolumes are independently mountable filesystem trees, each with their own inode number space.

`bcachefs subvolume delete` (alias: `del`) Delete an existing subvolume

`bcachefs subvolume snapshot` (alias: `snap`) Create a snapshot of a subvolume

Creates an instant, COW snapshot of a subvolume. Snapshots initially share all data with the source and only consume additional space as either diverges. Use --read-only for a frozen point-in-time copy.

`bcachefs subvolume list` (alias: `ls`) List subvolumes

Lists subvolumes in a bcachefs filesystem. Output includes path, subvolume ID, creation time, flags (ro, unlinked), and cumulative disk usage. Cumulative size is the total space that would be freed if the subvolume and all its snapshots were deleted.

Use --tree for a hierarchical view, --recursive (-R) to include nested subvolumes, --snapshots to include snapshot subvolumes, or --json for machine-readable output. Sort by name, size, or creation time with --sort.

`bcachefs subvolume list-snapshots` (alias: `ls-snap`, `list-snap`) List snapshots and their disk usage

Lists snapshots with disk usage attribution. The default tree view shows the snapshot hierarchy with each node's own disk usage—space consumed exclusively by that snapshot, not shared with ancestors. Use --flat for a tabular view showing both own and cumulative (total) usage per snapshot. Cumulative usage is the sum of a snapshot's own usage plus all ancestors back to the root, representing the total space that would be freed if deleted.

Use --json for machine-readable output including snapshot IDs, parent relationships, and sector counts.

5.8 Filesystem data

Reconcile is the background process that ensures all data and metadata matches configured IO path options: replication count, checksum type, compression, erasure coding, and device targets. Work enters the system when filesystem or inode options change, when devices are added or removed, or when data is written with options that don't match the current configuration.

`bcachefs reconcile status` Show reconcile status

> Shows pending reconcile work broken down by category, with data and metadata sectors remaining for each. Categories: replicas (replication count), checksum (checksum type), compression, erasure_code, target (device/target group placement), stripes (incomplete EC stripes), and high_priority (urgent work such as device evacuation). The pending category tracks work that cannot currently be fulfilled—for example, replicas=3 on a 2-device filesystem—and is automatically retried when filesystem configuration changes (e.g. a device is added).
>
> Scan pending indicates whether reconcile still needs to scan the extents btree for new work. The reconcile thread status section shows whether the thread is actively processing, waiting for IO clock pacing, or idle. While processing, progress indicators show IO wait duration and remaining work.

`bcachefs reconcile wait` Wait for reconcile to finish

> Waits for all pending reconcile work to complete, displaying live progress. In a terminal, shows an interactive TUI with per-category counters updating as work is processed; press 'q' to exit early. In non-interactive mode (pipes, scripts), polls silently and exits when complete.
>
> By default waits on all categories except pending (since pending work may be impossible to complete with current configuration). Use -t to wait on specific categories only.
>
> Note: with erasure coding enabled, there will always be some data in the stripes category from incomplete stripes waiting for foreground writes to fill them. This is normal as long as it stays in the tens-of-megabytes range. Currently this prevents reconcile wait from exiting on EC-enabled filesystems; use -t to wait on specific categories instead.

`bcachefs scrub` Verify checksums and correct errors, if possible

> Reads all data blocks on a running filesystem and verifies their checksums. When a checksum mismatch is found and a valid redundant copy exists (from replication or erasure coding), the corrupted copy is repaired automatically.

5.9 Encryption

`bcachefs unlock` Unlock an encrypted filesystem prior to running/mounting

Loads the encryption key for a bcachefs filesystem into the kernel keyring, allowing subsequent mount or fsck operations without a passphrase prompt. The mount command does this automatically when needed, so unlock is primarily useful for scripted workflows or when running fsck on an encrypted filesystem. Use -c to check whether a device is encrypted without unlocking. Use -f to read the passphrase from a file instead of prompting interactively.

`bcachefs set-passphrase` Change passphrase on an existing (unmounted) filesystem

`bcachefs remove-passphrase` Remove passphrase on an existing (unmounted) filesystem

5.10 Migrate

`bcachefs migrate` Migrate an existing filesystem to bcachefs in place

Converts an existing filesystem to bcachefs in place without copying data. Supported source filesystems: ext2/3/4, XFS, and single-device bcachefs. bcachefs metadata is written into free space alongside the existing data, and a bcachefs superblock is created at a non-default location. Run bcachefs migrate-superblock afterwards to move the superblock to the standard location.

btrfs is not currently supported because its FIEMAP implementation does not report which device an extent resides on.

`bcachefs migrate-superblock` Create default superblock after migrating

After an in-place migration, the bcachefs superblock is at a non-default offset (to avoid overwriting the ext superblock). This command creates a standard superblock at the default location so the filesystem can be mounted normally.

5.11 File options

`bcachefs set-file-option` Set attributes on files in a bcachefs filesystem

Sets per-file or per-directory IO path options, overriding the filesystem-wide defaults. See Section 2.1 for the list of available IO path options. When set on a directory, options are propagated recursively to existing children and inherited by new files.

Changed options take effect immediately for new writes. For existing data, reconcile applies the new options in the background—for example, setting a new compression algorithm will cause existing data to be rewritten with the new algorithm. Use --option=- to remove a specific option, or --remove-all to clear all per-file options.

`bcachefs reflink-option-propagate` Propagate IO options to reflinked extents

Propagates each file's current IO options (compression, checksum, replicas, targets) to its extents, including indirect (reflinked) extents. Reflinked data is shared between files, so propagation is gated by the REFLINK_P_MAY_UPDATE_OPTIONS permission flag on each reflink pointer. When a reflink copy is created, the destination's pointer gets this flag only if the copying user owns the source file—this prevents unprivileged users from altering IO path settings on shared data they do not own.

Old reflink pointers created before the flag was introduced lack it entirely. Use --set-may-update (requires CAP_SYS_ADMIN) to enable the flag on such pointers before propagating.

5.12 Debug

`bcachefs dump` Dump filesystem metadata to a qcow2 image

Dumps filesystem metadata (not data) to a qcow2 sparse image. The resulting file is typically small enough to attach to bug reports, and can be inspected offline with bcachefs list or restored with bcachefs undump. Use -s to sanitize: -s data zeros inline data extents, -s filenames also scrambles directory entry names. Sanitized dumps are safe to share publicly.

`bcachefs undump` Convert qcow2 dump files back to raw device images

`bcachefs list` List filesystem metadata in textual form

Lists btree contents in human-readable text. Operates on unmounted devices in read-only mode. Modes: keys (default) prints key/value pairs, formats shows btree node packing format, nodes shows btree node keys, nodes-ondisk shows the raw on-disk representation.

Use -b to select a btree (default: extents), -s/-e for start/end position, -l for btree depth, -k to filter by key type. With -c, runs fsck before listing. Output is used for debugging filesystem state, verifying btree contents, and inspecting on-disk layout.

`bcachefs list_journal` List filesystem journal entries

Lists journal entries from an unmounted filesystem. The journal is bcachefs's write-ahead log, recording every btree modification as a sequence of keyed entries. Output includes entry headers (sequence number, byte/sector size, version, last_seq, flush status, device locations) and optionally the contained btree keys.

Filtering options narrow output to specific btrees (-b), key ranges (-k), transaction functions (-t), or sequence ranges (-s). Use -D for a compact timeline showing only datetime stamps, -H for headers only, or -L for log-message-only transactions. Blacklisted entries (from aborted journal replays) are hidden by default; use -B to include them.

`bcachefs kill_btree_node` Kill a specific btree node (debugging)

5.13 Miscellaneous

`bcachefs completions` Generate shell completions

`bcachefs version` Display version

6 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. Additionally some of the filesystem wide options can be set via `bcachefs set-fs-option`

6.1 Options reference

`block_size`
fs, format uint default: 4k
Filesystem block size

`btree_node_size`
fs, format uint default: 256k
Btree node size, default 256k

`errors`
fs, format, mount, runtime str default: fix_safe
Action to take on filesystem error

`write_error_timeout`
fs, format, mount, runtime uint default: 30
Number of consecutive write errors allowed before kicking out a device

`metadata_replicas`
fs, format, runtime uint default: 1
Number of metadata replicas (journal and btree)

`data_replicas`
fs, inode, format, runtime uint default: 1
Number of data replicas

`encoded_extent_max`
fs, format uint default: 256k
Maximum size of checksummed/compressed extents

`metadata_checksum`
fs, format, runtime str default: crc32c
Checksum type for metadata writes

`data_checksum`

fs, inode, format, runtime str default: crc32c
Checksum type for data writes

`checksum_err_retry_nr`

fs, format, mount, runtime uint default: 3
Number of read retries on checksum error

`compression`

fs, inode, format, runtime fn default: none
Compression type for data writes

`background_compression`

fs, inode, format, runtime fn default: none
Compression type for background moves

`str_hash`

fs, format, mount, runtime str default: siphash
Hash function for directory entries and xattrs

`metadata_target`

fs, format, runtime fn
Device or label for metadata writes

`foreground_target`

fs, inode, format, runtime fn
Device or label for foreground writes

`background_target`

fs, inode, format, runtime fn
Device or label to move data to in the background

`promote_target`

fs, inode, format, runtime fn
Device or label to promote data to on read

`erasure_code`

fs, inode, format, runtime default: false
Enable erasure coding (DO NOT USE YET)

`casefold`

fs, inode, format default: false
Dirent lookups are casefolded

`casefold_disabled`

fs, mount default: false
Disable casefolding filesystem wide

`inodes_32bit`

fs, inode, format, mount, runtime default: false
Constrain inode numbers to 32 bits

`shard_inode_numbers_bits`

fs, format uint
Shard new inode numbers by CPU id

`gc_reserve_percent`

fs, format, mount, runtime uint default: 8
Percentage of disk space to reserve for copygc

`gc_reserve_bytes`

fs, format, mount, runtime uint
Amount of disk space to reserve for copygc Takes precedence over gc_reserve_percent if set

`root_reserve_percent`

fs, format, mount uint
Percentage of disk space to reserve for superuser

`wide_macs`

fs, format, mount, runtime default: false
Store full 128 bits of cryptographic MACs, instead of 80

`inline_data`

fs, mount, runtime default: true
Enable inline data extents

`promote_whole_extents`

fs, mount, runtime default: true
Promote whole extents, instead of just part being read

`acl`

fs, format, mount default: true
Enable POSIX acls

`usrquota`

fs, format, mount default: false
Enable user quotas

`grpquota`

fs, format, mount default: false
Enable group quotas

`prjquota`

fs, format, mount default: false
Enable project quotas

`degraded`

fs, mount str default: ask
Allow mounting in degraded mode

`mount_trusts_udev`

mount default: true
Trust udev when scanning for member devices

`no_splitbrain_check`

fs, mount default: false
Don't kick drives out when splitbrain detected

`verbose`

fs, mount, runtime default: (varies)
Extra debugging information during mount/recovery

`journal_flush_delay`

fs, mount, runtime uint default: 1000
Delay in milliseconds before automatic journal commits

`journal_flush_disabled`

fs, mount, runtime default: false
Disable journal flush on sync/fsync If enabled, writes can be lost, but only since the last journal write (default 1 second)

`journal_reclaim_delay`
fs, mount, runtime uint default: 100
Delay in milliseconds before automatic journal reclaim

`writeback_timeout`
fs, mount, runtime uint
Delay seconds before writing back dirty data, overriding vm sysctls

`fsck`
fs, mount default: false
Run fsck on mount

`fsck_memory_usage_percent`
fs, mount uint default: 50
Maximum percentage of system ram fsck is allowed to pin

`fix_errors`
fs, mount fn default: exit
Fix errors during fsck without asking

`ratelimit_errors`
fs, mount default: (varies)
Ratelimit error messages during fsck

`nochanges`
fs, mount default: false
Super read only mode - no writes at all will be issued, even if we have to replay the journal

`norecovery`
fs, mount default: false
Exit recovery immediately prior to journal replay

`journal_rewind`
fs, mount uint
Rewind journal

`recovery_passes`
fs, mount bitfield
Recovery passes to run explicitly

`recovery_passes_exclude`
fs, mount bitfield
Recovery passes to exclude

`recovery_pass_last`
fs, mount str
Exit recovery after specified pass

`noexcl`
fs, mount default: false
Don't open device in exclusive mode

`sb`
mount uint default: 8
Sector offset of superblock

`reconstruct_alloc`
fs, mount default: false
Reconstruct alloc btree

`version_upgrade`
: *fs, mount, runtime* str default: compatible
Set superblock to latest version, allowing any new features to be used

`nocow`
: *fs, format, mount, runtime, inode* default: false
Nocow mode: Writes will be done in place when possible. Snapshots and reflink will still caused writes to be COW Implicitly disables data checksumming, compression and encryption

`copygc_enabled`
: *fs, mount, runtime* default: true
Enable copygc: disable for debugging, or to quiet the system when doing performance testing

`reconcile_enabled`
: *fs, mount, runtime* default: true
Enable reconcile: disable for debugging, or to quiet the system when doing performance testing

`reconcile_on_ac_only`
: *fs, mount, runtime* default: false
Enable reconcile while on mains power only

`auto_snapshot_deletion`
: *fs, mount, runtime* default: true
Enable automatic snapshot deletion: disable for debugging, or to quiet the system when doing performance testing

`state`
: *device, runtime* str default: rw
rw,ro,failed,spare

`bucket_size`
: *device* uint
Specifies the bucket size; must be greater than the btree node size

`durability`
: *device, runtime* uint default: 1
Data written to this device will be considered to have already been replicated n times

`data_allowed`
: *device, format* bitfield default: (see description)
Allowed data types for this device: journal, btree, and/or user

`discard`
: *mount, fs, device, runtime* default: true
Enable discard/TRIM support

`rotational`
: *device, runtime* default: false
Disk is rotational; different behaviour for reconcile

`dev_readahead`
: *fs, mount, runtime* uint default: SZ_2M
Per-device readahead window size; summed across all devices to set the filesystem readahead

`ec_stripe_buf_limit`
fs, mount, runtime uint default: 5
Maximum percentage of total RAM for in-flight EC stripe buffers

6.2 File and directory options

A subset of filesystem options can be set on individual files and directories, overriding the filesystem-wide defaults. The per-inode options are: `data_checksum`, `compression`, `background_compression`, `data_replicas`, `foreground_target`, `background_target`, `promote_target`, `metadata_target`, `erasure_code`, `nocow`, `casefold`, and `inodes_32bit`. Note that `casefold` and `inodes_32bit` can only be toggled on empty directories.

Each inode stores its own copy of these options so that reads never need to walk parent directories. Each option has a separate "defined" flag that records whether the option was explicitly set on this inode; options without this flag are propagated from the parent directory on inode creation and rename. 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 set-file-option` command as it will correctly propagate options recursively.

