

# Raft: Storage Separation

Feb 25, 2025

Part of [Eng Design Doc: Raft Storage Efficiency](#)

## Background

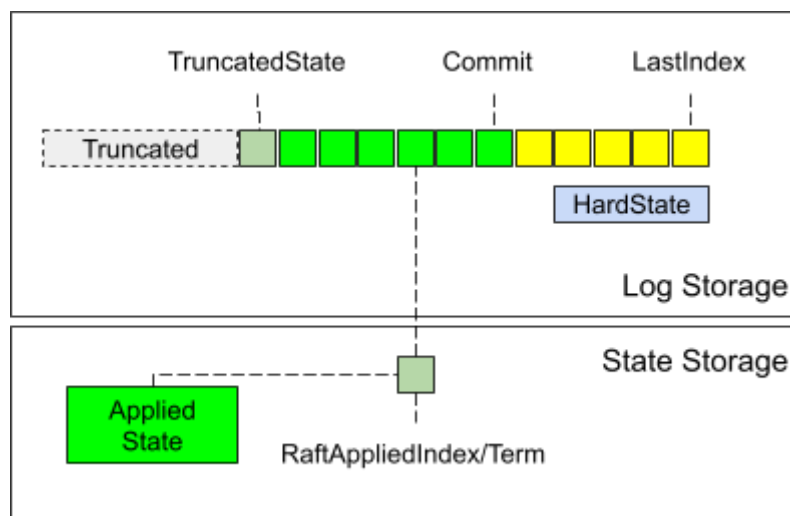
### Raft State Machine

The state of a single raft instance mainly consists of [HardState](#) and a suffix of the raft log. The log contains entries/commands in ([TruncatedState](#).Index, LastIndex]. A prefix of the log, specified by the commit index, is committed and immutable. The (Commit, LastIndex] suffix is mutable and may be wiped / overwritten multiple times by raft before it becomes committed. The latter can happen when there are leader changes.

The raft log content adheres to invariants mostly described by the [LogSlice](#) comment (except the part about being consistent with the leader's log):

- $\text{TruncatedState.Index} \leq \text{LastIndex}$ . The log entries have consecutive indices.
- TruncatedState is the ID of the entry preceding the first entry in the log (or the ID of the last entry if the log is entirely truncated).
- The terms of the entries are non-decreasing. The last entry term does not exceed  $\text{HardState.Term}$ .

Committed commands from the raft log are applied to the state machine asynchronously, after being declared committed. Command applications are atomic (with exceptions below), and maintain the entry ID ([RaftAppliedIndex](#), [RaftAppliedIndexTerm](#)) of the last applied command.



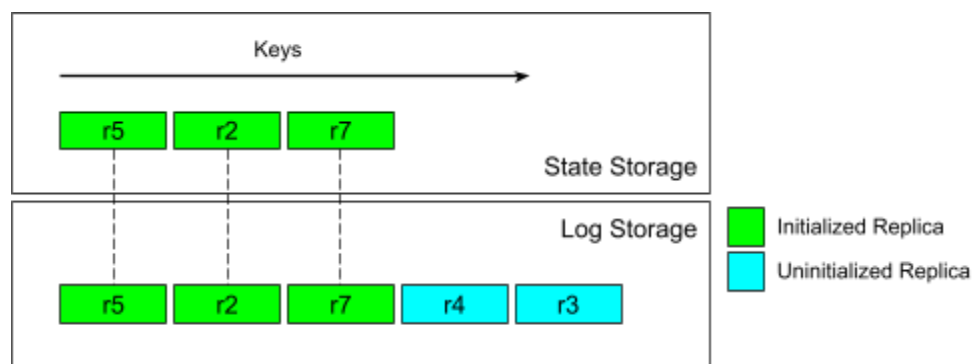
Once a committed entry from the log is applied to the state machine, it is no longer necessary to keep. Eventually, a prefix of the log is truncated. The log can not be truncated beyond the RaftAppliedIndex, because we must ensure continuity/durability of the entire raft state.

