The Case for CS Knowledge in SRE

ADAM MCKAIG

Adam McKaig is a staff Site Reliability Engineer at Datadog in New York, where he looks after a metrics system. Previously he has built things at Google, the *New York Times*, Bloomberg, and UNICEF. His favorite language is C++, which probably says it all. adam.mckaig@gmail.com

During my career as an SRE, I've become convinced that knowledge of traditional computer science topics like data structures and algorithms are, while not essential to hacking together something that kind of works, an essential part of building reliable and scalable systems. This wasn't always my position on the matter; as a self-taught programmer, I got a long way without a clue about the fundamentals, believing that my own empirical approach was superior and that the world would catch up soon enough. In this article, I'll share a few of the more interesting problems that changed my attitude, how they were diagnosed, and how they were solved with better data structures and/or algorithms.

Most systems start life as an idea and are hacked together at first. The priority is to get something into production as soon as possible and iterate on it without worrying about what comes next. There's nothing wrong with that, but it doesn't work for long, and the next phase—productionization, that is, scalability, reliability, and so on—necessitates an almost totally different approach and skill set. It's also the most interesting part.

The main difference between the pre- and post-productionization phase is that the implementation details don't matter during the former, so long as it works. Linear, log-linear, even quadratic algorithms are blazing fast on modern hardware while n is small, and RAM is as good as unlimited. But however much one is willing to spend on cloud bills, once n starts getting large in any dimension, consistent high performance can only be achieved by carefully choosing and implementing the appropriate data structures and algorithms to avoid having to compromise on features. Ideally, one would be able to predict the growth of every dimension of n and design accordingly in advance, but in practice it's usually done reactively, when some subsystem is approaching its performance limits.

It's highly instructive to implement every detail oneself, but rarely is it necessary in practice; even the most esoteric data structures and algorithms are readily available as packages for most languages. Much more important is to develop an intuition for their performance characteristics and to be able to spot those same characteristics in production workloads.

Practical Examples

These are real examples of things going wrong at scale. I've redacted sensitive details and condensed them for brevity, but these are issues encountered in production at large companies you've probably heard of.

Fixing an Assumption

My team was supporting an old C++ service, part of a messaging system, which was having trouble sustaining its required write throughput. The service was consuming create/update/delete events from a message bus, and providing an API to view the most recent messages sent or received by a given user. It had worked fine for a long time, but it couldn't keep up as the rate of events increased, and users were complaining that the API was serving stale data

during peak hours. This service was running on a single big machine, so the most obvious solution was to shard the service and run it on many machines. But that would take time, and we wanted to improve the situation sooner.

We improved things a bit by providing a lot more CPU and looked into the implementation. What we found was unremarkable: a big `std::map` (an ordered tree) holding the latest messages, keyed by the user ID and timestamp. Writes would either insert a record, or fetch a record, patch it, and replace it. Reads would find all messages with a matching user ID and return them, which was efficient because they were adjacent and already sorted. Old records were garbage collected in a background thread by periodically walking the entire tree.

Ordered trees are a great default for mixed workloads, that is, workloads which have a similar proportion of reads and writes. But when we looked at the data from production, we saw that the rate of reads was actually remarkably low compared to the writes, which accounted for the vast majority of work. These writes weren't slow, but they weren't fast enough to keep up with the desired volume. We also saw that our read latency was consistently far below the threshold at which we would be paged about it. So we investigated how we might speed up writes, knowing that we were able and willing to *sacrifice some read performance* to do so.

It was simple for us to swap out the map with an LSM (log-structured merge) tree, a data structure which resembles an ordered tree but offers far more scalable inserts at the cost of slower and less predictable reads, using an existing open source package. We dark-launched this change into production and observed, as we'd hoped, a tremendous improvement in throughput with only a modest regression in read latency. I don't recall anyone ever complaining about the latter.

This incident taught me that although most systems rightly expect mixed workloads and so optimize for that, that isn't always the reality in production, and making concessions on one side can yield big improvements on the other.

Consider Non-Requirements

Here's a totally different example. Much later, at a different company, I was supporting a distributed key-value datastore (of sorts) written in Go. The overall workload was fairly mixed: lots of writes and lots of reads. The system stored highly denormalized event data and was primarily used to answer arbitrary questions like, "What are the most-viewed widgets by users who looked at this widget this week?" in real-ish time.

One subsystem was causing trouble: the directory service, which basically kept track of which data were on which storage node, and how CPU-loaded each was, so that the query nodes could fan out incoming reads to the right places. This subsystem was read-heavy, and the load varied throughout the day as end users came and went. The rate of writes was more consistent, since it was simply proportional to the number of storage nodes, which periodically announced the ranges of keys they had and their overall CPU load. Both would change regularly as data was rebalanced by a separate subsystem.

The problem we were seeing here was that many directory reads were too slow during peak hours. Up to about the 90th percentile was fine, but above that, performance varied wildly. We were able to improve things by horizontally scaling (roughly doubling) the number of directory nodes, thereby reducing the rate of reads that each had to handle, but this caused two more problems: utilization of these nodes was now low enough that well-intentioned cost-saving alerts were going off, which needed silencing; and this increased load on the storage nodes, because they needed to send twice as many announcements! Clearly this was a temporary mitigation, so we looked into improving the read throughput.

The implementation was (roughly) an augmented interval tree, storing ranges of keys mapped back to the storage node they could be found on, and a map of nodes to their last-reported CPU load. Writes would update both of these: key ranges would be inserted into the tree, and the load would be updated. Reads would read from both: the tree would be queried for nodes containing matching keys or key ranges (of which there could be many), and the load of each node looked up from the map.