6.3 Inode number sharding

On systems with many CPUs, inode number allocation can become a contention point: every file creation needs a new inode number, and a single global cursor serializes all CPUs. The `shard_inode_numbers_bits` option (0-8, set at format time to the log2 of the number of online CPUs if not specified) partitions the inode number space by CPU ID, giving each CPU its own allocation range. This eliminates cross-CPU contention at the cost of non-sequential inode numbers.

6.4 Error actions

The `errors` option controls how the filesystem responds to metadata inconsistencies. Valid error actions are:

`continue` Log the error but continue normal operation without attempting repair `fix_safe` (default) Automatically repair errors that are safe to fix without user confirmation. Unsafe errors cause emergency read-only with a message to run fsck.

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

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

6.5 Checksum types

Valid checksum types are:

`none`

`crc32c` (default)

`crc64`

`xxhash`

6.6 Compression types

Valid compression types are:

`none` (default)

`lz4`

`gzip`

`zstd`

6.7 String hash types

Valid hash types for string hash tables are:

`crc32c`

`crc64`

`siphash` (default)

7 Debugging tools

7.1 Sysfs interface

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

7.1.1 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.

7.1.2 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.

`btree_node_mem_alloc` Allocate memory in the btree node cache for a new btree node

`btree_node_split` Split a full btree node into two new nodes

`btree_node_compact` Compact a full btree node on disk

`btree_node_merge` Merge two adjacent btree nodes

`btree_node_sort` Sort and resort entire btree nodes in memory, after reading from disk or for compacting

`btree_node_read` Read btree nodes from disk

`btree_node_read_done` Post-read btree node processing

`btree_node_write` Write btree node to disk

`btree_interior_update_foreground` Foreground time for topology-changing btree updates (splits, compactions, merges); roughly corresponds to lock held time

`btree_interior_update_total` Total time for topology-changing btree updates, including background transaction phase after new nodes are written

`btree_write_buffer_flush` Flush btree write buffer to btree

`btree_gc` GC pass recalculating oldest generation numbers

`data_write` Core write path: allocate space, compress, encrypt, checksum, issue writes, update extents btree

`data_read` Core read path: look up extents btree, issue reads, checksum, decompress, decrypt

`data_promote` Promote: write a cached copy of an extent to promote_target on read

`journal_flush_write` Flush journal writes: cache flush to devices then FUA journal writes `journal_noflush_write` Non-flush journal writes, without cache flushes or FUA

`journal_flush_seq` Flush a journal sequence number to disk for sync, fsync, and bucket reuse

`journal_pin_flush_btree` Flush btree journal pins

`journal_pin_flush_key_cache` Flush key cache journal pins

`journal_pin_flush_other` Flush other journal pins

`blocked_journal_low_on_space` Blocked: journal reclaim not keeping up with reclaiming space

`blocked_journal_low_on_pin` Blocked: journal pins (dirty btree nodes, key cache entries) not flushed fast enough

`blocked_journal_max_in_flight` Blocked: too many journal writes in flight

`blocked_journal_max_open` Blocked: too many journal entries open, not yet closed for writing

`blocked_journal_write_buffer_flush` Blocked: waiting for write buffer flush

`blocked_key_cache_flush` Blocked: waiting for key cache flush

`blocked_allocate` Blocked: bucket allocation waiting, copygc or allocator thread not keeping up

`blocked_allocate_open_bucket` Blocked: all open bucket handles in use

`blocked_write_buffer_full` Blocked: write buffer full

`blocked_writeback_throttle` Blocked: writeback throttle

`nocow_lock_contended` Nocow lock contention

7.1.3 Persistent counters

bcachefs tracks persistent counters in the superblock that survive across mounts. These are available in `/sys/fs/bcachefs/<uuid>/counters/` and count either events or sectors depending on type.

`sync_fs` Filesystem sync operations

`fsync` Fsync operations

`data_read` Sectors read from disk

`data_read_inline` Sectors read from inline data extents `data_read_hole` Sectors read as holes (zero-filled)

`data_read_promote` Sectors promoted to cache on read

`data_read_nopromote` Reads not promoted

`data_read_nopromote_may_not` Reads not promoted: not eligible

`data_read_nopromote_already_promoted` Reads not promoted: already cached

`data_read_nopromote_unwritten` Reads not promoted: unwritten extent

`data_read_nopromote_congested` Reads not promoted: device congested

`data_read_bounce` Reads requiring bounce buffer

`data_read_split` Reads split across multiple extents

`data_read_reuse_race` Read bio reuse races

`data_read_retry` Read retries

`data_read_fail_and_poison` Read failures with poisoned pages

`data_read_narrow_crcs` CRC entries narrowed on read

`data_read_narrow_crcs_fail` CRC narrowing failures on read

`data_write` Sectors written to disk

`data_update_pred` Sectors predicted for data update

`data_update` Sectors moved by data update (reconcile, copygc)

`data_update_no_io` Sectors updated without IO (key update only)

`data_update_in_flight` Data updates currently in flight

`data_update_fail` Failed data update sectors

`data_update_read` Sectors read for data update

`data_update_write` Sectors written for data update

`data_update_key` Sectors where btree key was updated

`data_update_key_fail` Failed btree key update sectors

`data_update_useless_write_fail` Useless data update write failures

`data_update_start_fail_obsolete` Obsolete: data update start failures

`data_update_noop_obsolete` Obsolete: no-op data updates `reconcile_scan_fs` Sectors scanned for filesystem reconcile

`reconcile_scan_metadata` Sectors scanned for metadata reconcile

`reconcile_scan_pending` Sectors scanned for pending reconcile

`reconcile_scan_device` Sectors scanned for device reconcile

`reconcile_scan_inum` Sectors scanned for inode reconcile

`reconcile_clear_scan` Reconcile scan entries cleared

`reconcile_btree` Btree sectors reconciled

`reconcile_data` Data sectors reconciled

`reconcile_phys` Physical sectors reconciled

`reconcile_stripe` Stripe sectors reconciled

`reconcile_set_pending` Sectors marked as pending reconcile

`evacuate_bucket` Buckets evacuated by copygc

`stripe_alloc` Stripe allocation attempts

`stripe_create` Stripes created

`stripe_reuse` Stripes reused

`stripe_create_fail` Stripe creation failures

`stripe_delete` Stripes deleted

`stripe_update_bucket` Stripe bucket updates

`stripe_update_extent` Stripe extent updates

`stripe_update_extent_fail` Stripe extent update failures

`copygc` Copygc runs

`copygc_wait_obsolete` Obsolete: copygc waits

`cached_ptr_drop` Cached pointer sectors dropped

`bucket_invalidate` Buckets invalidated

`bucket_discard_worker` Discard worker invocations

`bucket_discard_fast_worker` Fast discard worker invocations

`bucket_discard` Bucket discards issued

`bucket_discard_fast` Fast bucket discards issued `bucket_alloc` Bucket allocations

`bucket_alloc_fail` Bucket allocation failures

`open_bucket_alloc_fail` Open bucket allocation failures

`bucket_alloc_from_stripe` Buckets allocated from existing stripe

`sectors_alloc` Total sectors allocated

`bkey_pack_pos_fail` Bkey position packing failures

`btree_cache_scan` Btree cache scan operations

`btree_cache_reap` Btree nodes reaped from cache

`btree_cache_cannibalize` Btree cache cannibalize operations

`btree_cache_cannibalize_lock` Btree cache cannibalize lock acquisitions

`btree_cache_cannibalize_lock_fail` Btree cache cannibalize lock failures

`btree_cache_cannibalize_unlock` Btree cache cannibalize lock releases

`btree_node_write` Btree node writes

`btree_node_read` Btree node reads

`btree_node_compact` Btree node compactions

`btree_node_merge` Btree node merges

`btree_node_merge_attempt` Btree node merge attempts

`btree_node_split` Btree node splits

`btree_node_rewrite` Btree node rewrites

`btree_node_alloc` Btree nodes allocated

`btree_node_free` Btree nodes freed

`btree_node_set_root` Btree root changes

`btree_key_cache_fill` Btree key cache fills

`btree_path_relock_fail` Btree path relock failures

`btree_path_upgrade_fail` Btree path lock upgrade failures

`btree_reserve_get_fail` Btree reservation failures

`journal_res_get_blocked` Journal reservation blocked

`journal_full` Journal full events

`journal_reclaim_finish` Journal reclaim completions

`journal_reclaim_start` Journal reclaim starts

`journal_write` Journal writes

`gc_gens_end` GC generation pass completions

`gc_gens_start` GC generation pass starts

`trans_blocked_journal_reclaim` Transactions blocked on journal reclaim

`trans_restart_btree_node_reused` Transaction restart: btree node reused

`trans_restart_btree_node_split` Transaction restart: btree node split

`trans_restart_fault_inject` Transaction restart: fault injection

`trans_restart_iter_upgrade` Transaction restart: iterator lock upgrade

`trans_restart_journal_preres_get` Transaction restart: journal pre-reservation

`trans_restart_journal_reclaim` Transaction restart: journal reclaim

`trans_restart_journal_res_get` Transaction restart: journal reservation

`trans_restart_key_cache_key_realloced` Transaction restart: key cache key reallocated

`trans_restart_key_cache_raced` Transaction restart: key cache race

`trans_restart_mark_replicas` Transaction restart: mark replicas

`trans_restart_mem_realloced` Transaction restart: memory reallocated

`trans_restart_memory_allocation_failure` Transaction restart: memory allocation failure

`trans_restart_relock` Transaction restart: relock

`trans_restart_relock_after_fill` Transaction restart: relock after fill

`trans_restart_relock_key_cache_fill_obsolete` Obsolete: transaction restart relock key cache fill

`trans_restart_relock_next_node` Transaction restart: relock next node

`trans_restart_relock_parent_for_fill_obsolete` Obsolete: transaction restart relock parent for fill

`trans_restart_relock_path` Transaction restart: relock path

`trans_restart_relock_path_intent` Transaction restart: relock path intent `trans_restart_too_many_iters` Transaction restart: too many iterators

`trans_restart_traverse` Transaction restart: traverse

`trans_restart_upgrade` Transaction restart: lock upgrade

`trans_restart_would_deadlock` Transaction restart: would deadlock

`trans_restart_would_deadlock_write` Transaction restart: would deadlock on write

`trans_restart_injected` Transaction restart: injected for testing

`trans_restart_key_cache_upgrade` Transaction restart: key cache lock upgrade

`trans_traverse_all` Full transaction path traversals

`transaction_commit` Transaction commits

`write_super` Superblock writes

`trans_restart_would_deadlock_recursion_limit` Transaction restart: deadlock recursion limit

`trans_restart_write_buffer_flush` Transaction restart: write buffer flush

`trans_restart_split_race` Transaction restart: split race

`write_buffer_flush` Write buffer flushes

`write_buffer_flush_slowpath` Write buffer flushes via slow path

`write_buffer_flush_sync` Synchronous write buffer flushes

`write_buffer_maybe_flush` Write buffer conditional flush checks

`accounting_key_to_wb_slowpath` Accounting key to write buffer slow path

`error_throw` Errors thrown

7.1.4 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.

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

`journal_debug`
  Prints a variety of internal journal state.

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

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

`trigger_journal_flush`
  Echoing to this file flushes all journal pins (forcing dirty btree nodes and key cache entries to be written) and then issues a journal meta write.

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

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

7.1.5 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`.

7.2 Debugfs interface

Various internal debugging files are available under `/sys/kernel/debug/bcachefs/<uuid>/`.

The `btrees/` subdirectory contains one directory per btree (e.g. `btrees/extents/`, `btrees/inodes/`), each with three files:

`keys`
  Entire btree contents, one key per line.

`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.

`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.

The remaining files in the debugfs root are:

`btree_transactions`

Lists each running btree transaction 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_pins`

Lists what is preventing each journal entry from being reclaimed: dirty btree nodes that still reference that sequence number, key cache entries waiting to be flushed, or in-progress operations that need the entry to remain available for crash recovery. Useful for diagnosing journal space exhaustion.

`btree_deadlock`

Checks for and reports btree lock deadlock cycles among transactions.

`cached_btree_nodes`

Lists btree nodes currently in the btree node cache.

`btree_transaction_stats`

Per-transaction-function statistics: lock hold times, restart counts, and maximum memory usage.

`btree_node_scan`

Results from the last scan for btree nodes on raw devices, used during catastrophic recovery.

`write_points`

Lists active write points and the buckets they are currently writing to.

7.3 Listing and dumping filesystem metadata

7.3.1 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.

7.3.2 bcachefs list

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

7.3.3 bcachefs list_journal

This subcommand lists the contents of the journal for offline filesystems. The journal is a complete record of every btree update, accounting delta, log message, and timestamp, ordered by sequence number. Each entry can be traced back to the transaction function that created it (when `journal_transaction_names` is enabled). Filtering by btree, sequence range, or transaction name makes it possible to reconstruct the exact sequence of events leading to a problem - and to identify the right sequence number for journal rewind.

7.3.4 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`.

8 Subsystem details

8.1 Data paths

This section describes the end-to-end flow of read and write operations through the filesystem.

8.1.1 Write path

Writes go through a pipeline of optional transformations: encryption (ChaCha20), compression (lz4/zstd/gzip), and checksumming, applied in that order. The data is then written to each replica device and the extent metadata is updated in the btree atomically. If encryption or compression are not enabled, those stages are skipped entirely.

Write point selection determines which device(s) receive the data and how IO from different sources is segregated into separate buckets — see the foreground allocator documentation.

Direct IO can bypass internal buffering when no transformations are needed and the user buffer is properly aligned, avoiding an extra memory copy.

The normal (COW) write path allocates new disk space, encodes the data (encryption, then compression, then checksumming), writes it to each replica device, and inserts a new extent key into the btree. Because writes always go to new locations, the old data remains intact until the btree update commits — there is no window where a crash can leave partially-written data.

Multiple write points are used, selected by hashing the process ID, to segregate unrelated data and help prevent fragmentation.

Writes with checksumming or compression enabled must bounce the data through a temporary buffer for checksum stability (the kernel cannot guarantee that a user buffer won't be modified in flight). This is the main per-write overhead of encoded extents.

The `data_write` time stat tracks end-to-end write latency (from submission to btree update). The `data_write` persistent counter tracks total sectors written.

8.1.1.1 Nocow writes

The nocow write path overwrites data in place, bypassing the encoded-extent pipeline entirely: no checksumming, no compression, no encryption, no COW. This eliminates write amplification and bounce buffer overhead at the cost of data integrity features.

Nocow writes require per-bucket locking to avoid racing with the move path. This is normally invisible, but contention can appear under heavy concurrent nocow writes; the `nocow_lock_contended` time stat tracks this (see Debugging tools). When snapshots or reflinks create shared extents, even nocow files fall back to COW for those extents.

