Store State Machine

Feb 28, 2025

Part of Eng Design Doc: Raft Storage Efficiency

Overview

This document presents a high-level description of a design for the WAL/sync-less **Store State Machine** in CRDB [#38322]. It provides a mental model for the Store operation, and can be seen as “vision” that helps us make design choices leading in a coherent direction.

We consider the **Store** being a state machine that encompasses the **lifetimes** of all **replicas** that it hosts. The Store is **logically** separated into two storage engines: the **log engine** and **state machine engine**. The log engine is thought of as the “WAL” of the state machine, and the state machine engine contains the result of “applying” the **committed** commands/transactions from this WAL. We abstract from the physical implementation of engine separation, which is achievable in multiple ways: by actually having separate engines / on separate devices, or one engine tuned differently for two non-overlapping subsets of keys.

Assumptions about the logical engines:

- The log engine is synced frequently, with low latency. Commits to this engine are on the critical path of the cluster operation, most notably all writes into the database.
- The state machine engine is flushed relatively infrequently, and there is no runtime requirement for its data to become durable with low latency.
- Writes to the log engine are totally ordered. If a write batch is known to be durable, then all the preceding writes are durable too.
- The state machine engine provides the same ordering/durability guarantees. This applies both to regular writes and ingestions, and to ingestions combined/interleaved with writes.
- There is no ordering guarantee across the two engines.

For the most part, the operation of each replica is independent of / concurrent with other replicas. Each replica is a member of a raft consensus group, and operates according to the raft algorithm: appends and commits entries into the raft log, and applies the corresponding commands to its state machine.

The **raft state** of a replica, encompassing the raft log and metadata such as **HardState**, is contained in the log engine. The **applied state** of a replica is contained in the state engine, and is associated with a particular user keys range (defined by the **RangeDescriptor**).

Throughout its lifecycle, the Store handles **replica lifecycle** events that may change the **replica set** of the state machine, modify the set of raft instances hosted by the log engine, or change the content / bounds of replicas. These events can be thought of as **transactions** across the log and state engine, performing one or multiple primitive operations. The (potentially incomplete) list of such operations:

- Creation/deletion of raft state.
- Assignment of raft state to a replica and the corresponding part of the key space.
- (Re-)writing a replica state by ingesting a snapshot.
- Applying committed raft commands to a replica state machine.
- Changing a replica's metadata / key range.
- Splitting and merging replicas.
- Deletion of a replica.

For example, when a replica receives and applies a snapshot, the Store performs a transaction that:

1. Writes the new raft state, with the raft log initialized at the commit index of the snapshot.
2. Rewrites the replica data in the state machine with the content of the snapshot.
    - This may involve changing the replica's metadata / key range.
3. Removes the old raft state for this replica.
4. It may also "subsume" other replicas, but let's not consider it in this example.

In today's single-engine CRDB design, this transaction is performed atomically via a single ingestion into Pebble. With split engines, this can no longer be achieved: step 2 is written to the state engine, and steps 1 and 3 go to the log engine. The key task is in making it look atomic without compromising the correctness and durability of the Store state machine and its individual replicas.

Intuition

We make a few key observations about such cross-engine transactions under the given constraints, on the snapshot application example. It turns out that it's a general pattern:

1. Log engine write. Durable.
2. State machine write. Eventually durable.
3. Log engine deletion. Asynchronous, conditional to step 2 durability.

Step 3 is unsafe before step 2 is applied durably. Otherwise, if the Store crashes / restarts, it may observe the applied state preceding this transaction, and thus may need to replay committed commands from this raft log (and other logs "downstream" of it, e.g. after splits) to catch itself back up; but the necessary log is already deleted by step 3. We thus **delay** step 3 until after step 2 is durable.

Steps 1+3 can not be done atomically (even though the log engine allows it). Step 1 is on the critical path of the replica operation, and can not wait for step 2 durability because the state engine does not provide it with low latency. So we issue **log engine writes first**, and **log engine deletions are asynchronous**.

