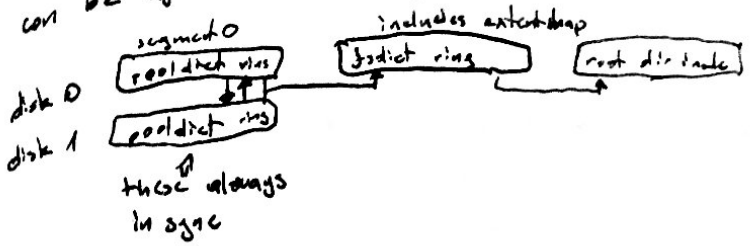
'pooldict' is present on all devices constituting the RAID/pool. It contains names/ids of backends and volumes, allocation of segments to volumes, extent info of 'fsdict' rings, crypto keys, list of segments awaiting wipe

'fsdict' is a chain containing information about the volume: free extent list (constrained by segments allocated by the pool), a map of extent ids to extent infos (+ possibly checksums). Both pooldict and fsdict are rings and can be updated atomically and by patches. Mirrored depending on setting.

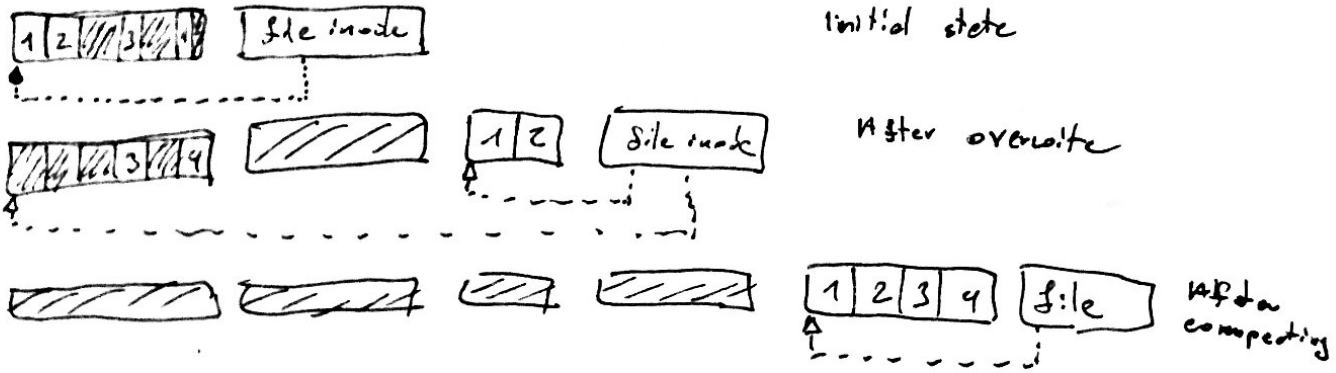


pooldict is a divided into 2 half-segments. Updates are stored as entire structures, no patches. fsdict is a 'chain' segment that starts with a full structure followed by however many patches. When the chain gets full a new segment is allocated, entire structure moved, and parent pooldict updated. Storing a patch de in a chain does not trigger an update in its parent.

fsdict can span many segments if extent map expands into level 2 or 3 (hashtable or Btree). That does not create a problem because fsdict itself is small enough to be moved.

extent map is a mapping from integer extent ids to extent infos.

Directory inodes map filenames to extent ids of file inodes. Each directory inode, file inode, symlink inode are stored in single extents. Root directory inode is always id=0. File content is stored in data extents divided into chunks.



Each time a file write is called, data buffered is stored in RAM. After a few second cycle, or when write buffer overfills, a new data extent is allocated for each dirty file. All new extents are allocated next to each other and within same segment, if possible. Each dirty file gets allocated an extent and a new inode, referencing new data extent. Extent map gets updated with new inode locations. File inode contains a map of byte ranges in content space to extent ids and byte ranges within them. If file gets overwritten then chunks from existing extents are remapped to a new extent. With each file overwrite extents become more unused. To reclaim unused space, extents need to be read and chunks need to be moved to a new, more compact extent.

This procedure can both compact (reclaim disk space), defragment (reorder chunks), scrub (verify checksums).
Long runs of zeros can be represented without extents.
All metadata and selected data extents can be mirrored across devices, scrubbed. If scrubbing finds a damaged extent, then other copies are immediately pulled.

When an extent is no longer needed, it gets put back onto the 'awaiting wipe' list. Only after the freelist and underlying state structures were synced to disk, then these extents can be overwritten to disk. After a wipe, an extent is put back onto 'awaiting deallocation' list. After the wipe was persisted to disk the extent is put back to reuse. and is considered free space.

A special pathname "/???ioctl???" is used as a conduit for calling special operations. Opening it creates a new, independent channel. Writing to it is equivalent to calling IOCTL. Reading returns error code for last operation. This channel can be used for managing snapshots, setcontexts syscall, cloning files, etc.
A pathname can refer to a file as well, like "./dir1/file2?ioctl?"
Probably a file descriptor could be referred to by same way.