Because nocow writes are not checksummed, they cannot be verified by scrub or self-healed from replicas. On encrypted filesystems, nocow data is stored in plaintext. The option is primarily useful for database and VM workloads that manage their own data integrity and need stable disk offsets or minimal write amplification.

8.1.2 Read path

Reads are transparent and self-healing: if a checksum failure or IO error occurs on one replica, bcachefs automatically retries from another replica. The failed device's error counter is incremented and the bad copy is rewritten from the good one. If all replicas fail, the error is propagated to the application.

With multiple devices, reads go to the lowest-latency replica. This is tracked per-device and adapts over time, so mixed SSD/HDD configurations automatically prefer the SSD for reads without explicit configuration.

End-to-end flow: extent lookup, device selection, disk read, checksum verification, decompression, decryption. For compressed or checksummed extents the full extent must be read even for partial requests, because checksums and compression operate on the whole extent.

The read path looks up the extent covering the requested range and reads the data from disk. With multiple replicas, reads stripe across replicas, preferring the one with the lowest current IO latency. For encoded extents, the entire extent must be read even if only a portion was requested, because the checksum covers the full extent and decompression requires the complete input. This per-extent granularity gives much better compression ratios and much smaller metadata (fewer checksums to store) than block-granular approaches. Buffered IO automatically reads entire extents into the page cache, so the only real downside is to small-block random read performance that doesn't fit in cache — a workload that is rare outside of benchmarks. Block-granular checksums may be added as an option in the future if there is user demand.

8.1.2.1 Error handling When a checksum mismatch is detected, the same replica is first re-read up to `checksum_err_retry_nr` times (default 3) to handle transient errors such as bitflips during bus transfer. If retries do not produce a good read and another replica exists, the read is retried from that replica. On successful retry, the failed replica is immediately repaired by rewriting it from the good copy — this is self-healing, and it happens transparently on every read. If erasure coding is available, the missing data can be reconstructed from parity even with no good replica. If no valid copy can be obtained, the read returns an IO error to userspace.

When an extent with a checksum error must be moved (e.g. by copygc or reconcile), the move path recomputes a checksum for the corrupted data so it can be written to the new location, but marks the extent as *poisoned*. Poisoned extents are tracked by the `BCH_EXTENT_FLAG_poisoned` flag: the data is known bad, but the filesystem can still operate on it (move it, account for it). Reads of poisoned extents return an error rather than silently serving corrupt data.

`KEY_TYPE_error` extents represent ranges where data has been permanently lost — for example, after a force device removal that left extents with no remaining replicas. Reads to these ranges return IO errors. These error keys are visible in `bcachefs list` output and can help diagnose which files were affected by data loss.

The `data_read` time stat tracks end-to-end read latency. The `data_read` counter tracks total sectors read; `data_read_bounce` counts reads that required a bounce buffer (encoded extents), and `data_read_retry` counts reads retried due to checksum failure or stale pointers.

8.1.2.2 Promote (caching) When `promote_target` is set, the read path can copy data from a slow device to a fast device on read. This is how bcachefs implements tiered caching: reads that hit a slow tier (e.g. HDD) are transparently promoted to a fast tier (e.g. SSD) so that subsequent reads are served from the faster device.

Promotion is opportunistic: it is skipped if the data already has a copy on the promote target, if the target is congested, or if the per-CPU promote semaphore is exhausted. Promoted copies are written as cached pointers, so they can be evicted under space pressure without data loss.

Relevant counters and time stats:

- `data_read_promote` — sectors promoted
- `nopromote_already_promoted`, `nopromote_congested`, `nopromote_unwritten` — reasons promotion was skipped
- `data_promote` — promotion write latency

8.1.3 Data structures

An extent's value (`struct bch_extent`) is a variable-length array of typed entries, each self-describing via a type field encoded in the low bits of the first word (a scheme similar to UTF-8: the position of the first set bit determines the type). The entries are defined by `union bch_extent_entry` and can appear in any order, with one rule: a CRC entry applies to all pointers that follow it until the next CRC entry.

8.1.3.1 Extent pointers (`struct bch_extent_ptr`)

Each pointer is the "where is the data" record: a device number, a 44-bit sector offset (supporting up to 8 PiB per device), and a generation number that must match the bucket's current generation to be valid (stale pointers are detected and dropped during reads). Flags distinguish cached pointers (evictable copies on a faster tier) from dirty pointers, and mark unwritten reservations.

8.1.3.2 CRC entries (`bch_extent_crc32/64/128`)

CRC entries carry the checksum, compression type, and the geometry needed to handle partially-overwritten extents: `compressed_size` and `uncompressed_size` record the original extent dimensions, and `offset` records how far into the uncompressed data the currently live region starts (the live region's size is in `bkey.size`).

Three variants exist as a space optimization. Most extents need only a `crc32` (8 bytes): it supports extents up to 128 sectors with a 32-bit checksum and no nonce. `crc64` (16 bytes) extends this to 512 sectors, adds a 10-bit nonce for encryption, and carries an 80-bit checksum. `crc128` (24 bytes) is the full-size form: 8192-sector extents, a 13-bit nonce, and a 128-bit checksum — required when encryption is enabled, since the ChaCha20/Poly1305 MAC is 128 bits. The write path picks the smallest variant that can represent the extent's parameters; the read path unpacks all three into a common `bch_extent_crc_unpacked` for uniform handling.

A CRC entry applies to every pointer after it until the next CRC entry. Initially all replicas share one CRC, but copygc or tiering may rewrite a single replica (possibly trimming it), producing a new CRC for just that pointer. This is why extents can contain multiple CRC entries.

8.1.3.3 Stripe pointer (`struct bch_extent_stripe_ptr`)

Links an extent to an erasure-coding stripe. The `idx` field identifies the stripe, and `block` identifies which block within the stripe this extent occupies. When a read fails, the EC subsystem can reconstruct the data from the stripe's parity blocks.

8.1.3.4 Flags entry (`struct bch_extent_flags`)

A bitfield of per-extent flags. Currently the only flag is `poisoned`: the extent contains data known to be corrupt (e.g. it failed checksum verification and could not be repaired). Poisoned extents are kept rather than discarded so that the filesystem can still account for them and move them, but reads return an error.

8.1.3.5 Reconcile entry (`struct bch_extent_reconcile`)

Embeds IO options (target, compression, replicas, checksum type, erasure coding) directly in the extent. This exists primarily for reflink indirect extents: since an indirect extent may be referenced by many inodes, there is no single "owning" inode to look up IO options from. The reconcile entry records what the extent's options *should* be so that background reconciliation can bring them into compliance.

8.1.3.6 Composition A typical extent value is a sequence of these entries. Some examples:

- Unchecksummed, single replica: `[ptr]` — just one pointer, no CRC. The pointer's offset is adjusted directly when the extent is trimmed.
- Checksummed, 2 replicas: `[crc32, ptr, ptr]` — one CRC covers both pointers (same data was written to both locations).
- After partial copygc of one replica: `[crc32, ptr, crc32, ptr]` — the second pointer was rewritten to a new location covering only the live portion, so it gets its own CRC with different size/offset fields.
- EC extent with encryption: `[crc128, ptr, stripe_ptr]` — the 128-bit CRC is required for the encryption MAC, and the stripe pointer links to the parity stripe.
- Reflink indirect extent: `[crc32, ptr, ptr, reconcile]` — the reconcile entry records the desired IO options for background processing.

8.1.4 Encryption

bcachefs uses authenticated encryption (AEAD) with ChaCha20/Poly1305. Unlike block-layer encryption (AES-XTS), which operates on fixed blocks with no room for nonces or MACs, bcachefs stores a nonce and cryptographic MAC alongside every data pointer, creating a chain of trust from the superblock down to individual extents: any modification, deletion, reordering, or rollback of metadata is detectable. Encryption is all-or-nothing at the filesystem level and can only be enabled at format time.

8.1.4.1 Key hierarchy The key hierarchy has three levels:

1. **Passphrase**: User-supplied, never stored on disk. Fed to the scrypt KDF (parameters stored in the superblock's `bch_sb_field_crypt`) to derive a 256-bit passphrase key. The KDF runs entirely in userspace, so alternative key sources (hardware tokens, key files) can be integrated without kernel changes.

2. **Master key**: A random 256-bit key generated at format time, stored in the superblock encrypted by the passphrase key. A magic value (`BCH_KEY_MAGIC`) stored alongside the encrypted master key allows verification of a correct passphrase without trial decryption of filesystem data. Changing the passphrase re-encrypts only the master key, not any filesystem data.

3. **Per-extent nonces**: Each extent is encrypted with the master key and a 128-bit nonce composed of the extent's 96-bit version number, compression type, and uncompressed size, combined with a per-CRC nonce offset. Data encryption uses the `BCH_NONCE_EXTENT` domain separator; the Poly1305 MAC key uses `BCH_NONCE_POLY`.

There is currently no key rotation mechanism: the master key is fixed for the lifetime of the filesystem. Key escrow, multi-passphrase unlock, and hardware key (TPM, FIDO2) support are not implemented.

8.1.4.2 Kernel keyring integration The kernel never sees the passphrase. Instead, userspace derives the passphrase key via scrypt and adds it to the Linux kernel keyring as a `user` type key with description `bcachefs:<UUID>`. At mount time, the kernel calls `request_key()` to find this key, uses it to decrypt the master key from the superblock, and caches the decrypted master key in kernel memory for the lifetime of the mount.

This design inherits the well-known pain points of the Linux keyring subsystem:

- **Session isolation**: Keys added to a session keyring are not visible from other sessions of the same user. An `ssh` session that runs `bcachefs unlock` does not make the key available to a different terminal, to systemd mount units, or to cron jobs. The key must be added to `KEY_SPEC_USER_KEYRING` (the per-UID keyring) to be visible across sessions, but this is not always the default.

- **Privilege boundaries**: `sudo mount` uses root's keyring, not the calling user's. Systemd units run in isolated session contexts. The key must be explicitly placed in a keyring that the mounting process can access.

8.1.4.3 MAC storage

The Poly1305 MAC is stored in the extent's CRC entry. By default, the MAC is truncated to 80 bits (`chacha20_poly1305_80`), which is sufficient for most threat models. The `wide_macs` option stores the full 128-bit MAC at the cost of 8 bytes per extent, and is recommended when the storage device itself is untrusted (e.g. USB drives, network storage) and an attacker can make repeated forgery attempts or perform rollback attacks. Metadata always uses 128-bit MACs regardless of the `wide_macs` setting.

8.1.4.4 Nonce reuse with external snapshots

AEAD algorithms require that a (key, nonce) pair is never reused for different plaintexts. bcachefs derives extent nonces from the extent's version number, which is unique within a single filesystem instance. However, if the underlying storage is snapshotted externally (LVM, ZFS zvol, VM snapshot, loop device on a reflinked file) and the snapshot is mounted read-write, both instances share the same master key and will derive the same nonces for new writes to the same logical locations. This breaks ChaCha20's semantic security.

bcachefs's own snapshot mechanism does not have this problem: internal snapshots share extents via reflinks with COW semantics, and new writes get new version numbers and therefore new nonces.

Mitigation: Never mount an external snapshot of an encrypted volume read-write — keep external snapshots read-only (`-o nochanges`). Alternatively, place LUKS between the snapshot layer and bcachefs (e.g. LVM → LUKS → bcachefs).

8.1.5 Erasure coding

Erasure coding uses Reed-Solomon parity (the same algorithm as RAID-5/6) to provide redundancy at lower storage cost than full replication. It is enabled per-inode via the `erasure_code` option and uses the `data_replicas` setting to determine parity count: `data_replicas=2` gives one parity block (RAID-5), `data_replicas=3` gives two (RAID-6).

8.1.5.1 Write path

Writes are initially replicated: one copy goes to a bucket queued for a new stripe, and an extra replica provides immediate durability. As full stripes accumulate, P/Q parity is written out and the extra replicas are dropped. This gives us erasure coding with no write hole and no fragmentation of writes — data is written out in the ideal layout, and since stripes are written once and never updated in place, parity is always consistent.

The extra replicas are cheap. Since device write caches are only flushed on journal commit (i.e. fsync), the allocator can return the extra-replica buckets to the write point for reuse as soon as the stripe commits. In bandwidth-heavy workloads with nothing doing fsyncs, the extra replicas can be overwritten while still in the device writeback cache — they only cost bus bandwidth, not real disk writes.

The allocator segregates EC and non-EC writes at the open-bucket level: a write requesting EC will only be placed in a bucket already tagged for stripe membership, and non-EC writes will never use such buckets. This means a bug in stripe creation, parity computation, or extent updating is structurally scoped to EC-enabled data: non-EC extents never carry `stripe_ptr` entries and are never read through EC reconstruction paths. Btree nodes are never placed in EC buckets; this is explicitly checked and flagged as a filesystem inconsistency.

If stripe creation fails partway (e.g. a crash between writing parity and updating extents), the extra replicas from the staging phase remain valid; the reconcile subsystem detects the incomplete state and retries. Changing the `erasure_code` option at runtime triggers reconcile to add or remove EC protection on existing data.

8.1.5.2 Stripe layout Each block in a stripe is one bucket on one device. Stripe width is determined dynamically: all eligible devices in the target group are used, up to a maximum of 16 blocks per stripe. Eligible devices must be read-write, have nonzero durability, and share the same bucket size (the most common bucket size among candidates is chosen; devices with a different bucket size are excluded). The minimum is `redundancy + 2` devices (3 for RAID-5, 4 for RAID-6). With n eligible devices, a stripe has $n -$ redundancy data blocks and `redundancy` parity blocks, maximizing storage efficiency. There is no configuration to limit stripe width to a subset of available devices.

Stripe fragmentation is tracked in the LRU btree. When all data blocks in a stripe become empty (sector count zero), the stripe is automatically deleted. Partially empty stripes are candidates for reuse: new stripe creation scans the fragmentation LRU for a stripe with matching parameters (same disk label, algorithm, and redundancy), copies the non-empty blocks into the new stripe, and fills the remaining slots with fresh data. This consolidation recovers space without a full copygc pass.

8.1.5.3 On-disk representation A stripe is stored as a `bch_stripe` key in the stripes btree (ID 6), keyed by stripe index. The fixed-size header contains:

- `sectors` — bucket size (all blocks in a stripe share the same bucket size)
- `algorithm` — Reed-Solomon variant (4 bits)
- `nr_blocks` — total blocks (data + parity)
- `nr_redundant` — number of parity blocks

- csum_type, csum_granularity_bits — checksum algorithm and block granularity for per-block checksums
- disk_label — target disk label (8 bits; a limitation noted for a future stripe_v2)
- needs_reconcile — flag indicating the stripe needs reconcile processing (e.g. after partial creation or option change)

After the header, three variable-length sections are packed in order: nr_blocks bch_extent_ptr entries (one per block, giving the device, offset, and generation for each bucket); a 2D array of checksums indexed by [block][csum_block] where the checksum block size is $2^{\texttt{csum\_granularity\_bits}}$ sectors; and nr_blocks __le16 sector counts tracking how many sectors of live data each block contains (used for fragmentation tracking and stripe deletion).