Steps 1 and 3 have a “conflict” / data dependency on the raft state, and we just saw they can’t be done atomically. We thus adopt a form of optimistic concurrency control / MVCC: place the new raft state under a new key (**LogID**), while the old one still exists and is readable (by replay). Logically, the old LogID is removed instantly and is not accessed at runtime.

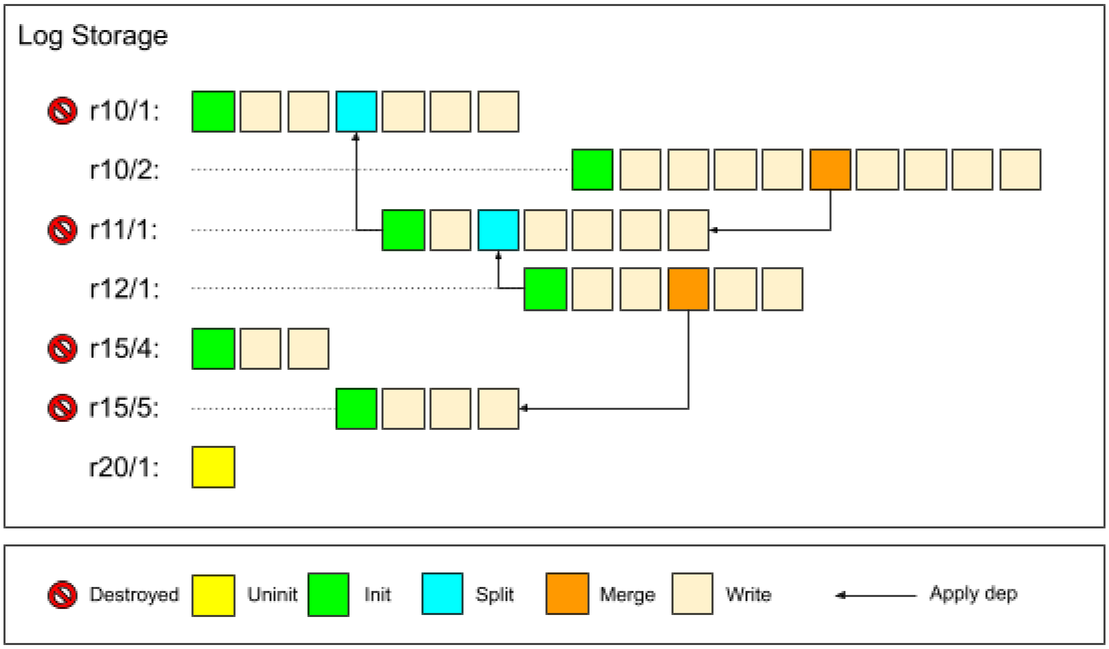
We observe that the write set of this **transaction** needs to be **persisted**, so that it can be finalized in the presence of crashes. Say, steps 1 (durable) and 2 (non-durable) are done, and then the replica commits a few raft entries. If there is a crash/restart, we now have a non-empty raft state, but the snapshot application initializing this replica was lost, so the replica is inoperable.

The write set is persisted atomically with the log engine write (step 1). In our example, it references an SST file on the state engine, so that the ingestion can be done locally on it. We should mention that the state engine should provide such a capability: (a) an interface for storing SST files durably, and (b) ingesting these files into the state machine. We do not require the log engine to provide ingestions.

Finally, note that the Store operation is **serializable**. For example, two replicas can be applying logs concurrently in any order, and the end state is the same. A snapshot initiating a new replica is applied before any command for this replica is applied, and, generally, there is precedence / dependencies between commands overlapping in the keyspace. To preserve the serializability across crashes / restarts, the **serialization** graph of these transactions is captured durably in the log engine.

