Fully Concurrent Reads Design Proposal

SHARED PUBLICLY

status: implemented
authors: @jpbetz, @jingyih
date: 2018-02-15

Motivation

Expensive reads in etcd cause head-of-line blocking; a read for a large range of the keyspace prevents any other reads or writes from progressing until it completes. This causes obvious latency and throughput performance problems that can limit the scalability of systems using etcd. This also impacts availability; in the presence of sufficiently expensive reads, all other requests to etcd can timeout, leader election churn can occur and health checks can fail.

Impact

- reduced P99 latency by 97.4% (data) - ~1s -> 25ms

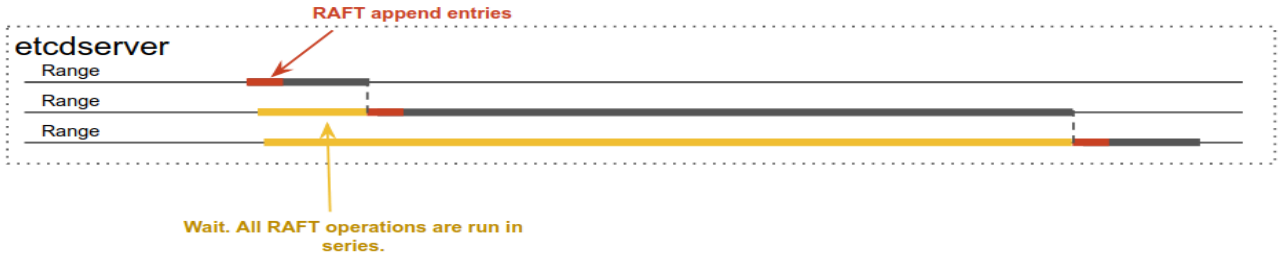
Goals

- Make etcd reads fully concurrent and ensure good throughput and latency for writes and fast reads, esp. in the presence of expensive reads.
- Define a roadmap for improving read concurrency over the next two etcd minor revisions: 3.4 and 3.5.
- Take a data driven approach to optimizing that focuses on operability and scalability gains that matter for the Kubernetes use case.

Background

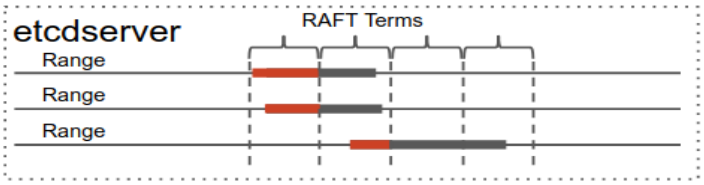
etcd 3.0: Lockstep operation execution

etcd 3.0 appends a Raft log entry for each linearizable read and processes them in series along with all other operations. In a HA cluster this means that there is at most one linzeriable read being executed across the entire etcd cluster at any point in time.

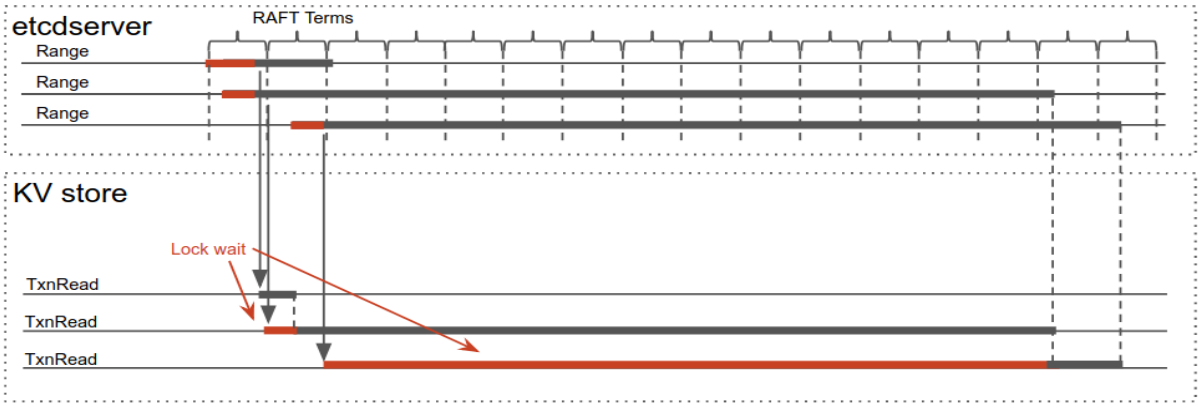


etcd 3.1: Raft " Processing read-only queries more efficiently" implemented

In etcd 3.1, an optimization to bypass the Raft log for read-only queries and still preserve linearizability was added. See section 6.4 in the Raft paper.



