tl;dr Go directly to the section “How to React” below to skip the historical details and get to solution space.

Overview

A common pattern we encounter when supporting C* clusters goes something like the following:

1.) Usually in response to some operational event (streaming, importing SSTables, etc.), compaction on a particular node falls behind. Specifically, the number of pending compactions grows, all compaction threads are busy, and compaction begins to consume excessive amounts of CPU (including additional garbage collection overhead).

2.) As this happens, the node continues to receive new reads, and more importantly new writes, and these new writes fill up new memtables which are flushed to put even more pressure on an already overloaded compaction subsystem. (Repair poses a similar risk, although this was addressed in CASSANDRA-15817.)

3.) While new reads and writes might arrive directly from clients over the native protocol or from other nodes via internode messaging/other coordinators, the former are clearly making a bad situation worse (and leading to inflated server-side latencies), so we disable binary/disable the native protocol server. In some cases, this relieves pressure on compaction and allows the backlog to clear (although in some cases we’ll also revisit whether there are too many concurrent compactors), at which point we allow native protocol messages to flow again.

In short, we have no automatic mechanism to back-pressure clients in response to excessive compaction activity. Even the recently added native protocol rate-limiting introduced in CASSANDRA-16663 is configured statically, although like enabling/disabling the protocol server entirely, it can be adjusted at runtime.

Quantifying the Problem

Before we can address the problem above, we have to be able to quantify exactly what we mean by compaction “falling behind.” There are a few signals to choose from in the space of existing compaction metrics, but ideally we’d like something that is accurate (it only gets loud when compaction is putting excessive load on the hardware), responsive (it gets loud as soon as there is a problem, not after we’ve been in trouble for some time), and intelligible to operators. (If it doesn’t behave reasonably with little/no configuration, we might as well not build it.)

Pending Compactions

Looking at prior art in the codebase, one way we’ve quantified lagging compaction is simply the number of pending compactions. (Specifically, this is the sum of currently active compactions globally and the total estimated pending compactions for all tables. See

`CompactionMetrics.pendingTasks`.) As of CASSANDRA-15817, if this number rises above `reject_repair_compaction_threshold` (configured in `cassandra.yaml` w/ a default of 1024), we reject incoming requests for repair.

This signal is easy to spot with our existing tools, and the threshold is simple to think about and configure. There are still ways to configure it that wouldn't help much, of course. If we leave the threshold too high, we won't respond quickly enough to compaction falling behind. If we use an unreasonably low value, we'll respond too quickly. Along those lines, it might make sense to avoid reusing `reject_repair_compaction_threshold` for this purpose, as we may want a different trigger threshold for the native protocol than repairs.

Finally, compactions being "behind" is certainly a decent indication of the hardware and the write path being under stress, but it might also be an indication that we should push back on read traffic. The more pending compactions, the greater the risk that reads will hit too many SSTables, and that has obvious latency side effects, but the pending compactions don't need the noise of the increased disk access either. This will become more relevant as we discuss approaches to throttling.

Unleveled SSTables

While we don't directly track this metric globally yet, the number of unleveled SSTables could (in an LCS-centric world) also produce a signal indicating compaction is behind. One advantage it might have over pending compactions is that misconfigurations around the number of compactors could create a synthetic backlog that doesn't directly indicate resource stress. Rather than being an estimate, unleveled SSTables are an immediate indicator, especially of the fragmentation reads might encounter.

Flushing vs. Compaction Rewriting

Another signal, suggested by Benedict Elliott Smith, that would indicate a growing compaction backlog, is if the rate of bytes/cells being written to memtables exceeds the rate of bytes/cells being read by compaction. (In simpler terms, the idea is we're in trouble if we add more data to memtables than we invalidate through compaction.) Aside from some esoteric caveats, like the fact this might not be usable with auto-compaction disabled, it might be more responsive than signals based on unleveled SSTables or pending compactions. (i.e. We may be able to respond earlier but more gradually than we would with a threshold that means we're already in trouble.)

We currently track global bytes compacted (which is updated at the completion of a compaction). Bytes flushed is tracked at the table level, but it should be trivial to track globally. The number of bytes flushed isn't necessarily what we described above, but bytes written to a memtable and flushed to disk might not be equivalent. (Using a more coarse measure, like rows or cells, could be comparable whether it relates to memtables or SSTables.) Thinking of the lifecycle of bytes/cells through the system, the same data that is written once to a memtable might be compacted multiple times. We might have to do a bit of experimentation to confirm how

the two metrics (bytes written/flushed vs. bytes read during compaction) would behave relative to one another under stable (and not so stable) write load.