WAG: The Write-Ahead Graph

At this point the reader starves for diagrams. Let us consider an example of a Store operation. To make it interesting, let it have splits / merges, ReplicaID changes, etc.



We observe that state machine changes form a DAG. The “edges” in this graph define data dependencies between the changes. For example, a single replica’s log must be applied in order. When a range splits, the LHS and RHS replica can continue applying commands concurrently, until/unless they need to be synchronized again (say, there is a merge for r10/2 and r11/1).

Assume that the state engine hasn’t done a single flush. In case of crashes / restarts, we must be able to restore the state machine from the information present in the log storage. So the log engine has the entire history: replica lifecycle events, current and past replicas, and their raft state / logs.

Recovery boils down to **replaying** the DAG. If the dependencies are met, the end state is deterministically defined, and does not change if some concurrent commands are applied in arbitrary order. For example, commands of non-overlapping replicas may interleave arbitrarily, and the end result is always the same.

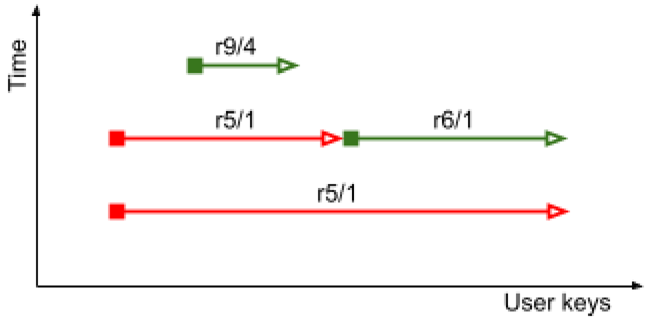
First, we should be able to figure out the current **replica set**. For each **RangeID**, the current **ReplicaID** is stored in a RangeID-local key in the log engine. If all the replicas of a RangeID have been destroyed, this is reflected in its **RangeTombstone** key. For our example, the replica set is {r10/2, r12/1, r20/1}, and other replicas are destroyed. Next, we replay all the dependencies leading into this state.

- r20/1 is uninitialized, i.e. it only contains raft state, and is not linked to a RangeDescriptor and the user key space. There is nothing to do with it during recovery.

- r12/1 was initialized by a split of r11/1, so recover the pre-split state of r11/1 first. For that, we must initialize r11/1 and replay it up to the split command. To initialize r11/1, we, similarly, must initialize r10/1, and replay it to the split command.
- r15/5 is destroyed, so we also want to cause the r12/1 replay at least up to the Merge command.
- r10/2 was initialized by a snapshot, and has no incoming dependencies (it actually has, wait for it), so we can apply the snapshot and initialize it. We then replay it up to the Merge command. It has a dependency on r11/1, just like r12/1, so both will wait for it to be replayed first.

There is a catch. The DAG we considered only reflects split / merge dependencies recorded in the logs. To see that this is not sufficient, let's try to initialize r10/2 first, and then replay initialization of r12/1. The latter unwinds back to r10/1, which overrides the r10/2 state that we already replayed. Seemingly, the solution is to introduce a dependency r10/2 -> r10/1 (later states of the same range depend on its past state removed first), and modify the replay sequence accordingly.

It turns out that this is only a special case of a more general type of dependency. Consider the following sequence: (1) r5/1 splits and creates r6/1, (2) r5/1 is removed from the Store, (3) r5 splits a bunch of times elsewhere in the cluster and creates r9, (4) r9/4 is added to the Store.

