Eng Design Doc:
Raft Storage Efficiency [0% draft]

<README: Document Usage Guide>

- ***Engineering stakeholders** are the audience of an eng design doc.*
- *Make a copy, and do not edit this template. Remove the help texts (blue) after completing the initial draft of this document, but before inviting review.*
- *Send to eng-design-docs@cockroachlabs.com when the document is ready for review, with the document title as the subject of the email.*
- **The sections are suggestions. Please remove, add, or edit them to suit your project and team.** *Design docs balance thoroughness with brevity. Even a conscientious reviewer has finite attention. Use it well.*
- This process fully replaces the previous "RFC" process, which is now deprecated.

Approval

The author of this document is responsible for identifying stakeholders and giving them appropriate roles here. Reviewers may suggest additional reviewers as part of the process. This is expected and encouraged.

To notify reviewers of this design doc, please tag them individually in a doc comment.

As a starting point, consider inviting:

- *As **Approver**, engineers who are subject matter engineering experts on affected teams, including your own.*
- *As **Informed**, other interested engineers and product stakeholders.*
- *As **Shepherd**, a member of the Eng design review working group, specifically someone without much at stake in this particular project. Their role is to help to identify reviewers and help cut off unproductive threads of discussion. If it's helpful, ask for a volunteer in #eng-design-process.*

Different teams or projects may call for different approval patterns. This is also expected and encouraged.

Reviewer	Team	Role (DACI Matrix)	Approved?
Tobias Grieger	KV	Driver *Usually the author*	☐
Pavel Kalinnikov	KV	Approver *Approval required*	☐
Arul Ajmani	KV	Approver *Approval required*	☐
Sumeer Bhola	Storage	Approver *Approval required*	
Peter Mattis	CTO	Informed *No approval required*	

Motivation

On each KV node, the replication layer currently uses the underlying pebble storage engine for two workloads: maintaining the raft log, and maintaining the state machine. These two workloads are polar opposites of each other:

- The raft log requires **durability**, follows a predictable **append-then-delete** pattern, and is **rarely read**.
- The state machine does **not** require durability, follows **unpredictable write patterns**, and is **read frequently**.

The result of sharing a pebble instance that isn't optimized to handle these workloads results in two main inefficiencies:

- **Higher write amplification**, meaning that each byte of user data has to be written to disk with high multiplicity. This incurs a cost through higher storage device and CPU usage.
- **Increased tail latencies**, resulting from strong durability being imposed on writes that do not require it.

The increased attention directed towards performance and efficiency has brought these inefficiencies to the forefront. In 2022, TiDB moved to a separated engine for the Raft log, now the main architectural difference between CockroachDB and TiDB. We have been talking about improvements in this area since at least 2017, and have made several pushes into this direction. Today, company objectives and team resources make this a project we should tackle again and complete.

Inefficiencies can be addressed once the workloads are suitably isolated from each other, so that the storage engine(s) can optimize for each. Past investigations and prototypes have sought to achieve this by "separating the raft log", that is, using two pebble engines per store, one per workload. We also considered, but ultimately decided against, an alternative approach that continues to house both workloads within a single pebble instance. We also aren't currently considering a specialized raft storage engine (as TiDB did), as we don't consider this necessary for optimal performance.

The main goals of the Raft Storage Efficiency project are thus

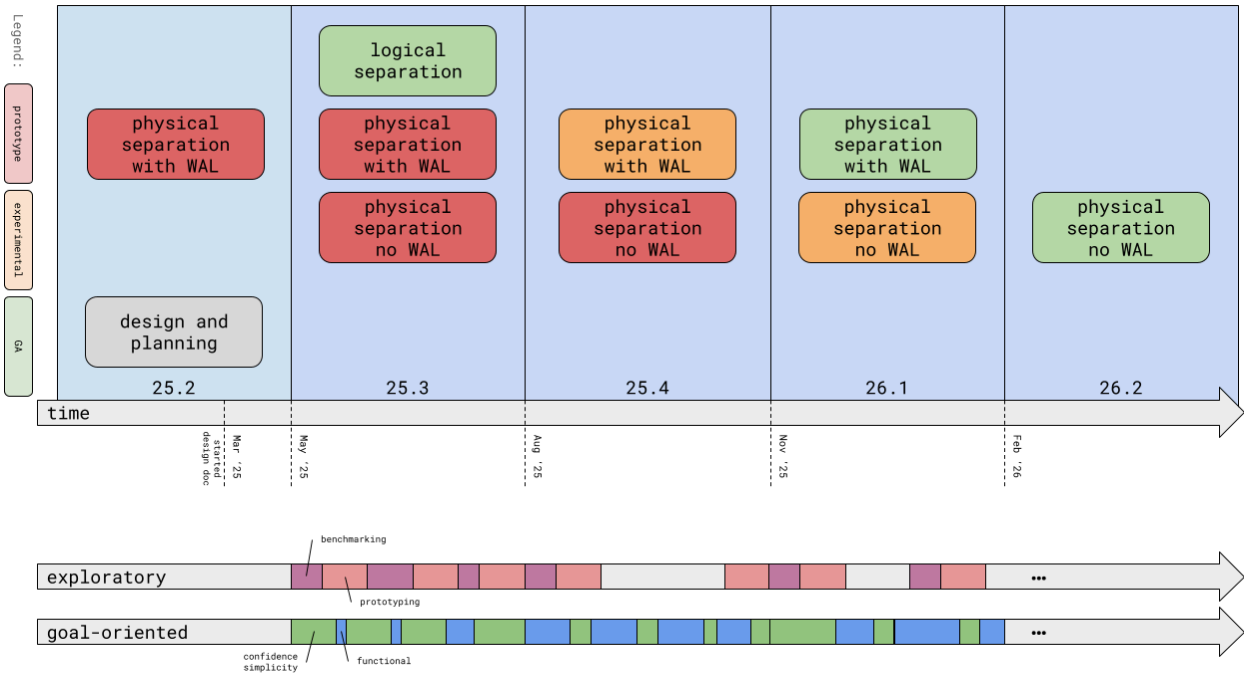
- splitting the combined workload into two, and using different pebble instances for each.
- optimizing the resulting Raft and State pebble instances optimally for each workload.