Internode Backpressure

TODO: this might not be something we attack for phase 1, but the chain of local compaction stress → local inbound backpressure → remote outbound backpressure → remote client backpressure is worth investigating

Rates of Change

For any of the above metrics, it may also make sense to characterize our “overload” signal as a rate of change. For example, if we’re looking at pending compaction tasks, we may want to track not the raw number of tasks, but the rate at which the backlog is growing. The idea is that we might be more responsive during the period where pending tasks are piling up (before we’ve hit a static configurable threshold). A rate-based signal may not be sufficient by itself though. For instance, a zero growth rate with 0 pending compactions shouldn’t produce the same signal as a zero growth rate with 4000 pending.

How to React

Whichever signal(s) from the above we decide to use, the primary action we should consider is throttling requests at the native transport, which could relieve stress on compaction by allowing fewer writes and allowing fewer reads doomed to read too many SSTables. (Some troubleshooting guides, for example, suggest responding to alerts around unleveled SSTables by disabling the native protocol entirely.)

Disabling the Native Protocol

Completely disabling the native protocol is one of our most common tools for relieving particular nodes under stress from compaction (in addition to disabling repairs, etc.). This is a reasonable manual intervention, and we can be mostly sure that we aren’t going to inadvertently cause an availability issue for one ore more rages in the cluster. However, a server-side/automatic implementation of this policy might not be appropriate, unless we can also automatically come to a reliable consensus on which replicas are allowed to disable binary. Slowing the rate of message consumption in a bounded, more gradual way is preferable.

Rate Limiting and a Lower Bound

We could respond to signals around compaction stress by adjusting the rate limiter that governs native protocol message throughput added in CASSANDRA-16663. We could translate the signal we receive into dynamically pushing the active limit lower, all the way down to a new configurable lower bound. (The bound would avoid a coordinated throttling to zero on all replicas for a range if they all fell behind on compaction.) This could come in many forms:

- As soon as a threshold is reached, bump the rate limit directly to the configured lower bound. When we cross back below the threshold, indicating that compaction is

recovering, move back to the original rate limit. This isn't very subtle, but it is straightforward. Perhaps the biggest concern would be possibly rapid bouncing between the upper and lower bounds driven by a flapping signal.
- Similar to the above, we could adjust the rate limit at a configurable gradual pace. (i.e. Move from the upper to lower bound limit at a rate of N requests/second.) This would be a bit more complex to configure, but it would ramp up the rate of requests more slowly when a node is in the final stages of its recovery from a compaction backlog episode.
- For a signal that isn't a simple threshold, perhaps a ratio of write rate to compaction invalidation rate, the ratio could map to appropriate levels between the upper and lower bound limits. (This is a bit hand-wavy right now...)

In all cases, actions taken in response to these signals could be applied by a simple scheduled process that wakes up periodically to assess the situation. (How often this happens could affect responsiveness, but avoiding explicit configuration with a reasonable default period would be preferable.)

Reads, Writes, or Both?

The original motivation for this discussion was finding a way to throttle writes in response to compaction backing up. This makes sense, given that clearly can pour more gasoline on the fire. The existing native transport rate limiter, however, does not distinguish between reads and writes. If we want to isolate writes in particular, we would have to build a sub-limiter.

Throttling Inbound Internode Messages

Hints, mutations, and read requests from other coordinators use internode messaging, and we have a few ways to limit that activity. Specifically, we have configurable limits on the global and per-IP/connection receive queue capacities for internode messages. (There is both a primary/local queue capacity and a reserve queue capacity, and both can be configured at runtime.) Drastically reducing global reserve queue capacity (or perhaps even zeroing it along with per-connection reserve queue capacity) would insulate the struggling node from operations originating elsewhere in the cluster, potentially even actuating their outgoing internode back-pressure. (In a cluster w/ RF=3 and even load balancing, this would affect 2/3 of all mutations...and of course all hints.)

Concerns

Having explored the space of what we might be able to do above, are any of those options *better* than the current SRE playbook, which entails manually disabling the native protocol server on machines with too many unleveled SSTables? The root cause of our problems around compaction being behind seem to boil down to repair/streaming (which we have guardrails to address now) and not ongoing writes. When we manually disable binary, we do it to reduce the number of timeouts we encounter from using the node as a coordinator. Perhaps throttling writes will allow compaction to recover more quickly, but if we still need to disable binary anyway to avoid timeouts, what does the former accomplish? Even throttling both reads and writes would allow a smaller number of reads to execute, and we may still observe timeouts.

If there is a way to accurately determine that a compaction backup is not being directly caused by streaming, possible solutions described above become more appropriate. Some condition along the lines of a higher than (historically) normal number of pending compactions coupled with a (historically) normal number of unleveled SSTables might fit.

Throttling inbound internode messaging reserve capacities might sidestep this problem. It would insulate the node from hint delivery, mutations and reads from other replicas, batches, and a number of other operations, but it would also leave the node available for coordination, at least until manually disabling the binary transport became necessary.