Each data extent that belongs to a stripe carries an inline bch_extent_stripe_ptr entry with three fields: idx (47-bit stripe index into the stripes btree), block (8-bit block number within the stripe), and redundancy (4-bit copy of the stripe's nr_redundant, so the read path knows the parity level without looking up the stripe).

Several auxiliary btrees support EC operations: the bucket_to_stripe btree (ID 26) maps each stripe-member bucket to the stripes referencing it, enabling the allocator and copygc to know when a bucket is part of a stripe; the stripe_backpointers btree (ID 27) stores backpointers indexed by stripe pointer for data on invalid or removed devices, enabling stripe repair without the original device; and the alloc btree tracks per-bucket stripe_refcount and stripe_sectors separately from dirty_sectors. Backpointers for stripe blocks point back to the stripes btree rather than the extents btree.

8.1.5.4 Reconstruction reads EC reconstruction reads happen when a device is offline or a checksum mismatch is detected: the read path fetches the remaining data blocks plus parity and reconstructs the missing block using the Reed-Solomon algorithm.

8.1.5.5 Consistency and self-healing Stripe triggers validate bucket accounting on every stripe insert or delete: parity bucket refcounts and dirty sector counts must be consistent. When a mismatch is detected, the bucket is protected from reuse (refcount held at a safe value) and the check_allocations recovery pass is automatically scheduled to perform a full repair. Stale pointers detected during reconstruction reads similarly trigger recovery.

If stripe creation fails partway (e.g. crash between writing parity and updating extents), the extra replicas from the staging phase remain valid data, and the reconcile subsystem detects the incomplete state and retries stripe creation.

8.1.6 Reflink

Reflink (cp -reflink, FICLONE ioctl) creates copies that share underlying storage. The original extent is moved to the reflink btree with a reference count, and a lightweight pointer (KEY_TYPE_reflink_p) is left in the extents btree. Reads through a reflink pointer require two btree lookups instead of one: first the reflink_p, then the actual data pointers in the reflink btree.

8.1.6.1 On-disk representation In the extents btree, a KEY_TYPE_reflink_p replaces the original extent. It contains an index (REFLINK_P_IDX, 56 bits) pointing into the reflink btree (ID 7), plus front_pad and back_pad fields. In the reflink btree, a KEY_TYPE_reflink_v stores the actual data pointers, CRCs, and compression metadata (identical to a regular KEY_TYPE_extent) preceded by a 64-bit reference count.

The pad fields exist because copygc or reconcile may split an indirect extent into fragments. Without the pads, fragments outside the pointer's nominal range would have their refcounts leaked. The pads remember the full range originally referenced so that triggers walk all fragments when updating refcounts. If the indirect extent is missing in the live data range (e.g. due to corruption), fsck sets the REFLINK_P_ERROR flag on the pointer; gaps only in the padded region adjust the pad instead.

8.1.6.2 Creation and lifecycle When cp -reflink (or the FICLONE ioctl) creates a reflink, the source extent is converted in place. A new KEY_TYPE_reflink_v is allocated at the end of the reflink btree (by seeking to POS_MAX), containing the original data pointers and a refcount initialized to zero. The source extent is replaced with a KEY_TYPE_reflink_p. If the source is already a reflink_p, no conversion is needed. A new reflink_p is then created in the destination file; btree triggers on the inserts increment the refcount.

On insertion or deletion of a reflink_p, the trigger walks the full referenced range (including pad) in the reflink btree and increments or decrements the refcount on each overlapping reflink_v fragment, expanding the pads if the indirect extent is larger than expected (due to a prior split). Writing new data over a reflink_p requires no special logic: the normal btree update inserts a regular KEY_TYPE_extent, the overwrite trigger decrements the refcount, and when a reflink_v's refcount reaches zero, its trigger converts the key to KEY_TYPE_deleted, cascading through the normal extent trigger to free disk space and remove backpointers.

Reflink is currently a one-way transformation: once an extent becomes indirect, it never converts back, even when the refcount drops to 1. The reflink_v trigger fires at refcount 0 to delete the indirect extent, but does not de-indirect at refcount 1 because the trigger would need to walk transaction updates to find the sole remaining reflink_p, and operations like fcollapse and finsert can cause transient refcount fluctuations ($1 \to 0 \to 1$) within a single transaction as extents are moved around. With IO option propagation, de-indirecting at refcount 1 is becoming a more pressing concern, since a lone indirect extent with one reference still pays the cost of an extra btree lookup on every read. Additionally, `reflink_p` keys are not merged during btree compaction because a merged pointer could span an unbounded number of `reflink_v` fragments; merging requires triggers to walk pending transaction updates and diff overlapping `reflink_p` ranges.

8.1.6.3 IO option propagation The `reflink_p` carries a `REFLINK_P_MAY_UPDATE_OPTIONS` flag that controls whether IO path options (compression, checksum type, replicas, targets) may propagate from the referencing file to the shared indirect extent. This is a security boundary: a reflink copy of data owned by another user must not allow the copier to decrease replicas or change checksum settings on data they do not own. At creation time, the source file's `reflink_p` gets this flag set, but the destination's does not (the VFS layer does not yet pass down the permission context needed to determine whether the copier has write access to the source).

A `reflink_v` has no backpointer to its owning inode, so it cannot look up per-inode IO options at read time. Instead, the indirect extent embeds a `bch_extent_reconcile` entry that stores the desired IO options alongside `*_from_inode` flags recording which options came from a per-inode setting rather than the filesystem default. At creation time no reconcile entry is added; the data is simply copied verbatim from the source extent.

When reconcile scans the extents btree and encounters a `reflink_p` with `REFLINK_P_MAY_UPDATE_OPTIONS` set, it follows through to the corresponding `reflink_v` keys in the reflink btree and updates their embedded reconcile entries with the referencing inode's current options. If the on-disk data does not match (e.g. the inode now requests zstd compression but the data is uncompressed), the reconcile entry's `need_rb` bits are set and the data is scheduled for background rewrite. Without the flag, reconcile does not propagate that `reflink_p`'s inode options to the indirect extent.

The `reflink_v` can only hold one set of IO options at a time. Since only the source file's `reflink_p` currently gets the `MAY_UPDATE_OPTIONS` flag, there is no conflict when multiple files reference the same indirect extent: the source file's options take precedence, and other referencing files cannot influence the indirect extent's IO path behavior. The read path always uses the CRC and compression metadata stored in the `reflink_v`'s extent entries (reflecting how the data was actually written), regardless of the referencing file's current options; the referencing inode's options only affect promote decisions.

8.1.6.4 Interaction with snapshots The reflink btree is not snapshot-aware: `reflink_v` keys are shared across all snapshots. The `reflink_p` keys in the extents btree are snapshot-aware, so when a file is snapshotted both subvolumes see the same `reflink_p` keys through normal snapshot visibility. Writing to either subvolume creates a new extent in that snapshot and decrements the shared refcount; the other snapshot's `reflink_p` is unchanged.

8.1.6.5 Consistency and self-healing

When the read path follows a `reflink_p` and discovers the corresponding `reflink_v` is missing or partially missing, the reference is repaired in-place: `front_pad` and `back_pad` are adjusted to trim the reference to the valid range, and the `REFLINK_P_ERROR` flag is set if the missing range overlaps actual data. If a previously-errored indirect extent reappears (e.g. after btree node recovery), the error flag is cleared automatically.

The `check_indirect_extents` recovery pass walks the reflink btree, validates extent sizes, and drops stale device pointers (generation mismatches). This pass can run online.

8.1.7 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 `min(blocksize/2, 1024)` bytes, it will be written as an inline data extent. Inline data extents can also be reflinked: the inline data is moved to the reflink btree as a `KEY_TYPE_indirect_inline_data` (which carries a refcount and the inline data bytes) and a `KEY_TYPE_reflink_p` is left in the extents btree, following the same mechanics as regular extent reflinks.

8.1.8 Move path

The move path is the shared IO engine behind copygc, reconcile, and device evacuation. It reads extents via the normal read path, writes them to a new location, and atomically updates pointers.

Background move IO is throttled by two runtime-tunable options, both adjustable via sysfs:

- `move_bytes_in_flight` (default 64 MB) — total bytes of outstanding move IO
- `move_ios_in_flight` (default 64) — number of outstanding requests

Individual consumers can be disabled: `copygc_enabled`, `reconcile_enabled`, and `reconcile_on_ac_only` (pauses reconcile on battery power).

8.1.9 Reconcile

The reconcile subsystem ensures that all data and metadata is stored correctly according to configured IO path options. It continuously monitors for mismatches between how data is actually stored and how it should be stored — whether caused by option changes, device additions or removals, degraded replicas, or any other reason — and rewrites affected extents via the move path.

If reconcile detects an inconsistency without an obvious cause (no option change, no device event), it records an error: something unexpected has happened and needs attention. Degraded data (under-replicated due to a device going offline or being removed) is repaired automatically as soon as sufficient devices are available.

The design is state-driven rather than event-driven: reconcile looks at what the current state *should be* and compares it to what it *is*. This means multiple operations compose naturally — for example, evacuating multiple devices simultaneously just works, because each extent is evaluated independently against the current desired state.

8.1.9.1 Work tracking Work enters the system in two ways: *triggers* on individual extent updates detect mismatches between current data placement and desired options, and *scans* propagate option changes across all affected inodes. Scans are triggered by device state changes (adding, removing, or changing a device's read-write state) and by inode option changes that affect a directory tree.

On SSDs, work is tracked in logical key order in the `reconcile_work` and `reconcile_hipri` btrees, which is cheap since it matches the natural extent btree ordering. On rotational devices, work is additionally tracked in the `reconcile_work_phys` and `reconcile_hipri_phys` btrees, which reorder work by device LBA so it can be processed sequentially. This avoids random seeks on HDDs and enables parallel processing with one thread per device.

The `reconcile_pending` btree holds work that failed due to insufficient space or devices. Pending work is only retried after device configuration changes, solving the "rebalance spinning" problem where the old rebalance thread would burn CPU retrying moves that could never complete.

8.1.9.2 Priority ordering The reconcile thread processes work in priority order: high-priority metadata (under-replicated or evacuating) first, then high-priority data, then normal metadata (e.g. moving stray metadata to `metadata_target`), then normal data, then pending retries.

The `bcachefs reconcile status` command shows current progress, and `bcachefs reconcile wait` blocks until specified work types complete.

8.1.9.3 Consistency and self-healing Reconcile is inherently self-healing: its entire purpose is to detect and fix mismatches between desired and actual data placement. Beyond normal background operation, the `check_reconcile_work` recovery pass validates the work btrees against actual extent state, removing stale entries and correcting incorrectly-categorized work items (e.g. normal-priority work that should be high-priority). This pass can run online.

Extent triggers automatically mark data for reconcile whenever a mismatch is detected — including degraded writes where the desired replica count could not be satisfied. When a failed device is replaced or a new device is added, all pending work in `reconcile_pending` is automatically re-evaluated.

8.1.10 Copygc

As a copy-on-write filesystem, bcachefs never overwrites data in place. Random overwrites leave buckets partially empty — the old data is obsolete but still occupies disk space until the bucket is reclaimed. The copying garbage collector (copygc) handles this: it finds the most fragmented buckets, relocates their remaining live data, and frees the buckets for reuse. This is automatic and continuous.

Performance near full: copygc needs free space to relocate data into, so a portion of free space is reserved exclusively for it (`copygc_reserve`, 8% by default, configurable 5-21%). Normal writes cannot dip into this reserve. As the filesystem fills beyond the reserve threshold, write latency increases because new writes must wait for copygc to free space first. This is the primary reason to avoid running a bcachefs filesystem above ~90% capacity under write-heavy workloads.

A fragmentation LRU btree tracks bucket fill levels so copygc can efficiently find the worst buckets without scanning the entire allocation state. Copygc reads live data from selected buckets (located via backpointers), writes it to new buckets, and updates extent pointers atomically.

Copygc relies on backpointers to find live data in fragmented buckets. If missing or inconsistent backpointers are detected during copygc, the backpointer recovery pass is automatically scheduled and run (see Backpointers).

8.1.11 Scrub

Scrub reads all data on a running filesystem and verifies checksums, detecting silent data corruption (bitrot). When a checksum mismatch is found and a valid redundant copy exists (from replication or erasure coding), the corrupted copy is automatically repaired — the same self-healing mechanism as the normal read path, but applied proactively to all data rather than waiting for application reads to discover corruption.

Scrub walks data in physical (LBA) order using backpointers, which is efficient for rotational devices and avoids the random access pattern that would result from walking the logical extent tree. It can be run on a specific device or on all devices. Progress is reported via sysfs and can be monitored with `bcachefs data scrub`. Nocow data cannot be scrubbed (no checksums).

8.2 Allocator

8.2.1 Buckets

The allocator manages space in units called *buckets* — contiguous regions on each device, typically 512 KB to 16 MB (set at format time with `-bucket_size`, default auto-selected based on device size). Each bucket tracks how many dirty and cached sectors it contains, plus metadata such as the oldest journal sequence number referencing it, stripe membership, and generation number.

Buckets cycle through a series of states:

`dirty` Contains live data or metadata. Cannot be reused until all data is moved or deleted.

`cached` Contains only cached copies (data with durable replicas elsewhere). Can be discarded when free space is needed.

`need_gc_gens` Legacy state, retained for compatibility. Previously used to prevent generation number wraparound; now effectively unused since the invalidate worker uses backpointers instead of generation bumping.

`need_discard` All data invalidated; waiting for a discard (TRIM) command to be sent to the device.

`free` Discarded and ready for allocation.

Bucket size affects fragmentation and overhead. Larger buckets reduce metadata overhead and improve sequential write performance, but increase internal fragmentation: if a bucket is only partially filled with live data, the remaining space is wasted until copygc moves the live data elsewhere and frees the entire bucket. The `bcachefs fs usage` command shows per-device bucket counts and fragmentation.

8.2.2 Foreground allocator

The foreground allocator handles allocation requests from active writes. When a write needs space, the allocator selects devices based on the applicable target option (`foreground_target`, `metadata_target`, etc.). Within the target group, devices are selected by striping across all available devices, weighted by free space — devices with more free space receive proportionally more allocations, so all devices in the filesystem fill up at roughly the same rate.