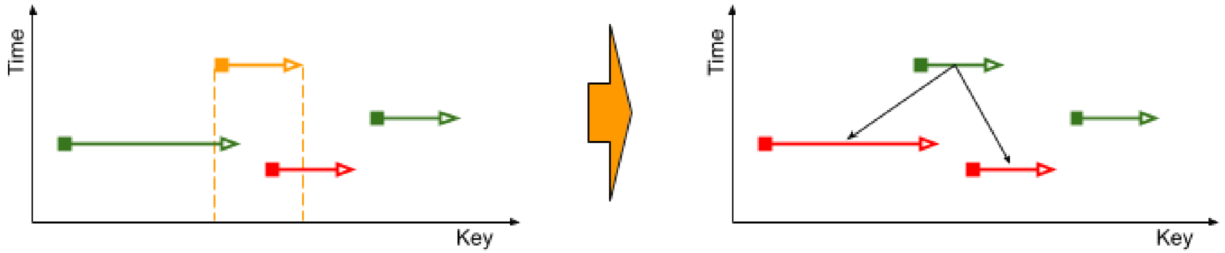


We can not replay r6/1 and r9/4 in arbitrary order. If we initialize r9/4 first, and then replay the sequence leading to r6/1, we wipe the r9/4 data. Or there will be an error while replaying r5/1 because it conflicts with the already existing r9/4 in the key space.

A replica can be replayed only after the replayable replicas "below" it in this diagram have been destroyed. In the example, an acceptable sequence is: (1) init r5/1 and replay it up to the split, (2) destroy r5/1 (3) replay r9/4 and r6/1 in any order.

To understand this type of dependency, imagine the "lifetime of a key" on a Store. It begins when the key is first covered by a newly initialized replica (necessarily a snapshot). Then there can be a sequence of splits / merges changing the owner of this key. These can be interleaved with Destroy + Init pairs of events where the key is removed from the Store, and then re-enters again via a snapshot application.

Observe that the split / merge dependencies are already covered in the example DAG. The missing bit is the Destroy <- Init dependency between consecutive (in time) replicas with overlapping RangeDescriptors. We already track this kind of information at runtime: the Store maintains a non-overlapping set of RangeDescriptors and "replica placeholders", and uses it to find replicas subsumed by a snapshot. With some tweaks, this data structure can help us observe all the necessary DAG dependencies.

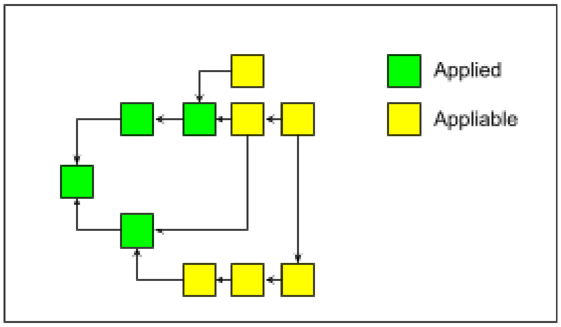


Specifically, we would need to remember the "frontier" of the destroyed key spans for a bit longer, until the destruction is durable on the state machine engine.

TODO: Probably there are some words involving "linearizable" / "serializable" that can be said to explain things here. All we care about is that the whole Store execution is serializable, and we get that by linearizing access to each state machine key/range, and remembering all the dependencies.

Checkpointing

We considered the case in which the state machine engine has never flushed, and so we had to recover it entirely from the information in the log engine. In reality, full replay is infeasible. But the state machine engine performs flushes regularly, so most of the state machine is already persisted. We should be able to restore the latest state by applying the "delta" from the log engine.



The applied state machine is a materialized "prefix" of the WAG. The replay procedure should start exactly at this prefix, and finish exactly at the one defined by the log engine. The state machine and log engines must have a shared “coordinate system” which allows finding out the delta and checkpointing progress.

The checkpoint is derived from the set of **RangeID**s, and RangeID-local information about their progress. Each RangeID has its current **ReplicaID**, **LogID**, and **RaftAppliedIndex**. If the last replica for the given RangeID is destroyed, these keys are empty, and **RangeTombstone** reflects the ReplicaID up to which everything is removed. For a merged range, the RangeTombstone is “infinity”.