- $\text{TruncatedState.Index} \leq \text{RaftAppliedIndex} \leq \text{Commit}$

This describes a typical / steady state of a single raft Replica: the replica is initialized (after a split, merge, or snapshot application) and keeps committing / applying regular commands such as KV writes.

## Store State Machine

The state machine of a [Store](#) consists of many individual replicas, for different Ranges. Broadly, there are **initialized** replicas (those that have a [RangeDescriptor](#) and are placed in the “user” keyspace), and **uninitialized** replicas (those that already have a raft replica, but aren't associated with a key range yet). The key ranges of initialized replicas do not intersect.

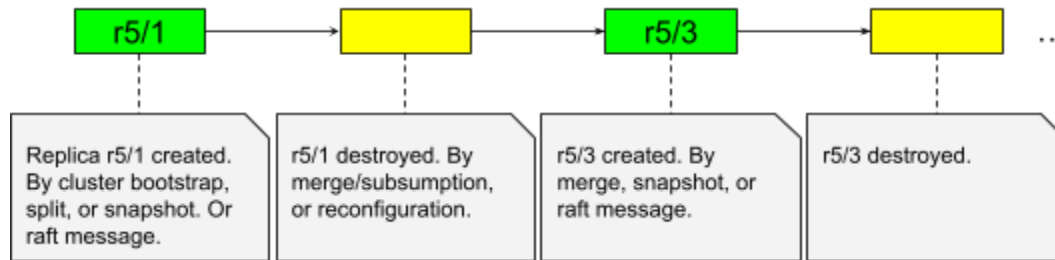


Uninitialized replicas perform a [very limited](#) subset of raft operations. They have an empty log (all indices are zero) and a non-empty HardState, and can only vote for a candidate / fortify a leader. Their purpose is to help bootstrap a new Range by electing its leader, e.g. after a split.

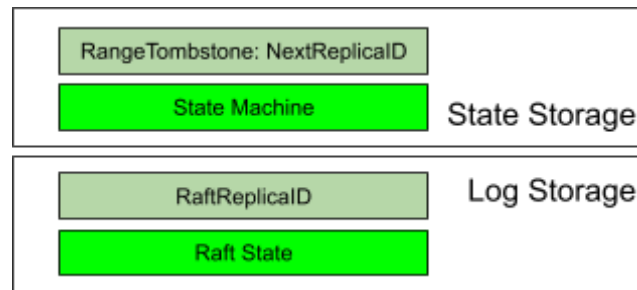
Replicas are operated independently / concurrently, i.e. each replica writes to the raft log and applies commands to the state machine as it sees fit. There is a limited set of “replica lifecycle” events (e.g. splits and merges) that may require coordination across two or more replicas.

## Replica Lifecycle

Replicas on the Store come and go. For example, a configuration change may remove a replica from one Store and place a new replica in another Store. For any RangeID, there is always at most one replica on a Store. It is possible that the same Store hosts a Range multiple times over the course of its lifetime, in which case the replica has a new/higher ReplicaID each time. The RangeID/ReplicaID pair thus uniquely identifies any replica in the system (corresponding to an instance of raft [RawNode](#)).



Talk about the [ReplicaID](#) and [RangeTombstone](#). For every RangeID on a Store, we have:



Invariants (TODO: approximation, needs refinement):


- $\text{NextReplicaID} > \text{RaftReplicaID} \Rightarrow$  replica does not exist (State Machine and Raft State empty)
- $\text{NextReplicaID} \leq \text{RaftReplicaID} \Rightarrow$  replica exists
  - Initialized if State Machine and Raft State exist
  - Uninitialized if only Raft State exists

Hazard: with split engines, there is no “replica ID” in the state machine. So there must be a careful sequence of sync(s) when removing a replica. We’ll get to it later.

Log storage is keyed by RangeID, but a replica is identified by a ReplicaID. The RangeID/ReplicaID pair is globally unique. From raft’s perspective, different ReplicaIDs are completely different, even if they correspond to the same Store (at different times). So we must be sure not to accidentally use/inherit the raft state that doesn’t correspond to the same ReplicaID. This is served by the [RaftReplicaID](#) key in log storage. Whenever ReplicaID is changed, or RangeTombstone passes this ReplicaID, the raft state must be cleared. It might be safe to “inherit” most of this state (e.g. the log and the term) across ReplicaIDs though, but we don’t consider it for simplicity.