If the target devices are full, the allocator falls back to any device in the filesystem rather than failing the write. This fallback is deliberate: target options express a preference, not a hard constraint. The only way to get a hard constraint is to use separate filesystems.

Each allocation request also specifies a watermark level (see Watermarks below) and a required number of replicas. The allocator picks devices that satisfy the replica count and durability requirements, avoiding placing multiple replicas on the same device.

8.2.2.1 Write points bcachefs automatically reduces fragmentation by segregating different types of IO into separate buckets. Data from different files, metadata, and internal bookkeeping each get their own write points (active allocation contexts). The key insight: data written at the same time by the same file tends to be deleted at the same time, so grouping it together means entire buckets become free at once rather than becoming fragmented.

User data write points are hashed by inode number, so different files naturally land in different buckets without any user configuration. Separate write points also exist for btree nodes, copygc, and the reconcile subsystem, keeping internal IO from mixing with user data.

8.2.3 Background allocator

The background allocator runs continuously, producing free buckets for the foreground allocator to consume. It manages three pipelines:

Invalidation Scans for buckets containing only cached data (or no data) and invalidates them. The invalidate worker walks backpointers to verify no live references remain before marking buckets for discard. Invalidated buckets move to the `need_discard` state.

Discard Sends TRIM/discard commands to the device for invalidated buckets, then moves them to the `free` state. On devices that do not support discard, this step is a no-op.

Freelist management Maintains a pool of free buckets ready for immediate allocation. The target free bucket count is tunable and determines how far ahead the background allocator works.

When free space is low, copygc kicks in to move live data out of mostly-empty buckets, freeing them for reuse. The copygc reserve ensures that copygc itself always has enough free space to make forward progress, even when the filesystem appears full to user writes.

8.2.4 Watermarks

The allocator uses a tiered watermark system to manage space pressure. Each watermark level reserves progressively more free buckets:

`stripe` Highest watermark; used for erasure coding stripe allocation. Most free space is available.

`normal` Standard user data writes.

`copygc` Copygc is allowed to dip into space reserved from normal writes.

`btree` Btree node allocation, which must succeed even under heavy space pressure.

`btree_copygc` Btree allocation during copygc.

`reclaim` Journal reclaim—must always be able to flush dirty btree nodes to free journal space.

`interior_updates` The lowest watermark, for btree interior node updates during splits and merges that must never fail.

This layered approach ensures that critical internal operations (journal reclaim, btree splits) can always make progress, even when the filesystem is full from the user's perspective.

8.2.5 Accounting

The accounting subsystem maintains exact, transactional counters for all space usage in the filesystem. Every write, delete, or metadata change that affects space usage atomically updates the corresponding accounting entries as part of the same transaction.

The system is designed to be extensible: accounting keys are type-tagged unions, so adding a new class of counters requires only defining a new tag and its associated fields. No schema changes, no migration — new counter types appear in the btree alongside existing ones, and old code simply ignores tags it does not recognize.

Accounting entries are stored in a dedicated btree as actual counter values, but updates are applied as deltas and aggregated by the btree write buffer before being flushed. This is how accounting can live in a btree without killing performance: many small increments are batched into a single btree update. Version numbers derived from journal position ensure that journal replay can safely deduplicate updates.

8.2.5.1 What is tracked

`replicas` On-disk usage by replication strategy — which devices hold copies and how many. This is what `bcachefs fs usage` reports as the main usage breakdown.

`dev_data_type` Per-device usage broken down by data type (user data, btree, cached, parity, etc.), tracking bucket count, live sectors, and fragmentation.

`compression` Per-compression-type statistics: number of extents, uncompressed size, and compressed size on disk.

`nr_inodes` Total inode count.

`snapshot` Per-snapshot on-disk usage.

`btree` Per-btree metadata usage (total sectors, node count).

`reconcile_work` Pending work for the reconcile subsystem, broken down by type.

`persistent_reserved` Sectors reserved by `KEY_TYPE_reservation` keys (e.g. fallocate).

In memory, frequently-accessed counters (replicas, per-device, compression) are maintained in percpu arrays for lock-free reads. Less frequently accessed counters (per-snapshot, reconcile work) are read from the btree on demand. The `bcachefs fs usage` command and the `BCH_IOCTL_QUERY_ACCOUNTING` ioctl both read from this system.

8.2.6 Replicas tracking

The replicas superblock field records every unique data replication configuration in use by the filesystem — each entry describes a data type (journal, btree, user data, parity) and the set of devices that hold copies. This is the authoritative source for determining whether the filesystem can operate with a given set of devices.

8.2.6.1 Mount decisions At mount time, bcachefs checks every replicas entry against the set of online devices. For each entry, it counts how many of the listed devices are present and compares against the entry's `nr_required` field (normally 1; higher for erasure coding where multiple blocks are needed for reconstruction). If any entry cannot be satisfied, the filesystem cannot mount — the data described by that entry would be inaccessible.

Write-side checks are stricter: the filesystem must have enough read-write devices to satisfy the configured replication levels for journal, metadata, and user data. A filesystem can mount read-only with fewer devices than it needs for read-write operation.

8.2.6.2 Lifecycle Replicas entries are added lazily: when new data is written with a previously-unseen device combination, the entry is added to the superblock as part of the transaction commit (via integration with the accounting subsystem). Entries are removed when their corresponding accounting counters reach zero — meaning no data with that replication pattern exists on disk anymore.

The tight coupling with accounting means the replicas field stays accurate without expensive scans: as data is written, moved, or deleted, accounting deltas flow through the write buffer, and the replicas field is updated to match.

As of version 1.36, user data replicas entries are no longer stored in the superblock — only journal and metadata entries are. With large numbers of devices, the combinatorial explosion of possible device sets for user data made superblock replicas entries a scalability bottleneck. User data replication is now tracked entirely through the accounting subsystem.

8.2.7 Backpointers

Every sector range on disk that contains data or metadata has a corresponding backpointer: a reverse reference from the physical location back to the logical btree entry that owns it. Backpointers answer the question "what data lives in this bucket?" without scanning the entire extents btree.

Backpointers are stored in a write-buffered btree, keyed by (device, sector offset, discriminator). The value records the btree ID, level, and position of the owning key, plus the data type and bucket generation number.

8.2.7.1 Maintenance Backpointers are created and deleted automatically by extent triggers: when an extent is written, a backpointer is inserted for each data pointer; when an extent is overwritten or deleted, the corresponding backpointers are removed. Updates go through the write buffer for batching.

The discriminator field handles cases where multiple extents share ownership of the same physical block (e.g. compressed extents that have been partially overwritten). Erasure code stripes get their own backpointers in a separate `stripe_backpointers` btree, since stripe backpointers have different position semantics and lifecycle from extent backpointers.

8.2.7.2 Operations that use backpointers

- **Copygc**: Finds live extents in fragmented buckets to relocate them, freeing the bucket for reuse.
- **Device evacuation**: Finds all extents on a device being removed and migrates them to other devices.
- **Scrub**: Walks backpointers in physical order to verify data integrity without random seeks across the extents btree.
- **Reconcile**: Tracks extents that need to be moved on rotational devices for optimal LBA ordering.

8.2.7.3 Consistency and self-healing Missing or inconsistent backpointers are detected at runtime — for example, when the move path looks up backpointers for a bucket and finds they do not match the bucket's sector counts. When a mismatch is detected, the relevant recovery pass is automatically scheduled and run (with rate limiting to avoid overwhelming the system).

Three recovery passes verify backpointer integrity bidirectionally: `check_extents_to_backpointers` ensures every extent has matching backpointers, `check_backpointers_to_extents` ensures every backpointer points to a valid extent, and `check_btree_backpointers` validates backpointers against bucket allocation state.

The key optimization is comparing backpointer sector counts against bucket sector counts: if they agree, the backpointers for that bucket are known to be consistent without walking the extents btree. Only buckets with mismatches need the more expensive bidirectional verification — essential for larger filesystems where a complete scan would be prohibitively expensive. Each backpointer also records the bucket generation number at creation time, so stale backpointers from reused buckets are detected and cleaned up automatically.

8.2.8 Data structures

The allocator's persistent state is spread across several btrees, each optimized for a different access pattern. The alloc btree is the authoritative record of per-bucket state; the others are derived indexes that accelerate specific operations.

8.2.8.1 Alloc key (bch_alloc_v4)

Every bucket on every device has a corresponding key in the alloc btree, keyed by (device, bucket number). The value is a bch_alloc_v4 struct containing:

- gen / oldest_gen — the current generation number and the oldest generation still referenced by extents. The difference between these determines whether the bucket needs a GC-gens pass.
- data_type — what kind of data the bucket holds (user, btree, cached, parity, stripe, etc.), computed by alloc_data_type() from the other fields (see below).
- dirty_sectors / cached_sectors / stripe_sectors — sector counts by category.
- stripe_refcount — number of erasure-coded stripes referencing this bucket.
- io_time[READ/WRITE] — timestamps for LRU eviction of cached data.
- journal_seq_nonempty / journal_seq_empty — journal sequence numbers tracking bucket state transitions, used by the noflush write optimization and the discard path.
- nr_external_backpointers — count of backpointers stored in the backpointers btree (as opposed to inline backpointers).

The format has evolved through four versions (v1 through v4). Earlier versions used variable-length varint encoding for fields; v4 switched to a fixed-layout struct for simpler access. v4 also added support for inline backpointers stored directly in the alloc key value, avoiding a separate btree lookup for buckets with few backpointers. All versions are converted to bch_alloc_v4 in memory; the on-disk format is upgraded lazily as keys are rewritten.

8.2.8.2 Bucket state derivation

A bucket's logical state is not stored as a field — it is *derived* from the alloc key contents by alloc_data_type(). The derivation follows a priority chain:

1. If stripe_refcount > 0: the bucket belongs to an erasure-coded stripe (BCH_DATA_stripe or BCH_DATA_parity).
2. Else if dirty_sectors > 0 or stripe_sectors > 0: the bucket contains live data; the type comes from the data that was written (user, btree, etc.).
3. Else if cached_sectors > 0: the bucket contains only cached data (BCH_DATA_cached).
4. Else if the NEED_DISCARD flag is set: the bucket is invalidated but awaiting TRIM (BCH_DATA_need_discard).

5. Else if `gen - oldest_gen >= BUCKET_GC_GEN_MAX`: the generation gap is too large (`BCH_DATA_need_gc_gens`).
6. Otherwise: the bucket is free (`BCH_DATA_free`).

This derivation means bucket state is always consistent with the underlying counters — there is no separate state field that could get out of sync.

8.2.8.3 Freespace btree The freespace btree indexes free buckets for fast allocation. Keys are (device, bucket number) with generation bits encoded in the high bits of the offset, so the allocator can scan for free buckets on a given device with a simple btree range scan. This is a derived index: entries are inserted/removed by the alloc key trigger when a bucket transitions to or from the free state. The foreground allocator cross-checks freespace entries against the alloc btree before using a bucket, catching any inconsistencies.

8.2.8.4 Need-discard btree A simple presence/absence index of buckets in the `BCH_DATA_need_discard` state. The discard worker iterates this btree to find buckets needing TRIM commands, sends the discards, then updates the alloc key to clear the need-discard flag (which removes the entry from this btree via the trigger). Maintaining a separate index avoids scanning the entire alloc btree to find the small fraction of buckets awaiting discard.

8.2.8.5 Bucket-gens btree Packs 256 bucket generation numbers into a single btree key (`bch_bucket_gens`). This provides cheap stale-pointer detection: when checking whether an extent pointer is stale, the code only needs to read a small bucket-gens key rather than the full alloc key. This is particularly important for the RCU read path, where looking up a full alloc key would be too expensive.

8.2.8.6 LRU btree Indexes cached buckets by their last-read timestamp, enabling the invalidation worker to evict the least-recently-used cached data first. Also used with a separate LRU ID for fragmentation-based eviction ordering, so copygc can prioritize the most fragmented buckets. Like the other auxiliary btrees, entries are maintained by the alloc key trigger.

8.2.9 Device labels and targets

Device labels are hierarchical paths delimited by periods — for example, `ssd.fast`, `ssd.slow`, `hdd.archive`. A target option can reference any prefix of the path: specifying `ssd` as a target matches all devices whose label starts with `ssd` (e.g. `ssd.fast`, `ssd.slow`), while `ssd.fast` matches only that specific label. Labels need not be unique — multiple devices can share the same label, forming a group.

Targets can also reference a device directly by path (e.g. `foreground_target=/dev/sda1`). Internally, both device references and label references resolve to entries in the disk groups superblock field, which maps label strings to device sets.

Four target options control where data is placed:

`foreground_target` Normal foreground data writes, and metadata if `metadata_target` is not set.

`metadata_target` Btree node writes.

`background_target` If set, user data is moved to this target in the background by the reconcile subsystem. The original copy is left in place but marked as cached.

`promote_target` If set, a cached copy is created on this target when data is read, if no copy exists there already.

All four options can be set at the filesystem level (format time, mount time, or runtime via sysfs) or on individual files and directories. Target options express a preference, not a hard constraint: if the target devices are full, the allocator falls back to any device in the filesystem.

8.2.10 Consistency and self-healing

The allocator performs runtime consistency checks during normal operation, detecting and repairing problems without requiring an offline fsck.

8.2.10.1 Runtime checks The foreground allocator validates every bucket before use: the freespace btree entry is cross-checked against the alloc btree to confirm the bucket is actually free, the generation number matches, and no other subsystem has a claim on it (open bucket, nocow lock, superblock region). If a mismatch is detected, the bucket is skipped and an asynchronous repair job is queued to fix the inconsistent entry without blocking allocation.

Bucket state transitions are also validated: if a bucket is being marked with a data type that conflicts with its current state (e.g. writing user data to a bucket the alloc btree says contains metadata), the inconsistency is flagged and a recovery pass is scheduled. Accounting counters are checked for sanity (e.g. negative sector counts indicate lost writes or corruption) and trigger recovery when anomalies are found.

8.2.10.2 Recovery passes When runtime checks detect problems, they automatically schedule the appropriate recovery pass with rate limiting to avoid overwhelming the system. The key allocator recovery passes are:

`check_allocations` Full garbage collection: marks all referenced buckets by walking extents, btree nodes, and stripes, then compares against the alloc btree and repairs data types, sector counts, and stripe references.

`check_alloc_info` Cross-checks the alloc btree against the freespace, need_discard, and bucket_gens btrees, repairing any mismatches. Can run online.

`check_lrus` Verifies LRU entries (used for cached bucket eviction order) match alloc btree timestamps; removes stale entries.

`check_alloc_to_lru_refs` Ensures every cached bucket has a correct LRU entry.

Recovery passes are ordered by dependency: `check_allocations` must run before the others, since it establishes the ground truth for bucket state. Passes marked `PASS_ONLINE` can run on a mounted filesystem without interrupting normal operation.