This allows all etcd members in a cluster to serve reads concurrent to each other. But on each individual etcd server member, the underlying etcd internals known as the "mvcc KV storage layer" is bottlenecked on a lock, which prevents any operations from being executed concurrently.



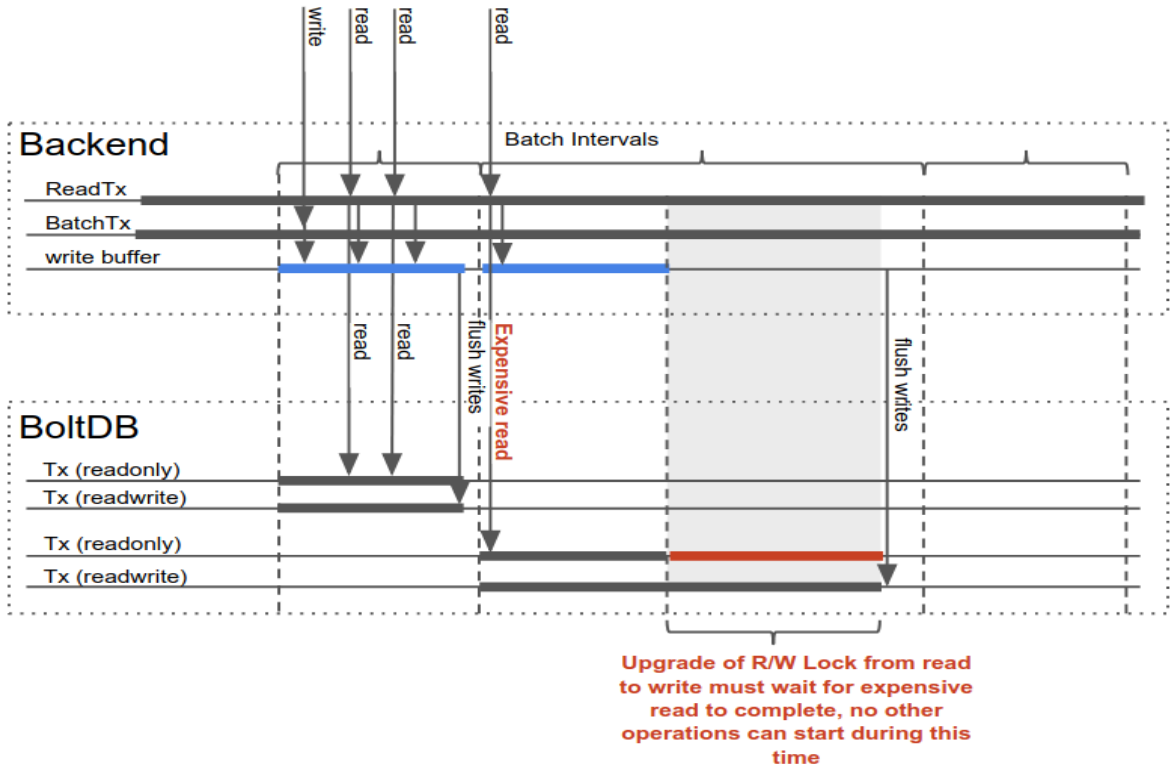
For "cheap" reads that only returns a handful of KVs per read, this lock mostly just reduces read throughput, but for "expensive" reads, all operations can get delayed to the point where requests timeout and etcd servers fail health checks.

etcd 3.2: Write Buffer + Read Concurrency

In etcd 3.2, PR #7105 makes reads concurrent using a read-write lock resulting in N reads OR 1 write concurrency. Also, writes are batched to boltdb on a “batch interval” to increase write throughput.

Every batch interval (100ms by default) a boltdb write transaction and boltdb read-only transaction are opened for the duration of the interval. During each batch interval, writes are written to both the write transaction and an in memory buffer, and all reads must “double read” against both the read-only boltdb transaction and the buffer, since the read-only transaction does not contain writes that occur during the batch interval. Each time a write completes, a read-write lock is upgraded to prevent any new reads from starting and wait for the already running reads to complete, so that a write buffer “writeback” can be executed, making the write visible to subsequent reads. The lock is also upgraded at the end of each batch interval. While upgrading the lock, no new reads can start and all writes are blocked until all reads complete, at which point the boltdb write transaction is committed and the next batch interval starts.