Since all of this work occurs on code paths that are as critical to correctness and performance as they are undertested - and the efficiency improvements, among other factors, depend on reducing durability requirements for state machine writes - it must be executed with the utmost caution to prevent data loss or corruption. The following diagram (discussed in more below) illustrates how we plan to balance the desire to provide user-facing improvements with the need to set them onto a solid foundation. We do so by breaking the work down into three deliverables:

First, we have **logical separation**, which is the foundation-building phase and the deliverable for the **25.3** release later this year. In this phase, we compartmentalize the two workloads in our codebase, introduce an abstraction that refers to two separate engines, but continue to back it by the single pebble instance we have today. We have determined that the bar on the affected code paths - splits, merges, snapshots, replica creation, replica destruction, log truncation, etc - needs to be raised significantly, as many of them are among the most poorly tested and poorly understood areas in the KV codebase. Correspondingly, continuing a process we have already begun now, we expect to initially spend much of our efforts on scrutinizing, extracting, and testing storage interactions in the aforementioned code paths while carrying out this logical separation - in effect leaving behind a simpler, clearer codebase.
Since we have a working prototype of the next stage (physical separation with WAL) already as well as a synthetic two-engine benchmark, in a concurrent workstream, we can cheaply continue to explore what specific optimizations engine separation would unlock, measure their impact, and forecast improvements. This benchmarking infrastructure will also be helpful to assess the impact of projects such as key-value separation, which will significantly reduce write amplification.

With logical separation in place and operating now on a solid foundation, the **25.4** cycle will be in service of “pre-productionizing” **physical separation with WAL**, in which the logical separation is underpinned by two instances of pebble, and we continue to use write-ahead logging not only on the Raft engine (where it is needed) but also on the State engine (where the WAL significantly simplifies crash recovery). This deliverable should be thought of as close to production ready, but may miss key features such as a migration, and may exhibit specific and well-understood correctness concerns which we still need to address.

While we round out and harden WAL-ful separation towards GA in the subsequent **26.1 release**, we simultaneously advance on **physical separation without WAL**, which reduces the write amplification of the state engine by an additional unit of one. This is a significant engineering effort, but one that we understand well given today’s design and can efficiently work towards.



Noteworthy Requirements

*Summarize "noteworthy" requirements here. Noteworthy requirements are those that **influence engineering design choices in meaningful ways**.*

*Below, the template includes some common categories of requirements. **Please add or remove sections as appropriate** for your project.*

Product

User Interfaces

Users will not interact with this feature directly. Performance gains are realized transparently with an upgrade to a supported CockroachDB version. User-visible flags remain unchanged; in particular, users can *not* specify different locations for each engine (though this would be technically feasible after completion of the current project).

A new set of metrics for the raft engine will be added, along with logging to the STORAGE channel.