The bottleneck here was of course the tree, because there wasn't much else to the system. Profiling indicated that reads were too often being blocked by writes, which had to lock the tree while they were mutating it.

Given the requirements, and without fundamentally changing how the system worked, we couldn't think of an obviously better implementation. We started designing a sharded directory service, making it a nested distributed system of sorts, but so many tricky edge-cases came up that we shelved it until it was really necessary—which in the end it never was. The solution presented itself when we went back and *reconsidered the requirements*.

We needed to maintain an up-to-date map of keys to nodes, which was small enough to fit on one node, fast to query, and fast enough to write that it didn't interfere with the reads. But it didn't need to be completely up-to-date: this was an OLAP system, not OLTP, and the map was always a bit stale because storage nodes only reported periodically. Could we put a cache in front of the tree, to speed up some reads in exchange for making the data slightly more stale? We couldn't think of a cache key which would actually be effective, since the keyspace was so large, but someone suggested: how about we cache the whole tree? We have plenty of spare RAM.

The resulting implementation was simple and effective: rather than one tree, we stored three. One was used to serve reads; one was updated as writes arrived. To update the read tree, the write tree was locked and copied to a third location, and only then were incoming reads briefly blocked as a pointer was swapped to point to the new read tree. This frequency was tunable, and in practice even doing so once a second was enough to virtually eliminate the variance in read throughput.

I think about this incident often when considering requirements and am reminded to carefully consider non-requirements, too. Here, freshness and low memory usage were non-requirements. The older implementation was simple but much slower than necessary because it fulfilled requirements which were unnecessary.

Undoing Lock Contention

Here's another example. More recently, I was supporting a disk-based time-series database, written in C++. This system had a mixed read/write workload, which is typical for time-series systems. The writes were small, usually containing a single point for a lot of metrics, and there were a lot of them. The reads were far fewer, but far larger, often fetching data for a single metric across a wide range of time.

My team was being paged because the error budget of our query availability was being depleted—slowly, but fast enough that we would run out by the end of the month if we did nothing. We could correlate the start of the problem with an organic increase in traffic, so we assumed that the problem would remain until we solved it—or until our customers got fed up and the traffic went away. We mitigated the problem by throwing extra capacity at it, but decided to investigate further.

We determined that a small fraction of the synthetic queries issued by our probers were taking so long to complete that they were timing out. They seemed to occur randomly (in both time and space) but, curiously, appeared to be correlated with small spikes in the fraction of *all* queries timing out. The problem was rare enough that we didn't have any relevant traces available, so we increased the fraction of traces until we caught a few of them. The same pattern presented in all of them: the query appeared to be fanning out to a few storage nodes, as expected, and returning quickly from all but one of them, which timed out.

We examined various metrics emitted by the node where the timeout occurred, around the time it did. RPC server latency was typical at the 90th percentile, but it spiked around the 98th for less than a minute, then went back to normal. CPU load was normal. Memory usage was up by a small amount. IOPS was as expected. None of these things seemed to be the cause, so we looked into the implementation. What causes random latency spikes when not under any kind of load?

The nodes in question had two jobs: store incoming data and make it available for querying. The implementation was roughly as follows: each unique time series was stored as a buffer of (timestamp, value) pairs. To quickly look up these series, a central metadata object served as an index, holding nested maps of field names and values, which in turn held pointers to the buffers. This was a big object, and it was protected by one big lock.

Writes and queries were able to scan for matching series while holding a reader lock, meaning that many such scans could occur at once, and the object would not change under them. Upon finding the pointers to the relevant series, points were appended or fetched from the vectors, which were protected by another read/write lock. But there was a special behavior for writes containing new series. Those were not present in the metadata object, and the buffers didn't exist. So before inserting the points, the implementation took an exclusive (writer) lock, allocated the new buffers for each new series, and inserted the relevant elements to the metadata object.

Experts speculated that the cause of those read latency spikes was likely to be *lock contention* on this metadata object. This was confirmed with instrumentation and profiling.

Unlike in my previous example, these nodes were resource-constrained, and these metadata objects already accounted for a significant fraction of the total RAM usage. We couldn't trade space for speed. We needed to make the writers hold the locks for less time.

We accomplished this by replacing the global metadata lock with narrow locks on the individual nested objects within it. When a write included previously unseen series, it would lock only the relevant map while inserting. This went all three levels deep (metric names, field names, and field values), resulting in many small locks instead of one large one. Writes might need to acquire multiple nested locks, but each was brief, and blocked only a fraction of reads rather than all of them. The new implementation was far more complex and idiosyncratic than the original, and it was right that it was put off. But when the time came, it was very satisfying to see it replaced with something so much more performant.

This project taught me that as throughput increases, so too does the importance of careful locking. Even very brief pauses can have a large impact if they're blocking many requests.

Conclusion

These experiences, and others, have changed my approach to growing and maintaining software. I'm writing about them because I wish that I'd become convinced sooner that this fundamental knowledge was important and worth studying, and perhaps concrete examples would have helped.

Finally, some unsolicited advice: Next time you're faced with a persistent performance or reliability problem, by all means do what is necessary to mitigate the problem first, but consider, then, identifying the underlying bottleneck. Are the performance characteristics of your data structures misaligned with the shape of your actual workload? Has some value of n become too large to ignore? These problems can be solved, and we must not be afraid to do so.