Events to cover:

- Initialization
- Log append
- Snapshot
  - +replica subsumption
- Truncation
  - sideloaded truncation

- Split
- Merge
- Removal
- Sideloading:  Raft: Sideloaded Log Storage + Pebble

For all events, identify operations spanning log/state storage boundaries, and define the order in which they must happen. With a single engine, they occur atomically. If engines are separated, there must be a sync/flush between the two parts, and the corresponding recovery procedure in case of unclean restart. For each sync, identify whether it must occur immediately or asynchronously. As an example: there must be a flush of the State Machine before the applied entries can be truncated from the log. This, however, is not on a critical path, so it is fine to passively wait for when it happens, and then truncate.

## [ReplicasStorage](#)

### Snapshots

A snapshot can initialize a replica (i.e. move the state machine from “empty” to a nonzero applied index for the first time), and it can move an already initialized replica forward to a larger applied raft log position.

The latter kind of snapshot deserves more attention due to the need to clear existing data and deal with the fact that the snapshot may skip an arbitrary number of splits and merges.

Examples. Assume a range  $[a, f)$ .

- a snapshot may arrive that reflects the application of multiple splits. The snapshot may cover only  $[a, b)$ , which are the new extents of the range at the snapshot's log position.
- a snapshot may arrive that reflects one or multiple merges: the snapshot may cover  $[a, x)$ . (There can be more than one merges because we don't wait for application on the LHS, only the RHS. So, there can be a lagging follower that hasn't applied a bunch of merges.)
- A snapshot may reflect a combination of splits, merges, and replication changes. Replication changes shouldn't complicate things much.

A store maintains an always-correct view of which keypace is owned by which Replica. Keyspace may be assigned to no Replica, but can never be shared between more than one. Maintenance of these invariants is complicated in practice, and there have been bugs in this area in the past, often related to range bounds changes during snapshots. This is not strongly related to the storage model of snapshots, so for the most part, we gloss over it here and simply posit that in the context of a snapshot, we may always assume that the Store isolates our keypace sufficiently from concurrent snapshots or other replicas. Snapshot processing also ensures that each snapshot only ever moves the applied raft index strictly forward (offending snapshots are dropped, though we may [use](#) their commit index). And finally, a snapshot always reflects a RangeDescriptor that the receiving replica is a member of (upholding this invariant that holds in general, not only for snapshots, but for all Replicas instantiated on a Store).

Nitty gritty:

lead-up:

- the snapshot sender creates a snapshot in `(*Replica).GetSnapshot` ([ref](#)). There will need to be additional synchronization between two engines once we have them (likely `raftMu` lock while grabbing `HardState` or sth like that). Snapshots can be delegated, and there are bugs ([#127348](#), [#127349](#)) related to the raft message created for snapshots when on non-leader replicas.
- the actual sending happens in `(*kvBatchSnapshotStrategy).Send` ([ref](#))
- the opening RPC for the snapshot contains raft log position and term (i.e. it's like an append that the receiver can assume is committed, i.e. no need to check leader)
- the snapshot contains all replicated spans, i.e. the entirety of the state machine data
  - caveat for shared storage, in which case we don't contain the user data (does this matter for sep raft log? added to proj tracker)
- the snapshot (once validated and received) enters the destination via `Store.processRaftSnapshotRequest`. This method steps the snapshot metadata into the `RawNode`. If the `RawNode` wants to apply it (which it typically wants to if it moves the applied index forward), we call `handleRaftReady` (passing in some snap metadata). Because this happens under a contiguous `raftMu` section, we know this ready cycle will be the one to handle the snapshot.
- we would like to know that the snapshot will come up for application in isolation, i.e. we're not going to see any attempts at applying entries or appending to the log in this ready cycle, which would enlarge the state space we have to consider. At the time of writing, this is not true: <https://github.com/cockroachdb/cockroach/pull/125530>