Non-functional Requirements

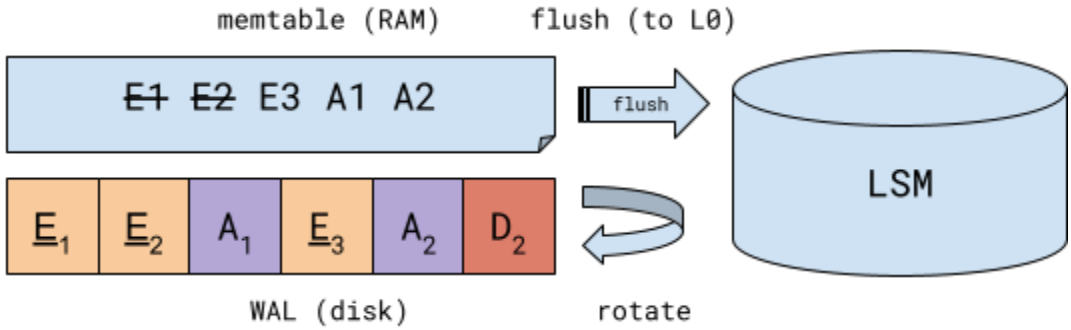
This project takes place in parts of the system that are critical to correctness, stability, and performance. We are taking a disciplined engineering approach that focuses on clear interfaces,

contracts, and testing. We balance this with a nimble approach to prototyping and benchmarking, which allows us to demonstrate and forecast the specific impact the work will have once productionized.

Dependency Relationships

We will occasionally require expertise from and/or a burst of close collaboration with the Storage team to answer questions or determine technical details. In particular, the IO components of admission control need to draw signal from both engines.

Detailed Implementation



Above we see the kind of diagram used throughout the document to illustrate the main components of a (simplified) pebble engine and relate it to the raft and state machine workloads.
This example portrays the status quo. Raft entries E_1 and E_2 were durably (note: underlined) written to the WAL in the course of a Raft log append. Subsequently, the user data contained in E_1 was written to the WAL as it was applied to the state machine (no durability required). Then, Raft log entry E_3 was durably appended to the Raft log, followed by the non-durable application of the user data in E_2. Finally, a log truncation D_2 occurred, which marked all Raft log entries at indexes ≤ 2 as deleted (note: strikethrough in the memtable).
Each WAL write is also inserted into the memtable, which is a sorted data structure with a fixed size budget from which reads are served. Once it fills up, it is compressed and split up into SSTables which get inserted into L0, the highest level of the LSM. A new memtable is instantiated, and a new WAL along with it.

TODO: more details of how everything works today and how it should work in the future in Raft: Storage Separation. Ultimately we want the many pages of nitty gritty in that doc and keep a more digestible high level overview here.

25.3 Release Cycle

For the 25.3 release cycle, we aim to achieve the ability to experimentally run with separate Raft and State Machine pebble instances (backed by the same store directory). This will be production-grade and well-tested code, though not yet suitable for production use due to lack of hardening and absence of a migration. We aim to use the 25.4 release cycle to fill in these gaps and to make further optimizations to the engine configurations.

25:3 benchmarking, prototyping, wal-less design. confidence build-up (refactoring).

RaftStorageEfficiency: Issue List

Describe what you want to build, and how you want to build it. Enumerate the technical challenges of the project, your proposed approaches, and how they will meet the requirements. Include enough depth that a technical reader of the document can appreciate the major pieces of work and their implications. This section often evolves substantially based on feedback.

Most projects need sections on:

- ***Architecture changes** such as new components or external dependencies, separation of existing components, or addition of new data flows/paths.*
- ***Deployment/Upgrades/Backwards Compatibility** discussing how new functionality will interact with existing systems, how new and old code will run side-by-side, the trade-offs considered, how new functionality is hidden until safe, etc.*
- ***Testing/Testability** on how your project design is informed by testability needs. Please outline your planned strategy and how this affects design choices like module boundaries.*

Some projects need sections on:

- ***Access Controls and Authentication**: If your work relates to permissions, roles, grants, logins, etc, please add a section highlighting the risk level of this work and steps you will take to lower risk.*
- ***Logging**: If your work logs new data, please describe how you will distinguish between "auditing" data (safe to log) and "sensitive" data (not).*
- ***Handling Sensitive Data**: If your project changes how CRL systems interact with sensitive data, describe how you will maintain the security of this data.*
- ***Persistence** of new data: If you're adding any new persistent state – for example, a new system table or new encoding – please add a section discussing the schema and indexing of this new state. Describe the expected access and retrieval patterns on this data, and any performance or scalability requirements. If this new persistent state has any interactions with Backup and Restore, please address this as well.*
- ***Introduction of major new library or service dependencies**: If you're doing this, please mention explicitly.*

Alternatives Considered

We considered an approach that avoids a physical separation into two pebble instances. A benefit of this approach was thought to be avoiding the need to "logically" separate the engines in the KV server code. There were two main reasons we ultimately favored a physical split:

- logical engine separation is a requirement for safely lowering durability on the state machine (which is a main goal of the effort), so it would be required with a single engine as well
- a number of optimizations become complex or near impossible on a single engine (see Raft and State Machine Storage -- a technical direction), and we are concerned about adding excessive complexity to pebble.

References