Data extent is divided into chunks (byte ranges), by default 128K
If a file gets written very slowly, or in many small non-contiguous chunks, then a file can end up fragmented into tiny chunks and tiny extents. However, these extents will sooner or later get consolidated during the scrubbing process. Internally, extents are comprised of chunks, which are data bytes followed by a checksum.
If data extent is encrypted, the crypto keys are stored in the file inode. When a file inode gets updated, old on-disk copy is always wiped after new copy is stored.
Entire data extent also has a checksum at the end, depending on all chunks within it.

If a volume is encrypted, then poddict contains the crypto keys needed to decrypt and authenticate poddicts, the choice of crypto and salted hash of the external key if it was encrypted by an external key, the choice of external-key input (keyboard, keyfile, QR code)!

cloning files: source file inode becomes marked as 'frozen', it cannot be changed anymore. It also has a reference count similar to hardlink counts. Dest file inode references the source file by id. From now on, opening the dest file will also load the source file inode. When the dest file gets closed, it also becomes frozen and its count is upped. The inode underneath it is not modified but it will be loaded whenever a descendent file is opened.

When a file is being compacted and defragmented: File inode is loaded and locations and sizes of all chunks are found. chunks are sorted by position within content space then by revision (generation id). Starting from left, each chunk gets trimmed or removed when surrounding chunks with newer generation overlap with it. Each chunk requires lookup of only two chunks ahead so processing the list is linear in time. When chunks were selected for preserving, they are concurrently loaded from disk, verified by checksum, decompressed, concatenated, and stored into a single new extent, which then gets sent to disk. Inode then gets updated with new chunk entries. There must be an algorithm that does not select chunks that are close to being compact and ordered. Another dict/ can be used to gather/accumulate number of bytes to be freed when rewritten. Chunks can be selected in order to have continuous groups of chunks larger than 8MB when segments are 16MB. Files could be postponed for rewriting when are still hot (less than 5min since last write) but not too cold either (more than two hours...).

Recovery: extent map values include a type (regular file, dir, symlink, data) and file inode contains parent directory ids (multiple when hardlinked). If directory inode cannot be read from any copy (many when mirroring) then files within it can be found by following extent map entries.

Cloning files: chunk map is moved to a new inode (with new id). current inode has empty chunk map and references the frozen inode same for the inode second hardlink. Back inode gets frozen but still can be compacted and defragged.

Snapshots create ids from same incremental counters as file writes and other ops. When a file gets compacted, overlapping chunks are only trimmed when there is no active snapshot between them. Live inodes need to keep all chunks until compaction, and directory inodes need to keep all file ids until compacting. Versioning simply provides a half range (rev, ∞) of ids that are newer compacted those to the left are compacted only when there is no snapshot id between chunks.

Large (16 MB) buffers are provided to the compressor and compressor splits the output into ~128 KB chunks. It is better to have the compressed stream split than splitting it before compression. First, it may decrease again from compression, second it will lead to too small chunks put into extents.

* Anonymous files require same infrastructure as replacing ordinary files. When a file becomes anonymous (when replaced or deleted) its id must be flagged as temporary (either within extent map or in a separate set. (Revisioning needs different solution)

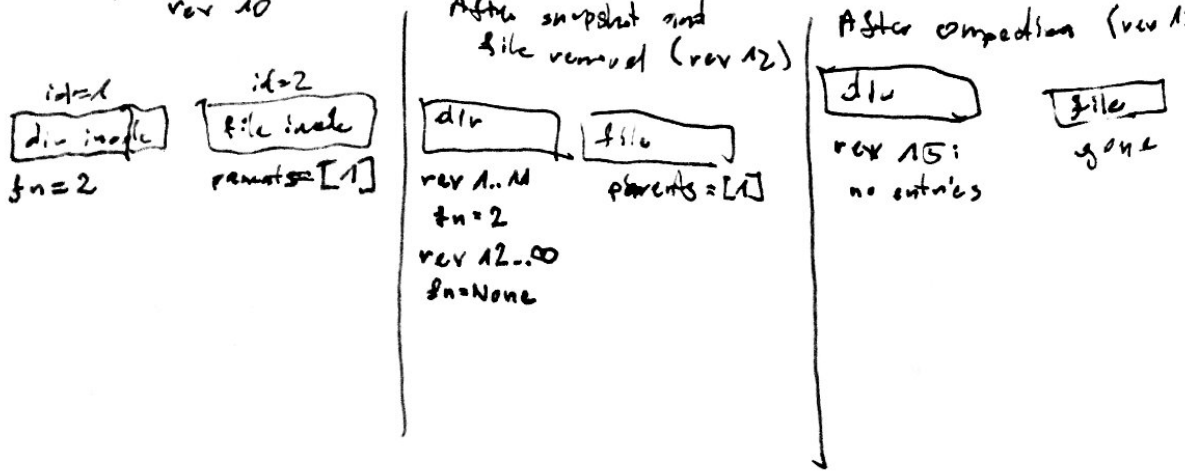
File can be opened at particular revision (read only) and then asked for compressed representation or its hash.