When applying commands to the state machine, we update the applied “checkpoints” in the same batch atomically. For example, applied raft commands bump the **RaftAppliedIndex**, and replica creations / snapshots / destructions bump the **ReplicaID**/**LogID**/**RangeTombstone**.

Garbage Collection

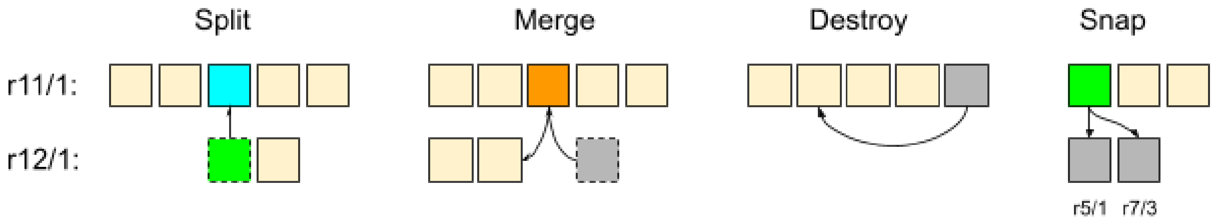
When the state machine flushes / persists the applied state, the corresponding WAG prefix in the log engine becomes obsolete and can be garbage collected. For example, raft logs can be truncated to **RaftAppliedIndex**, and raft state for LogIDs below the “applied” LogID can be removed. This state is already “logically” removed and won’t be accessed at runtime or by any future replay.

The deletion does not have to be instant. For example, the raft logs of an active ReplicaID/LogID can stay around for longer, so that the log is replicated to lagging followers (otherwise, they would need to receive snapshots to be caught up). There should be heuristics on when we truncate, minimizing the snapshots traffic and balancing that against log engine performance.

For garbage-collecting at runtime, we would have a queue of removable **LogID**/entries enacted when the corresponding “low watermark” is durable. Similar to how it’s implemented in loosely-coupled raft log truncations. It might be convenient to have a unified Pebble-wide “durability mark”, otherwise (doable) we would need to perform a read-back from the durable state, for each mark type. For example, Pebble could provide the sequence number of the memtable that a state machine batch went to.

WAG Nodes

For each **LogID**, we are only interested in 2 events: **initialization** and **destruction**. Initialization happens either by a split (statelessly), or by a snapshot (statefully). Destruction, similarly, is either stateless with a merge, or stateful (by replica GC or subsumption).



Replica destruction is either stand-alone, or atomic with a merge / subsumption. Atomicity is achieved by writing linked WAG nodes in a single batch. For example, in the snapshot with subsumption case: even though the Destroy and Snap are distinct nodes, and will be replayed in sequence, the fact that we committed them in one batch means the replay will have completed both and in the right order.

The Init and Destroy “nodes” of the WAG can be persisted in two LogID-local keys.

Recovery

The recovery is a classic Computer Science problem of topologically sorting / replaying a DAG. Broadly, we have a set of initial nodes (defined by the state machine engine watermarks), and a “target” set of nodes to reach (defined by the log engine watermarks). We replay everything in between, by following the dependency “edges” in this graph, and making sure that any command is applied only after visiting all its dependencies.

For a RangeID with an initialized replica, the target is its **Init** node (could be a **Split** or **Snap**). For a RangeID with no initialized replica, the target is the last LogID’s **Destroy** node (could be a **Merge**). Note that the replay can overshoot some of these nodes. For example, after reaching **Init**, we might be bumped to a higher raft applied index by a dependant whose **Init** waits for our **Split**. Or a replica can be past the **Init** to begin with (typical case).

Python

```
goal = {Init for all initialized replicas} + {Destroy for all the rest}
for node in goal {
        dfs(node)
}
func dfs(node) {
        if appliedPast(node) { return } // comapare the "watermarks"
        for dep in deps(node) {
                dfs(dep.node)
        }
        // all deps are fulfilled
        applyTo(node) // apply a split/merge/etc, or a slice of a raft log
}
```

There can be “concurrent” branches of the DAG, so the replay can be implemented as a worker pool, applying some branches in parallel. In a typical case, when there weren’t recent replica set changes, all the replicas are independent, and there is in fact nothing to replay, except latest committed raft entries if we want to.