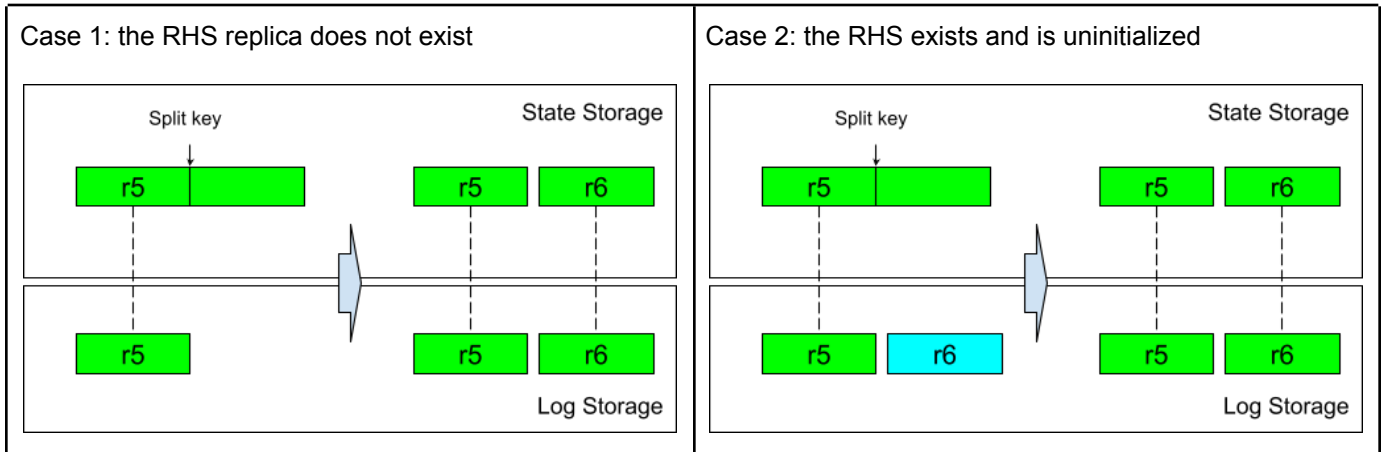
here it gets more involved.

- If the snapshot shrinks the key range, we clear the delta and apply the snapshot
- If the snapshot extends past the right hand side of the current bounds of the replica, the snapshot represents a merge (or multiple merges, splits, etc)x5. We may have an initialized right-hand side locally (TODO: don't we always, due to `WaitForApplication?`), but we don't want to keep any of that because the snapshot has what we know is the correct contents of our left-hand side at the log position, replacing anything we have on the right-hand sides.
- so we need to nuke the right hand sides. This is prepared in `maybeAcquireSnapshotMergeLock` ([ref](#)), where we grab the `raftMu` for all of those replicas. These replicas are called the "subsumed replicas" below.
- now comes the actual data ingestion. The snapshots were prepared as `SSTables`. (There is an optimization to ingest small snapshots as a `WriteBatch` instead, and we'll ignore it for now but it will need consideration too).
- we create an SST that rewrites the raft state (`writeUnreplicatedSST` -> `rewriteRaftState`). This
  - clears the log (i.e. it's a full truncation)
  - clears `HardState`
  - writes the updated `HardState` (emitted by the `RawNode`)
- then in `clearSubsumedReplicaDiskData`, for each subsumed replica:

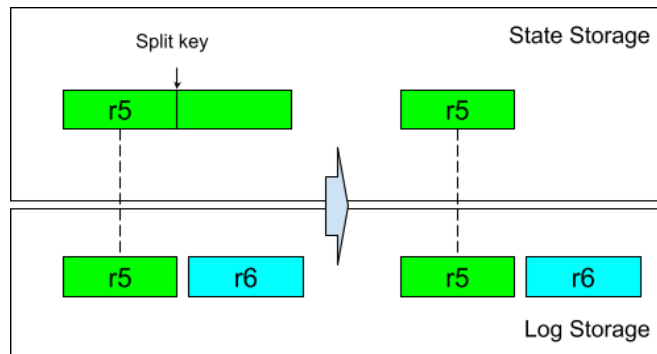
- create another SST (subsumedRepISSTFile) that clears both the replicated and unreplicated RangeID keys (so this nukes the raft state, but also touches the state engine - no good with sep raft log) via DestroyReplica.
- NOTE: [this comment](#) describes a case in which a snapshot may subsume a replica, but that subsumed replica's end key is further out than the snapshot's end key. It seems possible for this to happen, but it contradicts the comment in maybeAcquireSnapshotMergeLock which claims that the right boundary always has to line up. In case they don't we make more SSTs to clear the parts of the range that extend past the snapshot, too.
- I think all of this untested, geez
- now we create another SST that clears the replicated span-addressable data in the "widened" bounds (as per last comment), so this clears the RHS range(s) state machines.
- we now have a set of SSTs, and atomically ingest them (which we can no longer do in sep-raft-log, not to mention the fact that some of the SSTs probably span both engines).
- also sometimes we use pebble's "excise" instead of range deletions,