For workloads that only have cheap reads, this improves both throughput and latency for both reads and writes. However, since the read-write lock may be upgraded frequently, any reads expensive enough to that delay the read-write lock escalation quickly becomes the only operations running until they complete, delaying all future operations and can delay when batch interval completes, and again operations can get delayed to the point where requests timeout and etcd servers fail health checks.



Alternatives

To make reads fully concurrent, there are a couple obvious alternatives.

~~Remove the Write Buffer~~

Idea is that boltdb already supports concurrent execution of N reads + 1 write, so remove the write buffer and leverage boltdb directly. There is an issue with this: because boltdb is a copy-on-write DB, batching multiple writes into a single one provides a substantial write throughput boost which would be lost without the write buffer.

Make Expensive Reads Fully Concurrent

Continue the work started by Xiang in PR #9384. The idea in the PR is to prevent head-of-line blocking of expensive requests by delaying them until after a write buffer flush and then running them fully concurrently. A benchmark showed that this effectively eliminates the head-of-line blocking problem for simulated workload consisting mostly writes and fast reads plus a low volume of very expensive reads (100,000-100,000,000 KV range reads).

Make All Reads Fully Concurrent

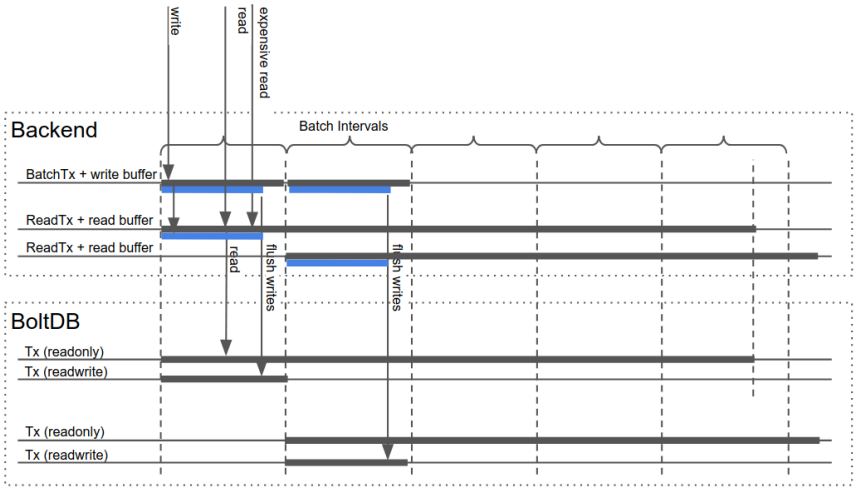
This is a variant of the “Fully Concurrent Expensive Reads” approach, but extended to all reads. The basic idea is the same. We want to allow reads to be able to run concurrent to writes and batch interval boundaries. This alternative is complicated by the write buffer, since the data required by the concurrent reads is both in boltdb and the buffer, and reads must surface a consistent view of the data from both sources in the presence of concurrent writes.

Design

We favor the “Make All Reads Fully Concurrent” alternative, which allows us to avoid having to identify expensive reads if we can adapt our read buffer to support N read + 1 write concurrency, which we believe we can do efficiently and without a major code overhaul.

Key changes are:

- Remove the R/W Lock in ReadTxn that prevents N read + 1 write concurrency
- Instead of “resetting” the ReadTxn at the start of each batch interval, create a new “active” ReadTxn at the start of each batch interval. Once the batch interval completes, the ReadTxn becomes a “passive” ReadTxn, and is open until all the reads it is serving complete. Each ReadTxn owns a read buffer that is only updated via writeback while it is “active”.
- Add read buffer support for N read + 1 write concurrency



Options for read buffer support for N read + 1 write concurrency

1. Prove that the read buffer already provides sufficient MVCC, and that reads are against established revisions (no reads that ask for “latest”)
2. Create a copy of the read buffer after each write (lazily on the 1st subsequent read, ideally) and use it for subsequent reads
3. Modify the read buffer to provide MVCC directly
4. Replace the read and write buffer with in-memory boltdb instance