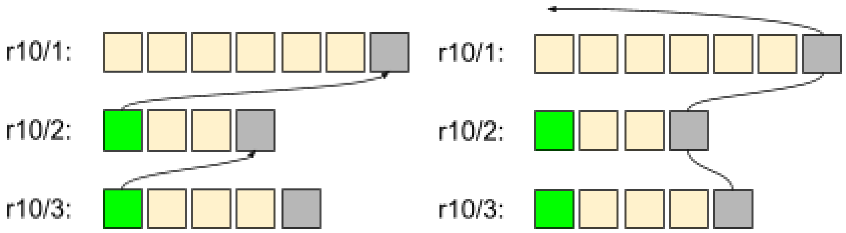
After the replay is done, the state machine engine is flushed, and replicas are started. The state machine flush allows starting the Store with the assumption that the current state is durable. Without this flush, the replay procedure would need to maintain the same data structures as the online Store. For example, the “not yet durably removed spans” structure that helps building the Init -> Destroy dependencies.

Note that the size of the replay DAG is limited by a bunch of factors. It will contain O(replicas) nodes. Furthermore, it will be limited by a multiple of the max amount of unflushed state machine data, such as the Pebble memtable size of ~64 MiB. It is affordable to load the WAG in memory during the startup.

Note that the replay only applies commands from the log to the state machine. It does not write to the log engine (such as creating a raft state for some replica), which is up-to-date. It may choose to GC the durably applied part of the WAG, but this is not strictly necessary and can be done at runtime.

Optimizations

The replay procedure applies all transitions that were committed before restart, without exceptions. There are scenarios when this work is redundant. For example, consider a replica r10/1 (1 is the LogID here) which applies a few snapshots in quick succession, and then the replica is destroyed.



The replay will repeat all the same steps, even though it is already known that its result will not be needed. So in this case we could make a shortcut and destroy r10/1 immediately. More generally, a replica removal can be expedited when it becomes clear that everything “downstream” of its current position is removed.

Implementing such optimizations would be a fun task, but the merit is questionable. The “replay all” approach is less risky and simpler to understand / debug. In practice, we do not expect redundant operations to be any significant fraction of the replay. Moreover, the size of the graph is reasonably small.

Partial Implementation

Question: can we relax the writes/ingestions total ordering requirement for the state machine engine? Today, not all ingestions are "flushable", and this guarantee does not exist.

Naively, this might require a WAL-full state machine engine, syncs after applying splits/merges, and idempotent raft log replay for the logs that have AddSSTable commands. It will look more like the ReplicasStorage design. The replayable WAG is reduced to only the tails of the raft logs (variations).

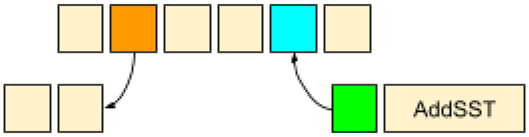
Conjecture: assume that, for every WAG dependency, the "before" and "after" write overlap by at least one key (such as RaftAppliedIndex, ReplicaID/LogID, RangeDescriptor, etc). Then the only required guarantee from the state machine engine in this case is that durability of a write implies durability of the whole chain of "happens before" writes according to such key overlaps. This implies a valid durable WAG prefix.

TODO: explain things through serializability and per-key linearizability.

When arbitrary interleavings of batches and ingestions are possible, Pebble at least provides this relaxed guarantee. And so we are mostly good today. The only wrinkle is AddSSTable which only touches the "user" keys, and doesn't necessarily "depend" on the previous command.