## Splits

TODO: all cases



### Case 3: the uninitialized RHS exists and is newer




How splits work:

- AdminSplit request
- Runs a [distributed transaction](#) that
  - Changes the end key of RangeDescriptor to be the split key
  - Writes a new RangeDescriptor for the RHS
  - Updates routing information in meta ranges accordingly
  - Commits with a [split trigger](#) annotation
- what's in the split endtransaction write batch ([src](#)):
  - LastReplicaGCTimestamp (not too important)
  - copy of AbortSpan of LHS
  - WriteInitialReplicaState is called (sets a raft applied index/term, LAI, desc, lease, stats, gcthreshold, gchint, version)
    - but also initializes a raft truncated state - which is really part of instantiating the raft state. This should not be done here but below raft
    - I added an item in our proj spreadsheet to think through how version migrations need to change / which kinds of migrations are harder with separate engine.
- When applying a command with a split trigger
  - first we enter splitPreApply ([see here](#))
  - we start out with roughly the write batch made during evaluation (full lease applied state, also sadly a truncated state in there)
  - should really have a datadriven test that prints out what exactly we've got (added to proj sheet)
  - check for right hand side replica: if it does *not* exist or has a higher replicaID than implied by split, we will discard the RHS of the split and leave the RHS alone:
    - right repl is raftMu locked (if it exists)
    - call clearRangeData on RHS (into our write batch)
    - awkwardly, it also nukes the RHS raft state but then re-adds it (need to back that out - we should not touch any raft state here at all, it's purely a

- state machine write - we're basically turning the "split" into a "delete my right half state user key data" operation)
- if not in the special case above, we instead have an (uninitiated) rhs replica at expected replicaID and raftMu is held.
    - we call SynthesizeRaftState (into our WriteBatch), which (essentially) updates the Commit field to a nonzero index (reflecting receipt of the right hand side of the split)
    - also call SetRaftReplicaID and persist a closed timestamp
    - the WriteBatch already contains a LeaseAppliedIndex etc, as this is all written during evaluation
  - the batch is committed
  - then we enter splitPostApply, which does in-memory management mostly to update the Store with the results of the split (shrink LHS desc, init RHS)
    - if we're in the "special case", bails out early since RHS replica is not there/is not to be touched. Otherwise:
    - this does call LoadReplicaState on the RHS to load the state (fine, just maybe want this libarized more, this loads the truncated state too which should be factored out)
    - calls Store.SplitRange to finalize in-mem updates, no engine use there

## Merges

Digest relevant details:

- [Tech note](#)
-  Range Merges and RACv2

How merges work:

- AdminMerge request
- Runs a [distributed transaction](#) that
  - Reads LHS and RHS descriptors
  - Verifies the merge is possible
  - Updates the LHS end key to equal the RHS end key
  - Removes the RHS descriptor
  - Sends a subsume request to the RHS (which is now a [write](#))
  - Waits until all RHS replicas **durably** apply the subsumption command (which is going to be the last command in the log? not)
  - Commits with a merge trigger annotation
- Commit trigger contains:
  - Copy over the abort span to the LHS
  - Copy over the read summary from the RHS. That's because the leaseholders aren't guaranteed to be colocated, so we might not have timestamp cache information from the RHS on the LHS leaseholder's store.
  - Range key fragmentation; force flush. Unrelated to us.
  - Loads the GC hint.