8.3 Subvolumes and snapshots

8.3.1 Overview

A subvolume is an independent directory tree within the filesystem, similar to btrfs subvolumes. Subvolumes appear as directories and can be created empty or as snapshots of existing subvolumes. Snapshots are writeable by default and can be snapshotted again, forming a tree of snapshots. They can also be created read-only.

Snapshots are O(1) to create regardless of filesystem size — no data or metadata is copied. Writes to either the source or the snapshot only diverge where modifications occur. Many thousands or millions of snapshots can exist, limited only by disk space.

Each snapshot tree has a *master subvolume*: the original non-snapshot subvolume from which all snapshots in the tree descend. The master subvolume is significant for quota accounting: quotas are charged based on the uid/gid/project recorded in the master subvolume's inodes. Snapshot subvolumes bypass quota enforcement entirely, because ownership changes within a snapshot would make it ambiguous which quota should be charged. If the master subvolume is deleted, quota accounting for that snapshot tree is skipped.

Subvolumes and snapshots can be managed with:

- `bcachefs subvolume create/delete/snapshot` — create, delete, and snapshot subvolumes
- `bcachefs subvolume list` — list subvolumes in tree view
- `bcachefs subvolume list-snapshots` — show snapshot tree with per-snapshot disk usage

8.3.2 Architecture

A subvolume holds a root inode number and a snapshot ID. The snapshot ID links the subvolume to a leaf node in the snapshots btree, which records the snapshot tree structure: parent, children (up to two), depth, and a skiplist for fast ancestor queries. Only leaf snapshot nodes are associated with subvolumes; interior nodes exist purely for tree structure. Snapshot trees are grouped under `snapshot_tree` entries, each recording the root snapshot and the master subvolume.

8.3.3 Key visibility

Four btrees are snapshot-aware: extents, inodes, dirents, and xattrs. Every key in these btrees includes a snapshot ID in its position (`bpos.snapshot`), so keys from different snapshots coexist in the same btree ordered by (inode, offset, snapshot). When reading from a snapshot, the iterator walks up the snapshot tree: a key is visible if its snapshot ID is an ancestor of (or equal to) the requested snapshot, and no closer ancestor has overwritten it. Deletion within a snapshot inserts a whiteout key that blocks visibility of the ancestor's version without affecting other snapshots.

To avoid a linear parent-pointer walk on every lookup, each snapshot node stores a 128-bit ancestor bitmap for O(1) checks when ancestor and descendant are within 128 snapshot IDs of each other, plus a randomized skiplist of three ancestor IDs for O(log n) convergence on deeper trees. During early recovery, before this data is validated, queries fall back to a simple parent walk.

8.3.4 Snapshot creation

When a snapshot is created, two new snapshot nodes are allocated as children of the source subvolume's current snapshot node. One child becomes the new snapshot's ID; the other replaces the source subvolume's snapshot ID. No keys are copied: both children inherit visibility of all ancestor keys through the snapshot tree. Subsequent writes to either subvolume create new keys tagged with that subvolume's snapshot ID, diverging only where modifications occur.

This is fundamentally different from btrfs, which clones entire COW btrees on snapshot. Because bcachefs snapshots share the actual btree keys (not copies), creation is O(1) regardless of filesystem size. Many thousands or millions of snapshots can be created, limited only by disk space.

8.3.5 Snapshot deletion

Deleting a snapshot (or subvolume) marks it for asynchronous cleanup by a background thread. The cost of deletion is proportional to the volume of data in the deleted snapshot, since the thread must walk every snapshot-aware btree to remove or relocate affected keys.

Deletion is two-phase:

1. **Runtime**: The background thread walks snapshot-aware btrees and removes keys belonging to dead snapshots. For leaf snapshots, keys are deleted or converted to whiteouts where ancestor visibility must be preserved. For interior nodes that have lost all but one child, keys are moved to the surviving child.

2. **Next mount**: Interior node removal is deferred to recovery because it requires updating depth and skiplist fields atomically across the subtree, which is only safe in the single-threaded recovery context.

A snapshot with two children cannot be deleted directly — you must first delete one of its child snapshots. Multiple snapshot deletions are batched and processed together in a single pass.

Progress can be monitored via `/sys/fs/bcachefs/<uuid>/snapshot_delete_status`. The `auto_snapshot_deletion` mount option controls whether the background deletion thread runs automatically.

8.3.6 Space accounting

Space usage is tracked per snapshot via the accounting subsystem. The `bcachefs subvolume list-snapshots` command shows per-snapshot disk usage attribution (own data vs. cumulative including children). Because snapshots share data through COW, the sum of individual snapshot usage will exceed the actual disk usage — the difference is shared data.

8.3.7 Consistency and self-healing

Several recovery passes validate snapshot consistency:

`check_snapshots` Validates the snapshot tree structure: parent/child links, depth fields, skiplist entries, and ancestor bitmaps. Detects and repairs orphaned or malformed snapshot nodes.

`check_subvols` Validates subvolume entries: ensures each points to a valid snapshot leaf, root inode exists, and master subvolume designation is consistent.

`delete_dead_snapshots` Runs the snapshot deletion cleanup for any snapshots marked for deletion but not yet fully cleaned up.

These passes can run online. The snapshot deletion thread itself is self-healing: if it is interrupted (crash, reboot), it resumes cleanup on next mount by re-scanning for snapshots still marked for deletion.

8.4 Superblock

The superblock stores all filesystem-wide configuration and per-device metadata. Each device maintains its own copy, and the copies are kept in sync: most fields are shared across all devices, with the exception of the device index and journal bucket locations.

8.4.1 Layout and redundancy

The primary superblock is located at sector 8 (4 KB from the start of the device). A `bch_sb_layout` structure at sector 7 records the locations of all superblock copies—typically three: the primary, one immediately following it, and one at the end of the device. Up to 61 backup locations can be recorded. The layout structure has its own magic number so that it can be found independently.

The superblock is written with a monotonically increasing sequence number (`seq`); on read, the copy with the highest valid sequence number is authoritative. The `bcachefs recover-super` command can reconstruct a device's superblock from backup copies or from another device in the same filesystem.

8.4.2 Fixed fields

The superblock header contains:

- **Identity**: filesystem UUID (immutable), user-visible UUID (mutable), filesystem label (up to 32 bytes)
- **Geometry**: block size, btree node size, number of devices
- **Versioning**: current metadata version, minimum version of any data still on disk (see the Metadata versions section)
- **State**: initialized and clean flags, sequence number, write timestamp
- **Options**: all persistent filesystem options are encoded as bitfields in the superblock flags—replication counts, checksum and compression types, error handling policy, targets, quotas, journal parameters, and more. Mount options override these at runtime; `bcachefs set-fs-option` persists changes.

8.4.3 Variable-length fields

The superblock is extensible via type-tagged variable-length fields (`BCH_SB_FIELD_*`). Some are per-device (journal bucket lists); most are shared across all devices (members, encryption, replicas, disk groups, error log, recovery state). See the On disk format section for the complete field list.

Key fields for operators:

`members_v2` Per-device metadata: UUID, bucket count and size, state (rw/ro/evacuating/spare), durability, data-type restrictions, error counters, performance measurements, and hardware identifiers.

`disk_groups` Device label hierarchy and group definitions, used for target-based allocation (see Device labels and targets).

`replicas` All unique replication configurations in the filesystem (see Replicas tracking).

`clean` Written on clean shutdown: contains btree roots and usage counters, allowing the next mount to skip journal replay entirely.

`errors` Persistent error log recording operational errors and fsck findings across mounts.

`ext` Extended metadata including required recovery passes and silenced errors.

8.4.4 Version upgrades

The superblock records both the current metadata version and the minimum version of any data still on disk. This two-version scheme allows the filesystem to upgrade incrementally: new data is written with the current version while old data retains the format it was written with. The `version_upgrade` option controls upgrade behavior at mount time: `compatible` (allow new features), `incompatible` (upgrade to latest), or `none` (don't upgrade). Downgrade information is stored separately so that a filesystem can be safely used by older tools after an upgrade if no incompatible features have been used.

8.4.5 Consistency and self-healing

Every superblock copy is checksummed; reads validate the checksum and fall back to alternative copies on failure. The sequence number provides unambiguous ordering when copies disagree. The `recover-super` command can reconstruct a completely overwritten superblock from the backup copies on the same device or from any other device in the filesystem. Recovery passes `check_alloc_info` and `check_topology` verify that superblock-recorded state matches the actual on-disk data.

8.5 Device management

bcachefs is a multi-device filesystem: a single filesystem can span any number of block devices, each contributing storage capacity and IO bandwidth. Devices need not be the same size or have the same performance characteristics—the allocator stripes across all available devices, biasing toward devices with more free space so that all devices fill at the same rate, and the read path tracks per-device IO latency to direct reads to the fastest available replica.

Devices can be added and removed at any time without unmounting.

8.5.1 Per-device metadata

Each device has a `bch_member` entry in the superblock containing:

- **Identity**: per-device UUID, device name, model string
- **Geometry**: bucket count, bucket size, first usable bucket
- **State**: rw, ro, evacuating, or spare (see below)

- **Configuration**: durability, data-type restrictions (`data_allowed`), discard (TRIM) support, rotational hint
- **Diagnostics**: cumulative error counters (read, write, checksum), performance measurements (sequential and random IO rates), last mount timestamp

8.5.2 Device states

Each device has a persistent state stored in the superblock:

`rw` Read-write: fully operational, participates in allocation

`ro` Read-only: can be read from but receives no new writes

`evacuating` Being emptied of data prior to removal

`spare` Reserved, not currently participating in IO

Device state is changed with `bcachefs device set-state`. Transitions that would reduce write redundancy below the configured replication level require the `--force` flag.

Separately from the persistent state, a device can be *online* (kernel has the device open) or *offline* (device is listed in the superblock but not currently accessible).

8.5.3 Durability

The `durability` setting controls how many replicas a copy on a given device counts for. The default is 1. Setting `durability=2` on a hardware RAID device tells bcachefs that data on that device already has internal redundancy—it counts as two replicas, so the filesystem does not need to keep an additional copy elsewhere. Setting `durability=0` means copies on the device do not count toward replication requirements at all—the device can only be used as a cache.

8.5.4 Caching

When an extent has multiple copies on different devices, some of those copies may be marked as *cached*. Cached copies are evicted in LRU order by the allocator when the device needs space. Caching behavior is controlled through the target options:

Writeback caching Set `foreground_target` and `promote_target` to the cache device, and `background_target` to the backing device. Writes land on the fast device first and migrate to the backing device in the background.

Writearound caching Set `foreground_target` to the backing device and `promote_target` to the cache device. Writes go directly to the backing device; frequently-read data is promoted to the cache.

The `durability=0` setting is essential for cache devices: it ensures bcachefs does not count cached copies toward the replica count, so losing the cache device never causes data loss.

8.5.5 Adding and removing devices

`bcachefs device add` Adds a new device to a mounted filesystem. The device is formatted with bcachefs metadata and integrated immediately—new allocations can land on it right away. A label can be assigned at add time with `-l`. Other per-device options (`--discard`, `--durability`) can be set at add time.

The new device must have a block size and bucket size compatible with the existing filesystem. After the device is added, its UUID is published via uevent so that `/dev/disk/by-uuid` symlinks are updated, and the reconcile subsystem is notified to scan for any work on the new device.

`bcachefs device evacuate` Migrates all data off a device, displaying progress as sectors are moved. Uses the reconcile subsystem internally; the device's state transitions to evacuating during the process. Requires metadata version $\geq$ `reconcile` (1.33).

`bcachefs device remove` Removes a fully evacuated device from the filesystem and erases its metadata. Force flags allow removal even if some data (`-f`) or metadata (`-F`) would be lost.

Two removal code paths exist: the legacy path walks the btree to find and relocate all references to the device, while the `fast_device_removal` path (default on newer metadata versions) uses backpointers to efficiently locate all data on the device without a full btree scan.

`bcachefs device online/offline` Bring a device back online or take it offline without removing it. Offline devices retain their superblock membership and can be brought back later. Bringing a device online includes a splitbrain check against the running filesystem's sequence numbers; onlining also triggers a reconcile scan to detect any data that may need re-replication.

Offlining a device requires that the remaining online devices can still satisfy both read and write requirements—the kernel checks that at least one device can serve reads and at least one can accept writes for every replica group. If offlining would leave the filesystem unable to operate, the request is rejected unless forced.

The typical device removal workflow: `bcachefs device evacuate /dev/sda` (wait for completion, watching progress), then `bcachefs device remove /dev/sda`.

8.5.6 Block layer hot-remove

When the block layer reports a device as dead (e.g., a USB drive is unplugged, or a disk is removed from a hot-swap bay), bcachefs receives a notification and attempts a graceful response. If the device can be offlined without leaving the filesystem unable to operate, it is taken offline automatically. Otherwise, the filesystem transitions to emergency read-only mode to prevent data corruption from writes that can no longer reach all required replicas.

8.5.7 Data-type restrictions

The `data_allowed` member field restricts which data types a device can hold: journal, btree, or user data. This allows dedicating fast devices to metadata while slower devices hold only user data, or restricting a device to journal-only for write-ahead log isolation. Restrictions are set at format time or via `set-fs-option` and are enforced by the allocator.

8.5.8 Degraded mode

When a device is unavailable (failed, offline, or physically disconnected), the filesystem can continue operating in degraded mode if sufficient redundancy remains. The number of tolerable failures per replica group is `nr_devs - nr_required`: with 3-way replication, one device can fail without data loss.

The `degraded` mount option controls behavior when devices are missing:

`degraded=true` Allow mounting with missing devices (read-only access to degraded data)

`degraded=run` Allow mounting and normal operation with missing devices

`degraded=very` Allow mounting even if writes cannot maintain the requested replica count (**dangerous**—creates splitbrain risk)

While degraded, the filesystem has reduced safety margin—further device loss may cause data unavailability. The reconcile subsystem will automatically repair degraded data by re-replicating to available devices.

8.5.9 Resize

`bcachefs device resize` grows a device to use additional space (shrinking is not yet supported). If no size is specified, the device grows to fill its underlying block device. Resize works online—no unmount required. The new size is subject to a maximum bucket count (`BCH_MEMBER_NBUCKETS_MAX`); resize will fail if the requested size would exceed this limit. After resize, the reconcile subsystem is notified to account for the newly available space.

`bcachefs device resize-journal` adjusts the per-device journal size independently of the data area.

8.5.10 Device failure and error tracking

Each device tracks cumulative error counters (read, write, checksum) in the superblock members section. These counters persist across mounts and help identify failing hardware before catastrophic failure. The `write_error_timeout` option (default 30 seconds) controls how long sustained write errors must persist before the device is automatically set to read-only.