“Prove that the read buffer already provides sufficient MVCC” (#1) is ALMOST feasible. Since etcd first reads from it’s in-memory index to find the exact revision of each key it needs before fetching data (from either boltdb or the read buffer), we’ve clearly met one of the requirements for #1-- that reads are “reads are against established revisions”. Since etcd writes revision -> KeyValue KVs to the “key” bucket boltdb and the read buffer uses the exact same structure, we also met the other requirement for #1-- that “read buffer already provides sufficient MVCC”. The special case we need to consider is etcd “compaction” and buckets other than the “key” bucket. Unfortunately, the ‘meta’ bucket, which keeps the compact keys, consistent index keys, restore chunk keys, would get incorrect responses if we treat that bucket as MVCC.

“Create a copy of the read buffer after each write” (#2) could work if we do one of:

- Limit the maximum number of writes in a batch interval to something small like 100. Worst case array element copies then becomes bound to 100^2/2=5000 when each write is followed by a read that needs a new copy of the read buffer.
- Don’t copy the key bucket when copying since it’s already MVCC compliant.

We could implement #3 by introducing a “mvccTxReadBuffer”. The difference from the existing txReadBuffer would be:

- An update index would be added, and each entry in the mvccTxReadBuffer would track which updated index is was written at.
- Instead of filtering out all but the newest version of a key in txReadBuffer.merge() during writeback, filter them out during read and only up to the update index of the reader, so it has a consistent MVCC view of the read buffer.

The advantage of **"Modify the read buffer to provide MVCC directly"** (#3) is that it modifies the write buffer layer to more closely emulate the semantics of the boltdb interface that it wraps. The disadvantage is that it's complex/challenging to implement, and doesn't actually provide a fundamental performance boost for the key bucket since it's already MVCC.

"Replace the read and write buffer with in-memory boltdb instance" (#4) could eliminate the need to implement something like #3 since boltdb already supports N reads + 1 write concurrency and transactions. The two main things we to watch out for are:

- The read and write buffers are deltas over the underlying persistent boltdb store
    - For the "keys" bucket, all operations are append only (deletes are just a "tombstone" append), so a union of the buffer keyspace and the underlying keyspace provides the correct result.
    - For the "meta" bucket, or any other yet-to-be-invented bucket, deletions of a key in the buffer needs to be recorded such that the key in the underlying store is masked out
- We will need to keep multiple in-memory boltdb's around since we cannot close the db until all reads it is serving complete, and we will need to create one for each batch interval that has any reads or writes.
    - Need to ensure boltdb instantiation is not too expensive (it shouldn't be?)
    - How to create in-memory boltdb instances? Do we need to use a tmpfs?

We will explore these options as we prototype an implementation, biasing toward the simplest and most obviously correct solution that meets our performance needs.

Compaction

Conveniently, since compactions do not go through the write buffer, we don't need to do anything to support them. They'll continue to work exactly as they do today; once the compaction completes, all subsequent read transactions requiring the compacted data will return ErrCompacted.

However, with this proposed change, it becomes possible for far more expensive requests to run concurrently. This can create a new problem since boltdb only frees the 4k pages it writes data to when the page becomes unreachable--that is, when no read transactions still need it. So by enabling high concurrency for large reads, we might prevent boltdb from being able to free up old pages, which, in the worst case, could result in runaway disk space growth. We're not sure how much of a problem this could be in practice, but if it is a problem, we could address it by periodically checking if expensive reads are against a compacted revision, and if so, stopping

the reads before they complete and returning ErrCompacted to allow boltdb to reclaim the disk space of the compacted data earlier and avoid potential disk space exhaustion problems.

Rate Limiting

At any point of time there will be 1 “active” and N “passive” ReadTxns. By rate limiting the number of expensive reads, the number of “passive” ReadTxns can be bounded. Rate limiting also ensures fairness or writes and non-expensive reads and so we plan to implement it along with this improvement. Since etcd’s index calculates the number of KVs in a read response before executing the read, we can use the response count to enforce a rate limit. E.g. all requests that return 1000 KVs or more will be rate limited to 100 requests at a time.

Future Work

- Large response streaming - avoid accumulating large responses in memory
- Execute expensive reads at a lower priority by splitting them across multiple boltdb transactions with waits in between to constrain their resource usage.