- When applying a command with a merge trigger, we have guarantees:
  - The trigger is on the LHS replica
  - The RHS replica is colocated with the LHS
    - Can it be already gone / replaced by a later replica? No because there is still an intent on the RangeDescriptor.
  - All commands in the RHS are applied, so nothing to wait on here (could be relaxed: wait for commit index sync up the stack; then wait for application here)
- The merge application
  - [Holds](#) the RHS raftMu
  - [Removes](#) the RHS raft and state machine data, but leaves the “user keys” in place. These are inherited by the LHS atomically with the txn commit and RangeDescriptor changes.
  - That’s it?? (obv, there is in-memory synchronization between the replicas, but not much happening storage-wise)

## Scratch Ideas

The lifecycle of a RangeID on a Store encompasses multiple incarnations of this RangeID (under different ReplicaIDs). We want a unified coordinate system for this RangeID, to reason about durability and order of things across two engines.

A “watermark” that completely describes the lifecycle state of the RangeID: **(ReplicaID, commit index)**. In the context of the state machine engine, the commit index == applied index. When a replica is removed, the watermark is a **tombstone** that is logically in between **(ReplicaID, max index)** and **(ReplicaID+1, 0)**. The watermarks compare lexicographically, as tuples. Having this watermark both in log and state storage trivially prevents accidental log/state mismatches described [here](#).

The tombstone isn’t perfectly replayable in this form. An out-of-band DestroyReplica can be executed in the middle of the committed log sequence. If we want perfect replay, these out-of-band ops need to be interleaved into the log coordinate system (stored alongside the replica’s log, or in a “global” one). More speculation on this idea a few paragraphs below.

During runtime, the 2 watermarks tend to diverge / are asynchronous. Typically, the state machine one follows the raft one: for example, raft entries are known to be committed in the raft engine, but not yet applied to the state machine. It can be the other way around as well: a replica destruction has been applied to the state machine (and written a tombstone) first before doing the same in the log engine.


Let the log storage watermark be **(id1, c)**, and the state storage watermark **(id2, a)**.

Easy to see: there are exactly 8 cases of asynchrony between the 2 tuples. These can be squashed to 4:

1.  $(id1 == id2) \ \&\& \ (c > a)$ : there are unapplied committed entries. We just keep applying.
  - special case:  $c$  is a tombstone. Remove the state machine (“apply” this tombstone)
2.  $(id1 == id2) \ \&\& \ (c < a)$ : the applied state ran in front of the committed. Can happen if the commit index is not synced before entries are applied. We just recover and keep applying after index  $a$ . We may assume  $c := a$ .
  - special case:  $a$  is a tombstone. Remove the log state.
3.  $(id1 < id2)$ : replica  $id1$  already removed from the state machine, but remains in the raft storage. We proceed and rewrite/initialize the raft state to match  $id2$  mark.
4.  $(id1 > id2)$ : there is an uninitialized ( $c == 0$ ) replica in the raft engine, newer than the state machine one. Remove the state machine. Be sure not to introduce witness replicas here ( $c != 0$ ).

Better idea below.

<https://cockroachlabs.slack.com/archives/C02KHQMF2US/p1740599708583879>

Latest:  Store State Machine

Oplog idea:

- Have an “oplog” / WAL for all replica lifecycle events on the Store (inits, splits, merges, snapshots, replica removals, etc). Put it into the log storage which we’re syncing all the time.
- Before applying any such event, append it to the oplog and sync.
- Apply the durable oplog commands and “applied oplog index” in one batch. Can do combined batches across multiple oplog commands / replicas. State machine commands from the oplog must be applied in order. Could sound like a downside, but note that Pebble already sequences all writes, so we just need to make sure it does it in the exact order we like. The oplog is not very contentious, so sequencing it shouldn’t be a hard problem.
- Asynchronously, after the applied oplog index is flushed, truncate the oplog.

Assumption:

- All ingestions are flushable. So, snapshot and AddSSTable applications are durably sequenced with all other state machine writes. This is not the case today, and the only gap. Without it, we need state machine WAL/syncs and bespoke replay [strategies](#).