When a device is set to read-only due to errors, reads can still be served from it. If reads also fail, the device should be taken offline entirely to prevent journal stalls—the journal cannot reclaim space if it cannot read back btree nodes from a failed device.

8.5.11 Consistency and self-healing

Device membership is tracked in the superblock and cross-validated against on-disk data during recovery. The allocator checks freespace and alloc btrees against each other before using a bucket. Backpointer walks verify that all data on a device is accounted for. If a device is removed or fails, the reconcile subsystem detects under-replicated data and re-replicates it to remaining devices automatically.

8.6 Journal

The journal is a write-ahead log for metadata. Instead of writing every btree update directly to its btree node on disk, bcachefs records updates in the journal first. This allows metadata writes to be batched and sequential, dramatically improving performance.

8.6.1 How the journal works

Each journal entry (`struct jset`) contains a list of typed sub-entries: btree key updates, btree root pointers, timestamps, IO clock values, and diagnostic messages. Entries are assigned monotonically increasing sequence numbers that survive crashes and are never reused.

The journal is stored as a ring buffer of buckets on each device. As new entries are written, they advance through the ring. Old entries are reclaimed once all the btree nodes they refer to have been flushed to disk.

8.6.2 Journal pins and reclaim

A *journal pin* holds a reference from a dirty btree node (or key cache entry) to the journal sequence that contains its latest update. Journal space for a sequence cannot be reclaimed until all pins referencing that sequence are released—which happens when the corresponding btree node is written to disk.

The journal reclaim thread runs in the background, identifying which btree nodes are pinning the oldest journal sequences and flushing them. Under normal operation this is invisible; under heavy write load, reclaim may need to work harder to keep up.

8.6.3 Space pressure

When free journal space drops below 25%, or the pin list fills to 75% capacity, the journal enters a reclaim watermark state. In this state:

- New metadata writes may be throttled
- The reclaim thread is woken to aggressively flush btree nodes
- Space is typically freed within milliseconds as nodes flush

If the journal fills completely, metadata operations block until space is freed. This is rare under normal workloads and resolves automatically. A sustained "journal full" condition typically indicates that btree node writes are bottlenecked—often by a slow device or high IO contention.

8.6.4 Flush and ordering

Journal writes come in two flavors:

Flush writes Ordered to stable storage with disk cache flushes. These provide durability guarantees—data acknowledged to applications via `fsync()` is protected by flush writes. A configurable delay (`journal_flush_delay`, default 1000 ms) batches updates before flushing.

No-flush writes Written without ordering guarantees. These can be lost on power failure but are much cheaper. Used between flush points to reduce IO overhead.

On multi-device filesystems, flush writes issue a preflush to all devices first, ensuring all pending data writes are ordered before the journal entry.

8.6.5 Mount and recovery

On mount, the journal is read from all devices. The recovery window is determined by two sequence numbers: `last_seq` (the oldest entry still needed) and the sequence of the last valid flush entry. All entries in this window are replayed in order, re-inserting their btree keys into the btree. Journal replay is idempotent—replaying the same entry twice is safe.

On clean shutdown, a special `clean` field is written to the superblock containing the btree roots and usage counters, allowing the next mount to skip journal replay entirely.

Sequence blacklisting: After an unclean shutdown, some btree nodes on disk may reference journal sequences that were never durably committed. These sequences are added to a blacklist stored in the superblock; any btree node data referencing a blacklisted sequence is ignored during recovery. Once the affected nodes are rewritten with new sequences, the blacklist entries are garbage collected.

8.6.6 User-facing options

`journal_flush_delay` Milliseconds before auto-committing the journal (default 1000). Lower values reduce the window of data loss on crash; higher values improve throughput.

`journal_flush_disabled` Disable journal flushes entirely. **Dangerous**—data loss is expected on any unclean shutdown.

`journal_reclaim_delay` Milliseconds before triggering background reclaim (default 100).

`journal_transaction_names` Log function names in journal entries for debugging (default enabled).

Journal size is configured per device and can be resized online with `bcachefs device resize-journal`.

8.6.7 Consistency and self-healing

Every journal entry is checksummed. Entries that fail checksum validation are skipped during replay, with the filesystem falling back to the last known good entry. The sequence blacklist mechanism ensures that partially-written state from crashes cannot corrupt the btree. Journal entries are replicated across devices according to the `metadata_replicas` setting; if one device's journal is unreadable, recovery proceeds from the other copies.

8.7 Btrees

bcachefs is, at its core, a transactional key-value store built on b+trees. Every piece of filesystem state—file data mappings, inodes, directory entries, allocation tracking, accounting—is a key-value pair in one of 28 btrees. There are no separate inode tables, bitmap allocators, or per-inode extent trees. This uniform design means all metadata operations share the same transaction, caching, and recovery infrastructure.

8.7.1 The btrees

The btrees fall into several functional groups:

Core filesystem data:

`extents` Maps file offsets to physical disk locations (extent pointers). Snapshot-aware: different snapshots of the same file can have different extents in the same btree.

`inodes` Inode metadata (size, permissions, timestamps, etc.). Snapshot-aware.

`dirents` Directory entries mapping names to inode numbers. Snapshot-aware.

`xattrs` Extended attributes. Snapshot-aware.

`reflink` Shared extent pointers for reflinked (deduplicated) data.

Allocation and space management:

`alloc` Bucket allocation state—one key per bucket, tracking data type, dirty sectors, generation number, and other per-bucket metadata.

`freespace` Free space index, keyed so free extents can be found by size and location.

`need_discard` Buckets waiting for TRIM/discard before reuse.

`bucket_gens` Bucket generation numbers, used to detect stale pointers cheaply without reading the full alloc key.

`backpointers` Reverse mappings from physical disk locations back to the btree keys that reference them (see Backpointers). Enables efficient device removal, scrubbing, and data migration.

Snapshots and subvolumes:

`subvolumes` Subvolume metadata (root inode, parent, flags); see Subvolumes and snapshots.

`snapshots` Snapshot tree structure (parent/child/sibling links, depth).

`snapshot_trees` Roots of snapshot trees.

`subvolume_children` Parent-child relationships between subvolumes.

`deleted_inodes` Inodes pending deletion (may be visible in some snapshots but not others).

Reconcile (background maintenance):

`reconcile_work`, `reconcile_hipri` Work items for re-replication, migration, and other background data operations, at normal and high priority.

`reconcile_pending` Work items not yet ready to process.

`reconcile_scan` Scan state for discovering new work.

`reconcile_work_phys`, `reconcile_hipri_phys` Physical (device-keyed) variants of the work queues.

Other:

`quotas` User, group, and project quota counters.

`stripes` Erasure coding stripe descriptors.

`bucket_to_stripe`, `stripe_backpointers` Mappings between buckets and erasure coding stripes.

`lru` Least-recently-used tracking for cache eviction.

`logged_ops` In-progress logged operations for crash recovery of multi-transaction operations.

`accounting` Space accounting broken down by replica set, disk group, compression type, and snapshot.

8.7.2 What the btree design enables

The uniform btree design has consequences that are visible to users:

Consistent performance at scale. Btree nodes are large (128K–256K), making trees very shallow—typically 2–3 levels even at petabyte scale. Combined with the filesystem's own node cache (independent of the Linux page cache), this means metadata lookups rarely need more than one disk read, even under memory pressure.

Atomic cross-object operations. Because all metadata lives in the same transactional system, operations that touch multiple objects—creating a file (inode + dirent + accounting), writing data (extent + backpointer + allocation)—are atomic. There is no window where the filesystem is inconsistent, even on crash.

Online fsck and repair. The btree transaction system supports online fsck: recovery passes can run on a mounted filesystem, checking and repairing metadata while normal IO continues. This is possible because the same transaction/locking infrastructure protects both normal operations and fsck.

Efficient background maintenance. Backpointers enable operations that would otherwise require full btree scans: device removal, data scrubbing, re-replication, and rebalancing can all find the relevant data by walking backpointers rather than scanning the entire extent tree.

Snapshot efficiency. Snapshot-aware btrees store all snapshot versions of a key in the same btree, sharing the tree structure. Taking a snapshot is O(1)—no data or metadata is copied. Reads in a snapshot context find the correct version through the snapshot ID in the key position.

8.7.3 Node structure

Btree nodes are log structured internally: new keys are appended rather than inserted in sorted order. Each node consists of multiple sorted sets (bsets)—the initial sorted set from the last full rewrite, plus additional bsets appended by subsequent updates. Lookups merge results from all bsets. When a node is split or compacted, all bsets are merged into a single sorted set.

This log-structured format has two important properties: writes are sequential (good for both SSDs and spinning disks), and a node can be written to disk under a read lock rather than a write lock, since new data is only appended. This keeps btree lock hold times short and avoids blocking readers during writeback.

Keys within a bset are packed: a per-bset format descriptor records which fields are common across all keys, and those fields are stored in a compressed representation. This typically saves 30–50% of metadata space and improves cache utilization.

8.7.4 On-disk format

A btree node occupies a contiguous region on disk (typically 128K–256K). The first sector contains a `btree_node` header; subsequent sectors contain `btree_node_entry` records. Each of these structures wraps a single **bset**—a sorted array of packed keys.

The btree_node header contains:

- A checksum and magic number for validation.
- A sequence number (`BTREE_NODE_SEQ`) used to detect stale reads and order writes.
- Flags encoding the btree ID and the node's level in the tree (0 for leaves).
- A closed `[min_key, max_key]` interval describing the key range this node covers.
- A `bkey_format` descriptor used for packing keys in the node's first bset.
- The first bset, inline in the header.

The first bset in a node is the result of the last full compaction—it contains all live keys, fully sorted. Subsequent `btree_node_entry` records are journal-order appends: each contains a bset of keys that were written since the last compaction, sorted within the bset but not merged with earlier bsets. On read, all bsets are merged to produce the full sorted key set.

Bset structure. A bset (`struct bset`) is a sorted array of packed keys preceded by a small header: a sequence number, the journal sequence number of the most recent key in the set (used during recovery to discard bsets whose journal entries were lost), flags, and a count of the data that follows. The packed keys are laid out contiguously, each self-describing its total size via the `u64s` field.

Key position (bpos). Every btree key is addressed by a `bpos`: an (inode, offset, snapshot) triple. The inode and offset fields together identify the logical object and position; the snapshot field enables snapshot-aware btrees to store multiple versions of the same logical key in a single btree, distinguished by their snapshot ID. The entire bpos is treated as a single large integer for comparison, so keys are sorted first by inode, then offset, then snapshot.

Key structure (bkey). A `bkey` is the full unpacked key: a bpos, a size field (used by extent keys to describe a range of offsets), a version (for versioned key-value pairs), and a type tag identifying the value format. The `u64s` field gives the total size of key plus value in 8-byte units. Following the key header is a type-specific value (e.g., extent pointers, inode data, a directory entry hash).

Key packing. On disk, keys are stored in a packed representation (`bkey_packed`) to reduce metadata size. A `bkey_format` descriptor—stored in the btree node header—specifies, for each of the six key fields (inode, offset, snapshot, size, version_hi, version_lo), a base offset and a bit width. Fields that are identical across all keys in a bset (e.g., inode number within a single file's extents) can be stored in zero bits. The packed key retains only the 3-byte header (u64s, format, type) followed by the variable-length packed fields as a single bitstring. This typically compresses keys to 8–16 bytes instead of the 40-byte unpacked form. The format field in each key selects between the node-local packed format (`KEY_FORMAT_LOCAL_BTREE`) and the full unpacked format (`KEY_FORMAT_CURRENT`), so unpacked keys can coexist with packed keys in the same bset when a key doesn't fit the format.

Interior node pointers. Interior (non-leaf) btree nodes contain keys whose values are `bch_btree_ptr_v2` structures: pointers to child nodes. Each pointer contains the child's sequence number (for staleness detection), the number of sectors written, a `min_key` (the actual minimum key in the child, which may differ from the key's bpos in the parent after node splits), and one or more `bch_extent_ptr` entries giving the physical device and offset where the child is stored. The key's bpos in the parent gives the child's *maximum* key; the child covers the range `(prev_key, this_key]`.

8.7.5 Node cache and locking

In-memory btree nodes are kept in a hash table indexed by their physical on-disk pointer (not their logical position). This is because btree node split and compact operations are copy-on-write: new nodes are allocated, the parent is updated to point to them, and the old nodes are freed. Indexing by physical pointer avoids the need to atomically update the hash table during these operations.

The `struct btree` objects themselves are never freed during normal operation (only at shutdown), which means locks can be dropped and retaken without reference counting. This enables the aggressive lock-dropping discipline that keeps btree lock hold times bounded to in-memory operations: after dropping a lock, a sequence number check determines whether the node has changed and needs to be re-traversed.

Btree roots are pinned in memory and accessed directly. All other nodes may be evicted from the cache and reused for different on-disk nodes at any time when unlocked, so after locking a node the caller must verify it is still the expected node.

8.7.6 Iterators

A btree iterator (`struct btree_iter`) maintains a full path from the root of a btree down to a specific key in a leaf node. At each level, the iterator holds a pointer to the btree node, a node-level iterator (which merges across the multiple sorted bsets within that node), and a lock sequence number for efficient relock-after-restart.

The fundamental invariant is that iter.pos must always be consistent with the node iterators: at every level, the node iterator points to the first key >= iter.pos, and the previous key compares strictly less. This is the correct position for inserting a new key at iter.pos.

Iterators support several modes: peek() returns the next key and advances iter.pos to match; peek_slot() returns the key at exactly iter.pos (synthesizing a deleted key for empty slots); peek_prev() iterates backwards. Extent iterators have special semantics: extents are indexed by their end position, so the first extent covering a range starting at iter.pos is found by searching for the first key strictly greater than iter.pos. When iterating over extents by slots, holes between extents are synthesized, guaranteeing that the returned keys exactly cover the keyspace with monotonically increasing positions.

Multiple iterators within a transaction coexist without invalidating each other. On transaction restart, each iterator's saved position allows it to resume from where it left off.

8.7.7 Key cache and write buffer

Some btrees benefit from specialized access patterns:

The **key cache**: Fast single-key lookups for btrees where the same keys are read and updated frequently. The primary user is the alloc btree: extent updates need to update allocation information for the affected buckets, and doing a full btree lookup for each would be expensive. The key cache keeps recently accessed alloc keys in a hash table, allowing updates to be applied directly without a btree traversal. Cached entries are flushed to the btree by journal reclaim.

The **write buffer**: Batching layer for btrees that receive many small updates which are more efficient to apply in bulk. Backpointers, LRU entries, and accounting updates are written to the write buffer rather than directly to the btree; the buffer is sorted and flushed periodically, coalescing updates to the same key and amortizing the cost of btree traversal across many updates.

