Accelerating Kafka in AWS

Michael Nordberg Follow

Oct 8, 2018 · 9 min read

Using Kafka for building real-time data pipelines and now experiencing growing pains at scale? Then no doubt—like Branch—you are also running into issues directly impacting your business' continued ability to provide first class service.

While your own Kafka implementation may vary, we hope you can leverage our learnings and ultimate success in optimizing Kafka to

effectively scale your data processing and support your growing business.

In this article, we'll walk you through:

- Branch's historical usage of Kafka
- Resulting hardware related problems
- Our chosen solution
- Performance cost and effectiveness
- Avoiding a great danger in the migration process
- Theorizing what could have been done better
- Exploring possible future improvements

Usage

In its own words, Apache "Kafka is used for building real-time data pipelines and streaming apps. It is horizontally scalable, fault-tolerant, wicked fast, and runs in production in thousands of companies." (ref: https://kafka.apache.org)

Branch utilizes Kafka for rapid event processing, feeding big data systems, the occasional unfortunate backfill, and longer term storage of logs to use in advanced troubleshooting. In total, we manage three clusters with 2–400K messages in per second across approximately 115 topics consuming 135TB+ on disk with continued growth. Certainly

not one of the industry giants, but these are mission critical to our operations and customers.

Historically Branch had been running Kafka within the AWS Elastic Compute Cloud (EC2) on **d2.xlarge** instances (30.5GiB RAM, 4 vCPUs, and 3x 2000GiB HDD). In the age of solid state drives, spinning rust may seem quaint, but it's still a reasonable choice for long sequential reads and writes, which is generally what you'd expect within Kafka.

Problems

Everything's fine, until it's not…

When in a healthy state, the performance and responsiveness of Branch's d2.xlarge based Kafka clusters were sufficient and economical. They supported our needs for several years. However, with growth, scale, and added product and pipeline complexities we began to encounter issues directly impacting our ability to provide our customers first class service.

Some of the issues encountered included:

- I/O timeouts to ephemeral local disks resulting in odd Kafka states and hung/panic'ed systems.
- Random nodes would become confused about leader states.
- Poor balancing of data across three mount points required over-provisioning.

- Inability to “read from beginning” on large topics without incurring I/O issues.
- Unable to safely/quickly repartition/rebalance topics across brokers (without use of throttling), making proper growth difficult and time consuming.

When nodes become IO/Bound, they can fail and affect other nodes in a domino effect, as the cluster attempts to heal itself.

There is also a loss in productivity, the high labor cost, and the negative impact on morale. Our engineers would rather spend their time on engaging and innovative work for our customers, instead of processing backfill jobs and performing clean-up work.

Solution

We knew we wanted faster disks and a modem kernel to improve I/O stability and survive the occasional broker failure with quick recovery time. We also needed *fast* repartitioning to allow for rapid resizing of clusters to better control operational costs while minimizing labor costs.

AWS continues to provide new classes of Elastic Compute Cloud (EC2) instances, generally with increased performance and decreased cost. For high-performance systems, like Aerospike, we use i3.4xlarge instances. For comparison:

EC2 instance	d2.xlarge	i3.4xlarge
RAM	30.5 GiB	122 GiB
CPU	4 vCPU E5-2676 v3	16 vCPU E5-2686 v4
DISK	3 x 2000 GiB HDD	2 x 1900 GiB NVMe
NETWORK	moderate	High (up to 10 Gigabit)

These are powerhouse machines with very fast NVMe drives, however.. They only provide 3800 GiB versus 6000 GiB of combined storage.

Let's examine the annual operational cost per GiB of disk space (relative to us-west-1 pricing using https://www.ec2instances.info for pricing reference). Pricing for both On Demand and Reserved instances will be shown.

1000 GiB annual cost	d2.xlarge	i3.4xlarge
On Demand	$1,140.26	$3,172.04
Reserve	$716.86	$2,169.25

That's a substantial increase in cost. Additionally, we found that the balancing of data between mount points in not ideal, with variances between 15..30% observed during testing. Despite there being good reasons for this (beyond the scope of this article), this can represent an expensive waste if you're trying to maintain an 80% disk utilization.

However, it was also observed that most of the logs were plaintext and somewhat repetitive… *Ah-ha!*

To switch to sending and consuming compressed data would have been a major undertaking. Instead, we decided to look at compression at the storage layer. We arrived at using ZFS.

"ZFS is a combined file system and logical volume manager designed by Sun Microsystems. ZFS is scalable, and includes extensive protection against data corruption, support for high storage capacities, efficient data compression, integration of the concepts of filesystem and volume management, snapshots and copy-on-write clones, continuous integrity checking and automatic repair…" It also includes software RAID support.
(Ref: https://en.wikipedia.org/wiki/ZFS)

Using ZFS, we combined the two NVMe drives using RAID-0 and then tested enabling lzjb versus lz4 compression. Lz4 was the clear winner, achieving ~3.33x compression compared to lzjb's ~1.54x.