Statement:

- A combination of the applied oplog index and raft applied indices for all initialized replicas deterministically defines the current applied state.

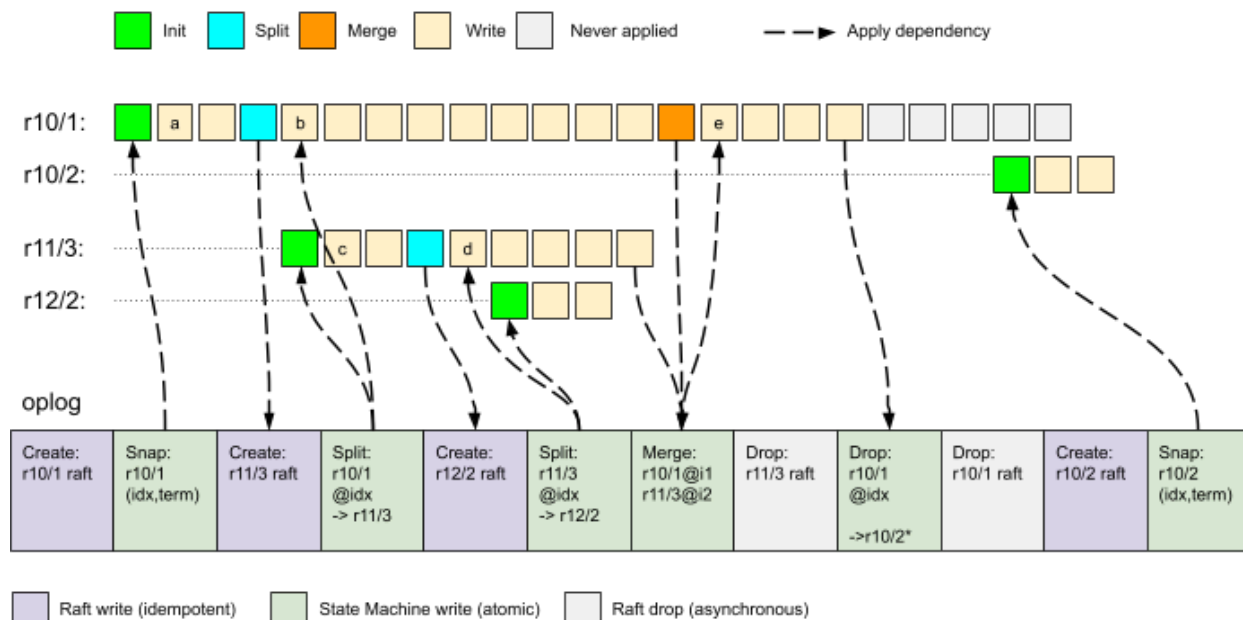
Recovery:

- Replay the DAG of the unapplied oplog, interleaved by raft logs where there is a dependency. In the process, we will reach a state consistent with shortly before the restart.

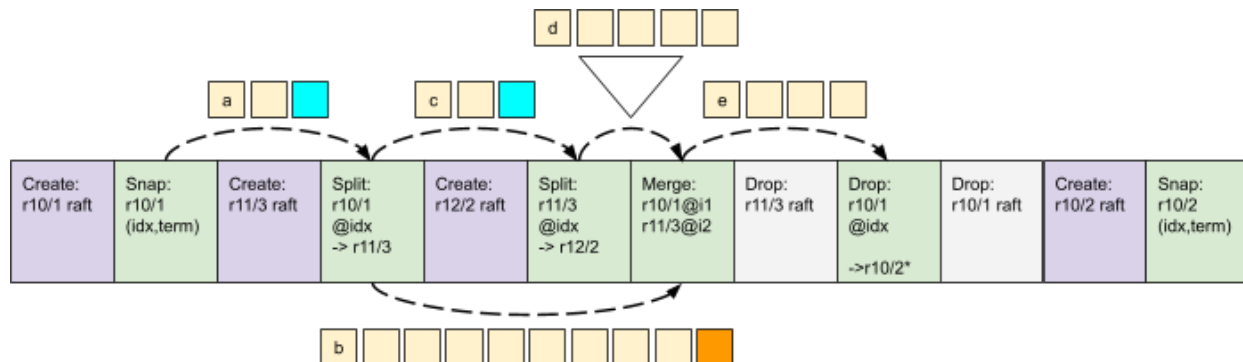
- This is only needed because there are commands in the oplog writing raft state to the raft storage. These must be replayed idempotently, to bring the raft and state storages in sync.