When actual file (current ongoing revision) is opened for hashing, a range of chunks must be frozen and thus withheld from compaction.

Compressed representation can be provided (also during fs sync at particular revision (no sense in updating file while compressed stream is provided). Chunks must be locked / frozen for certain revisions until compressed stream is closed. Alternatively all chunks can be locked. Locked means withheld from being removed. These chunks can still be compacted and defragmented and scrubbed.

Terminology: segment extent chunk id compact defrag scrub verify checksum extent info revision snapshot transaction

Removed/replaced (unlinked) files have complete inodes, that remain in the extent map. Instead its parent directory inode adds an entry that following current revision the filename no longer exists. During compacting process, the directory inode removes stale entries.



journal: Whenever an undoable or permanent action is taken an entry is added to a data structure that keeps track of everything that happens within the fs. This only includes user invoked syscalls, not internal compacting and such. Each user op gets assigned unique incremental id. The data structure is called a journal, with no connection to journaling internal in Ext3. Entries are grouped in chunks (at most one chunk per checkpointing window - few seconds), chunks grouped in chains / extents. Each chunk refers to the previous one, sometimes in a previous chain. When compacting, earliest chain is loaded, directory and file inodes are marked as requiring compacting. Many extents can be removed from the journal before inodes are compacted. Marking for compacting can be done through a flag in extent map. fs dict keeps ids of latest and newest journal extent. Earliest is used for knowing new history and compacting, newest is used for attaching entries. Also fs dict keeps Rev numbers of earliest and newest entries in the journal. Earliest is used during compacting (changes before that can be discarded) and newest is used for generating future ids.

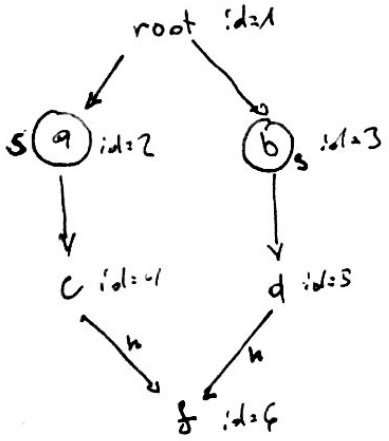
Truncation operation adds a special chunk info (that does not refer to disk location) specifying new file size. During compacting these special chunks are taken into account when deciding which chunks to discard.

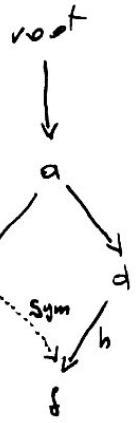
Transactions are snapshots that are auto-created on interruption. When a transaction is started, a new snapshot is created and marked / flagged as trans. fs dict contains a dict mapping snapshot ids to [state of file/dir ids. Every file and directory modified under a transaction is put into the dict. On commit, the dict gets cleared. On revert, which also happens during recovery, each inode from the set is loaded and reverted back to the starting version. Each transaction points to a set of modified inodes, their corresponding extent infos from before the snapshot, revision number of the snapshot, a set of allocated and deallocated extents. Ongoing user ops are recorded in these structures. Allocated space is recorded twice, deallocated space is recorded only here.

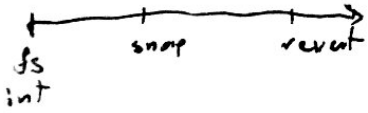
Removing directory trees: Just as when a file is removed, an entry is added to its parent directory inode that at certain revision the filename is no longer associated with any id (cargo does not exist). there is no reason why the same could not be used for removing directories.

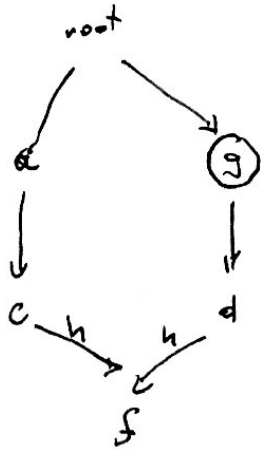
Terminology: file and filename, filename component.

Reverting: User selects a revision to which he wants to revert to. In fsdict, a range of revisions is added to the "reverted list, the range starting at selected rev and ending at current rev. Whenever the user opens a file or dir its chunkmap or filename map is filtered to not include revisions overlapping with the reverted list.









snapshots are enumerated in dict: key of snaps up, inode id affected, key to get back to, path used.

Reverting: Entries is added to affected inode about the range of revisions ignored (from selected to present). Whenever inode is opened, it inherits these entries from all parents. Hard-linked files open inodes for all paths to the root. Reverting can also be reverted, by adding a negative entry.

Transactions: inodes must be frozen for revisions starting at trans key in memory. Thawed when commited, or aborted. At most all untouched.

- ✗ When truncated to zero, maybe change file crypto keys. Replacing the file already does that.
- o Provide means to purge old file content when it's truncated not replaced. Change crypto keys without undo.