9 ioctl interface

This section documents bcachefs-specific ioctls:

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

BCH_IOCTL_FS_USAGE
: Obsolete. Formerly queried filesystem usage; replaced by BCH_IOCTL_QUERY_ACCOUNTING.

BCH_IOCTL_DEV_USAGE
: Obsolete. Formerly queried per-device usage; replaced by BCH_IOCTL_QUERY_ACCOUNTING.

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, evacuating, 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 some operations 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

`BCH_IOCTL_FSCK_OFFLINE`
: Runs offline fsck on an unmounted filesystem via ioctl, reporting progress and results through a file descriptor.

`BCH_IOCTL_FSCK_ONLINE`
: Runs online fsck on a mounted filesystem, executing the same recovery passes that `-o fsck` would run at mount time.

`BCH_IOCTL_QUERY_ACCOUNTING`
: Queries disk accounting data from the accounting btree: per-replica-set usage, per-device usage, compression ratios, and other counters. Replaces the obsolete `FS_USAGE` and `DEV_USAGE` ioctls.

`BCH_IOCTL_SUBVOLUME_LIST`
: Lists subvolumes in a filesystem.

`BCH_IOCTL_SNAPSHOT_TREE`
: Queries the full snapshot tree with per-node disk accounting.

`BCHFS_IOC_REINHERIT_ATTRS`

Most disk management ioctls (`DISK_ADD`, `DISK_REMOVE`, `DISK_ONLINE`, `DISK_OFFLINE`, `DISK_SET_STATE`, `DISK_RESIZE`, `DISK_RESIZE_JOURNAL`, `SUBVOLUME_CREATE`, `SUBVOLUME_DESTROY`) have v2 variants that add structured error reporting via an embedded error message buffer.

10 On disk format

10.1 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:

`journal` Journal bucket list for this device `members_v1` Member device list (v1)

`crypt` Encryption key and KDF settings

`replicas_v0` Replica entries (v0)

`quota` Quota timelimit and warnlimit fields

`disk_groups` Device label strings and label path tree structure

`clean` Clean shutdown: btree roots and usage counters, allowing journal replay to be skipped

`replicas` Replica entries: lists of devices with extents replicated across them

`journal_seq_blacklist` Blacklisted journal sequence numbers

`journal_v2` Journal bucket list (v2)

`counters` Persistent counters: IO statistics, error counts, and cumulative metrics across mounts

`members_v2` Member device list (v2)

`errors` Persistent error log: records errors detected during operation or fsck so they survive across mounts

`ext` Extended superblock: required recovery passes, accumulated silenced errors, and other flags

`downgrade` Minimum on-disk format version for safe downgrade, with required recovery passes

`recovery_passes` Tracks which recovery passes have been run successfully

`extent_type_u64s` Per-extent-type size limits

Several field types have versioned variants (e.g. `members_v1`/`v2`, `replicas_v0`/`replicas`, `journal`/`journal_v2`) for backward compatibility; the kernel reads both old and new formats but writes only the current version.

10.2 Journal

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

`btree_keys` Btree key updates

`btree_root` Btree root pointers, recorded every journal write

`prio_ptrs` Legacy, no longer used

`blacklist` Blacklist a single journal sequence number `blacklist_v2` Blacklist a range of journal sequence numbers

`usage` Maximum key version for encryption nonce derivation

`data_usage` Legacy: usage accounting moved to accounting btree

`clock` IO clock: total reads and writes in sectors since filesystem creation

`dev_usage` Legacy: per-device usage moved to accounting btree

`log` Free-form log message for fsck actions and diagnostic events

`overwrite` Old value being overwritten, for debugging and journal_rewind

`write_buffer_keys` Write buffer keys, transformed to btree_keys before writing to disk

`datetime` Wall clock time at journal write

`log_bkey` Structured log entry containing a btree key

10.3 Btrees

bcachefs uses 28 separate btrees for different data types, each identified by a numeric ID. Below is the complete list:

`extents` File data extent pointers and reservations *(snapshot-aware, extent-based)*

`inodes` Inode metadata *(snapshot-aware)*

`dirents` Directory entries *(snapshot-aware)*

`xattrs` Extended attributes *(snapshot-aware)*

`alloc` Bucket allocation state

`quotas` User, group, and project quota counters

`stripes` Erasure coding stripe descriptors

`reflink` Reflink shared extent pointers *(extent-based)*

`subvolumes` Subvolume metadata

`snapshots` Snapshot tree structure

`lru` Least-recently-used tracking for cache eviction *(write-buffered)*

`freespace` Free space index *(extent-based)*

`need_discard` Buckets waiting for discard/TRIM `backpointers` Reverse pointers from data extents back to btree nodes *(write-buffered)*

`bucket_gens` Bucket generation numbers for stale pointer detection

`snapshot_trees` Snapshot tree roots

`deleted_inodes` Inodes pending deletion *(snapshot-field, write-buffered)*

`logged_ops` In-progress logged operations for crash recovery

`reconcile_work` Reconcile work items *(snapshot-field, write-buffered)*

`subvolume_children` Subvolume parent-child relationships

`accounting` Space accounting by replicas and disk groups *(snapshot-field, write-buffered)*

`reconcile_hipri` High-priority reconcile work items *(snapshot-field, write-buffered)*

`reconcile_pending` Pending reconcile work items *(snapshot-field, write-buffered)*

`reconcile_scan` Reconcile scan state

`reconcile_work_phys` Physical reconcile work items *(write-buffered)*

`reconcile_hipri_phys` Physical high-priority reconcile work items *(write-buffered)*

`bucket_to_stripe` Bucket to stripe mapping for erasure coding

`stripe_backpointers` Stripe backpointers *(write-buffered)*

Btrees marked as “snapshot-aware” include a snapshot ID in every key position, allowing different snapshot versions to coexist within the same btree. Some btrees are “write-buffered”: updates are batched in memory and flushed periodically rather than applied directly to btree nodes, reducing write amplification for high-frequency per-IO metadata (backpointers, LRU timestamps, accounting counters). Write-buffered updates are unordered and eventually-consistent: the btree does not reflect pending updates until the next flush, and flushing sorts by key position, discarding temporal ordering. Flushing deduplicates redundant updates, then walks btree nodes in order for efficient bulk insertion; this sort-merge-sweep is inherently single-threaded, making its efficiency critical for multithreaded workloads.

10.4 Btree keys

`deleted` Transient during btree updates; inserting a deleted key removes any existing key at that position. Stripped during btree node writes.

`whiteout` Blocks visibility of ancestor snapshot versions of a key

`error` Marks an extent as containing unrecoverable errors

`cookie` Used by reconcile to track option changes via incrementing version numbers in the reconcile_scan btree

`hash_whiteout` Whiteout for hash table btrees (dirents, xattrs) that preserves hash chain integrity

`btree_ptr` Btree node pointer (v1, legacy)

`extent` File data extent with device pointers, checksums, and optional compression metadata

`reservation` Disk space reservation for a file region

`inode` Inode metadata (v1, legacy)

`inode_generation` Tracks generation numbers for deleted inodes

`dirent` Directory entry with name hash, inode number, and type

`xattr` Extended attribute with name hash and value

`alloc` Bucket allocation metadata (v1, legacy)

`quota` Quota counters for a user, group, or project

`stripe` Erasure code stripe metadata with parity pointers

`reflink_p` Pointer from the extents btree to an indirect extent in the reflink btree

`reflink_v` Indirect extent with refcount in the reflink btree

`inline_data` Small file data stored inline in the btree

`btree_ptr_v2` Btree node pointer (v2) with sequence number and min_key for more efficient traversal

`indirect_inline_data` Reflinked inline data with refcount

`alloc_v2` Bucket allocation metadata (v2, legacy)

`subvolume` Subvolume metadata with root inode and snapshot ID

`snapshot` Snapshot tree node with parent, children, and subvolume references `inode_v2` Inode metadata (v2) with journal sequence number

`alloc_v3` Bucket allocation metadata (v3, legacy)

`set` Empty value; presence of the key is the information

`lru` LRU tracking entry for cache eviction

`alloc_v4` Bucket allocation metadata (v4, current)

`backpointer` Reverse pointer from a bucket back to the extent referencing it

`inode_v3` Inode metadata (v3, current) with compact encoding

`bucket_gens` Packed bucket generation numbers (256 per key)

`snapshot_tree` Root entry for a snapshot tree structure

`logged_op_truncate` Logged truncate operation for crash recovery

`logged_op_finsert` Logged file insert/collapse operation for crash recovery

`accounting` Disk accounting delta (replicas, compression, device usage)

`inode_alloc_cursor` Per-CPU inode number allocation cursor

`extent_whiteout` Whiteout specific to the extents btree, blocking visibility of ancestor snapshot extent versions

`logged_op_stripe_update` Logged stripe creation/update operation for crash recovery

10.5 Metadata versions

Each on-disk format change is identified by a version number using `BCH_VERSION(major, minor)`. The version history records what changed and when, serving as a changelog for the on-disk format. The filesystem's superblock records both the current version and the minimum version of any data still on disk, allowing compatibility code to be safely dropped.

`bkey_renumber` *(0.10, 2018-11)* Renumbered bkey type values

`inode_btree_change` *(0.11, 2020-03)* Swapped inode/offset fields in bpos for better locality

`snapshot` *(0.12, 2021-03)* Snapshot support: snapshot field in bpos, KEY_TYPE_snapshot, KEY_TYPE_subvolume

`inode_backpointers` *(0.13, 2021-04)* Backpointers from extents to inodes

`btree_ptr_sectors_written` *(0.14, 2021-07)* sectors_written field in btree_ptr_v2 for partial write tracking `snapshot_2` *(0.15, 2021-09)* Snapshot fixes: subvolume root inode tracking

`reflink_p_fix` *(0.16, 2021-10)* Fixed reflink pointer refcount handling

`subvol_dirent` *(0.17, 2021-10)* Linked subvolumes to directory entries

`inode_v2` *(0.18, 2021-11)* KEY_TYPE_inode_v2 with improved field layout

`freespace` *(0.19, 2022-03)* Freespace btree for faster allocation

`alloc_v4` *(0.20, 2022-03)* KEY_TYPE_alloc_v4 with comprehensive allocation metadata

`new_data_types` *(0.21, 2022-04)* Refined data type classifications in the allocator

`backpointers` *(0.22, 2022-06)* Backpointers btree mapping physical locations back to logical extents

`inode_v3` *(0.23, 2022-10)* KEY_TYPE_inode_v3 with compact encoding

`unwritten_extents` *(0.24, 2022-11)* Preallocated extents that read as zeros

`bucket_gens` *(0.25, 2022-12)* bucket_gens btree for stale pointer detection

`lru_v2` *(0.26, 2022-12)* Improved LRU tracking for cache eviction

`fragmentation_lru` *(0.27, 2023-02)* LRU tracking for fragmented extents

`no_bps_in_alloc_keys` *(0.28, 2023-03)* Backpointers moved fully to backpointers btree

`snapshot_trees` *(0.29, 2023-05)* snapshot_trees btree for hierarchical management

`major_minor` *(1.0, 2023-07)* Major version bump: format stabilization, major.minor versioning scheme

`snapshot_skiplists` *(1.1, 2023-07)* Skiplist structure for O(log n) ancestor queries

`deleted_inodes` *(1.2, 2023-08)* deleted_inodes btree for crash-safe unlink

`rebalance_work` *(1.3, 2023-10)* Background rebalance work tracking

`member_seq` *(1.4, 2023-12)* Sequence numbers on member devices for stale superblock detection

`subvolume_fs_parent` *(1.5, 2024-02)* fs_parent field on subvolumes for path resolution

`btree_subvolume_children` *(1.6, 2024-02)* subvolume_children btree for parent-child lookups `mi_btree_bitmap` *(1.7, 2024-04)* Per-device bitmap tracking which btrees have data

`bucket_stripe_sectors` *(1.8, 2024-06)* Stripe sector tracking in alloc keys for erasure coding

`disk_accounting_v2` *(1.9, 2024-01)* On-disk accounting btree for persistent space accounting

`disk_accounting_v3` *(1.10, 2024-08)* Accounting key validation and endianness fixes

`disk_accounting_inum` *(1.11, 2024-08)* Per-inode space accounting

`rebalance_work_acct_fix` *(1.12, 2024-08)* Fixed rebalance work accounting bugs

`inode_has_child_snapshots` *(1.13, 2024-10)* Flag on inodes indicating child snapshot data exists

`backpointer_bucket_gen` *(1.14, 2024-11)* Bucket generation in backpointers for stale detection without alloc key reads

`disk_accounting_big_endian` *(1.15, 2024-11)* Big-endian disk accounting fix

`reflink_p_may_update_opts` *(1.16, 2024-12)* reflink_p keys carry independently updatable extent options

`inode_depth` *(1.17, 2024-12)* Directory depth tracking in inodes

`persistent_inode_cursors` *(1.18, 2024-12)* Persistent inode number allocation cursors

`autofix_errors` *(1.19, 2024-12)* Default error action changed to fix_safe

`directory_size` *(1.20, 2025-01)* Directory size tracking in inode metadata

`cached_backpointers` *(1.21, 2025-02)* In-memory backpointer caching for faster lookups

`stripe_backpointers` *(1.22, 2025-02)* Backpointers for erasure-coded stripes

`stripe_lru` *(1.23, 2025-02)* LRU tracking for stripe cache eviction

`casefolding` *(1.24, 2025-02)* Case-insensitive filename support with per-directory flags

`extent_flags` *(1.25, 2025-03)* Flags field on extent keys for extent-level metadata

`snapshot_deletion_v2` *(1.26, 2025-05)* Improved snapshot deletion with better interior node handling `fast_device_removal` *(1.27, 2025-05)* Fast device removal when no stale pointers exist

`inode_has_case_insensitive` *(1.28, 2025-05)* Casefold flag on inodes for directory handling

`extent_snapshot_whiteouts` *(1.29, 2025-08)* KEY_TYPE_extent_whiteout for efficient snapshot extent deletion

`31bit_dirent_offset` *(1.30, 2025-08)* Extended directory entry offset to 31 bits

`btree_node_accounting` *(1.31, 2025-09)* Per-btree-node space accounting

`sb_field_extent_type_u64s` *(1.32, 2025-11)* Superblock field for extent type size limits

`reconcile` *(1.33, 2025-11)* Reconcile system for IO options inheritance and background data movement

`extented_key_type_error` *(1.34, 2025-12)* KEY_TYPE_error changed to zero-byte value

`bucket_stripe_index` *(1.35, 2026-01)* Stripe index in alloc keys for efficient stripe-bucket lookups

`no_sb_user_data_replicas` *(1.36, 2026-01)* Removed redundant user_data replicas from superblock