For example, DestroyReplica can log the (ReplicaID, applied) index of the raft log that it “depends on”. When replaying, we won’t replay the raft log further than this index even if the log has more.

The DAG of command applications to the Store state machine:



The “apply” DAG during the replay is fully defined by the oplog. A simple implementation would read the entire unapplied oplog (typically short or empty), and build all the dependencies in memory. This produces the “plan” of the replay - a mini-DAG of the oplog commands, with concrete raft log slices attached to the dependency “edges” between them. For the above example it looks like:



TODO: snapshots with subsumption.

After the oplog is replayed, the runtime replicas are started, continue applying the rest of the commands in the “online” mode, and appending to the oplog when new replica lifecycle events happen.

The oplog is the answer to the [question](#) of how to untangle the “pure” / “standalone” command application from the in-memory state in places where there is such a dependency. For example, when applying a split, we [source](#) the RHS closed timestamp from Replica and CT side transport. We just need to include this ephemeral information into the “locally evaluated” oplog command.

TODO: build a minimal set of examples explaining the key design decisions. Plan:

1. Assumption 1: state machine is separate from the raft engine. Can not update both atomically.
2. Assumption 2: state machine has no WAL and flushes infrequently.
3. Scenario: one range applies a snapshot and rolls the raft state forward.
  - If the state machine regresses, we have no way to roll back the raft state and log.
  - If the log state regresses, we could synthesize it. But we should be able to infer it from the state machine, and it might not be perfectly replayable.
  - Solution: save the snapshot application “command” to log storage entry. Replay on restart.
4. Scenario 2: leading to having per-ReplicaID raft storage.

Minimal example:

1. r10/1 commits and applies a few commands.
2. r10/1 is removed by replica GC.
3. r10/4 is initialized by a snapshot.
4. r10/4 commits and applies a few commands.
5. A crash/restart rolls back the state machine to before step 1. We lost the snapshot and are unable to recover r10/4. So we need to have the snapshot stored and replayable.

Why need to key raft state under ReplicaID:

1. r10/1 applies a few commands.
2. r10/1 splits -> r11/1 is created.
3. r11/1 applies a few commands.

Implementing the oplog requires an overhaul of the entire replica lifecycle code surface, but looks highly beneficial. The entire Store state machine becomes well-defined and deterministically testable. A wide set of crash/durability [tests](#) can be implemented without running a full TestCluster, or infecting the code with various knobs that help emulating crashes in particular places (of which there are many).

Producing [such a test](#) today is a significant challenge: we have to inject a bunch of callbacks and synchronization to force specific sequences of events, fight with moving parts of a Replica / Store, raft messaging / transport, gRPC / network, circuit breakers, etc. In the end, the test only covers one carefully crafted scenario (of many possible), and is [flaky](#) because it's hard to account for all corner cases.

In contrast, with a deterministic Store state machine, producing such a test would boil down to: (1) writing an automatically validated sequence of commands (e.g. as a datadriven test file, or generated in code), (2) starting a TestServer/TestCluster from this initial state and making sure it doesn't crash, (3) running a few follow-up writes to make sure the cluster is operational.

The oplog is a “flight record” of the Store state machine. A simple disabling of the oplog truncations (and, if needed, raft log truncations and removals) in a running cluster results in having a replayable history of the entire Store / cluster operation. This can be a great help in reproducing and fixing bugs in KV and Storage, or generating “golden” tests for catching regressions in the state machine determinism (particularly in mixed version scenarios).

The “flight recording” reduces the effort needed to write a crash test: instead of writing state machine commands manually, take an existing recorded run, and modify it slightly (e.g. insert a crash in the right place). This allows exhaustive testing: insert crashes in all possible places automatically.

Even without storage separation and performance benefits, this is Quality work with a big Q.