Can we still workaround without having a state machine WAL? Seemingly, we could try something like solution 2, but placing a checkpoint in the log engine. However, AddSSTables wouldn't automatically flush the latest split/merge, so the state can regress arbitrarily. Solution 6 (flush barrier) would be somewhat good, in combination with solution 2. We would then need to idempotently replay logs to reapply AddSST.

Think more about the following example though. Is it problematic to replay from before the merge and split, with AddSST ingestion applied? Assuming all writes are idempotent, we might be ok. The "correct" replay applies all the writes and AddSST; the "weird" replay starts with AddSST applied, then does all the same writes, then writes the AddSST. All writes are "blind", so the result is equivalent: the out-of-order AddSST is completely overwritten.



AddSSTable synchronizes with any snapshot overlapping in the user key space. So, worst case, we would be replaying from the last snapshot overlapping with the AddSSTable.

So we might be good with just making AddSSTable a special WAG command. And modifying the "target" condition of the replay to reach the committed AddSSTables, not just Init. Or replay all committed logs.

Alternatives Considered

If the DAG replay is deemed too complex (we don't think it is), an alternative is to store the replica lifecycle transactions into a sequential log. The runtime Store has already done the necessary checks / ordering, so we could replay things in the same order. It would still be interleaved into the raft logs, so the approach is not substantially different or simpler, and the enforced order is more of a distraction.

Compute dependencies on replay instead of recording them "as it happened". Each user space key ordering is coarsely defined by the RangeDescriptor generation. A simple way to account for it is to replay logs in the order of this generation. Wrinkle: RangeDescriptor.Generation can sometimes be inverted, so it's not reliable. Example: apply a snapshot, remove the replica, apply an earlier snapshot (pre-split).

Scratch / Details

The RangeID-local keys are restructured as follows:

Log Engine	**State Engine**
RangeID=1/ReplicaID RangeID=1/LogID=100/Init = {snapshot: …} RangeID=1/LogID=100/Destroy = {...} RangeID=1/LogID=101/Init = {...} RangeID=1/LogID=101/Entry/1234 -> raftpb.Entry ~~RangeID/Tombstone = {NextReplicaID}~~ RangeID/LogID/HardState RangeID/LogID/TruncatedState RangeID/LogID/Entry/index RangeID/LogID/ApplyingIndex RangeID/LogID/Init RangeID/LogID/Destroy	 RangeID/LogID RangeID/RangeTombstone RangeID/Applied // applied index etc RangeID/{replicated}
Defines the latest state. For a RangeID, the LogID is monotonic, and incremented densely. Within a LogID, the commit index is monotonic. Existence of a higher LogID implies the lower ones are logically deleted. The RangeTombstone	Catches up with the log engine. Same coordinate system: (LogID, AppliedIndex) is monotonic. Everything from (LogID, AppliedIndex) to the log engine (LogID, Commit) can be applied.

defines all logically deleted LogIDs. Physical deletion of LogID is subject to durability of this deletion in state engine first. The TruncatedState defines all deleted entries in the current LogID. The LogID lifecycle is “logically” linear: Init, apply committed entries up to some index, Destroy, eventually GC. Init might be skipped for uninitialized replicas. LogID rotates in two cases: (1) a new ReplicaID gets a new LogID; (2) after applying a snapshot it also gets a new LogID.	The LogID inits and destroys sit in between LogIDs and are applied when the LogID is rotated or tombstoned.

Example:

Log engine: ● r10: {ReplicaID: 5, LogID: 3, Applying: 120} ● r10/LogID=2: {Destroy: 90} ● r10/LogID=3: {Init: snapshot@100} State engine: ● r10: {LogID: 2, Applied: 80}	Apply/replay sequence: 1. LogID=2/(80, 90]. Moves Applied to 90. 2. LogID=2/Destroy. Moves RangeTombstone. 3. LogID=3/Init (snapshot). Moves LogID to 3, Applied to 100. 4. LogID=3/(100, 120]. Moves Applied to 120.