"***RAID-0! That's dangerous!***" We rely upon Kafka's replication to protect against data loss. In our operation of i3 instances within AWS we have rarely seen a single NVMe drive fail (if at all), as compared to the entire i3 instance itself failing. Also, we'd rather lose an entire set of data than maintain a corrupted set of data. RAIDing the disks together solved the data imbalance issue, allowing for greater utilization of a given instance.

"***ZFS is a CPU hog! Lz4 uses too much CPU (compared to lzjb)!***" In moving from 4 vCPU Xeon Haswell to 16 vCPU Xeon Broadwell, we had ample CPU. Under stress testing, the CPU never became the concern.

The difference between the two compression algorithms was negligible, amounting to roughly 3%.

"***That.. That sounds nice, actually.***" It is, and there's more.

The NVMe drives are fast. Measuring their un-tuned performance with bonnie++ shows:

Bonnie++ results	D2.xlarge HDD	I3.4xlarge NVMe
Block K/sec OUT latency	183,973 556 ms	756,920 830 ms
Block K/sec IN latency	212,767 118 ms	1,531,058 4307 us

ZFS also provides a powerful layered caching system. Of main interest to Branch and Kafka is the first level cache, **ARC** (a variant of adaptive replacement cache) and its utilization of **transaction groups** to make writing more efficient. With 122 GiB of RAM, the i3s have four times as much RAM as the d2s. This allows for a considerable amount of data to be consumed directly from RAM, instead of needing to make an expensive trip to disk. (ref: https://en.wikipedia.org/wiki/ZFS#Caching_mechanisms:_ARC_(L1),_L2ARC,_Transaction_groups,_SLOG_(ZIL))

Combined with the large network bandwidth allowed to the i3s, systems management of the cluster is no longer approached with fear. Removing or adding broker nodes to the cluster, a process that formerly took days to perform, can now be performed in just a few hours.

Instead of carefully moving one topic at a time with rate limiting, we've been able to select every topic for rebalancing at once and allow it to proceed without throttling. (But there is a catch. More on that later.)

*"**Hold on. What about cost?!**"* Cost, unfortunately, is a mixed story.

Returning to our "cost per 1000 GiB" model, and focusing solely on our preference for using Reserved instances, the storage cost for i3.4xlarge instances is 3.03 times more expensive than the d2.xlarge instances. Meaning that to break even on cost, the compression rate achieved by ZFS with lz4 must be 3.03x.

Across our clusters we have a compression range of 2.33 to 3.55x, with a weighted average of 2.80x, which means we're paying ~8% more than before. (Which isn't entirely true… as we were previously over-provisioned due to concerns about trying to reduce cluster size.) But consider…

For an ~8% increase in cost, we gained:

- Stability
- Speed
- Greatly reduced labor costs
- Agility and safety to quickly grow/shrink clusters
- A tremendous file cache
- Incredible ZFS instrumentation

- Happy co-workers and customers
- And the ability to sleep at night.

We were also able to use the occasion to switch to a new kernel, upgrade to the newer version of the Prometheus Node Exporter, and make several other welcome improvements.

Migration

The migration itself was difficult. In general, we succeeded by reducing the retention period on topics where that could be temporarily allowed, such as for longer term debug logs, to reduce the size of the migration. Topic were moved from d2 brokers to i3 brokers, one at a time. For the most part, this was achieved with few incidents. Our services are tightly instrumented and alert early, so some systems issued overly-paranoid alerts, such as when consumer and producer clients had to learn the new locations of their topics and partitions.

We also encountered one particularly dangerous issue! Disk flooding due to how expiration policies are performed.

To understand the problem, first understand how Kafka stores the data you send it. Let's assume there's a topic name "beispiel" that exists with 64 partitions, with a replication factor of three, with a retention period of one day, which consumes 1000 GiB of disk space. Let's examine partition number 7, which has one primary (on broker-A) and two replicas (on broker-B and broker-C).

On broker-A, there is a directory named “/mnt/kafka-logs/beispiel-7”. Inside that directory you will find a series of files, including “.log” files. All but the most recent “log” file will measure 1GB in size (by default configurations).

Periodically, Kafka will look inside “/mnt/kafka-logs/beispiel-7” to see if there are any 1GB files ready to be expired, based on the **timestamp(s) of the file**…

Now—let’s say we move primary partition 7 from broker-A to a new broker, broker-M. Essentially the directory and all of its contents will be moved from A to M. All of the files on M will now have **new timestamps**.

Right… Your expiration policy is now **useless** for the next ~24hrs! Your 1000 GiB of disk space will double to ~2000 GiB and then start to sharply drop back down to 1000 GiB. Unless you’re operating with an initial disk consumption of less than 50%, you’re going to flood your storage.

To avoid this (and the particulars of this implementation may be version specific so please carefully review the documentation and perform safe experiments before following this) you can apply a temporary **retention.bytes** configuration against the topic. To determine the right “bytes” size, the equation is:

((([Total bytes on disk for the topic] / [number of partitions]) / [replication factor])) x [overage allowance percentage])

So, for topic “beispiel”, to allow the disk consumption to only grow by 10% above its current usage, the equation would be:

(((1000GiB / 64) / 3) * 1.10) ~= 6151645870 bytes, applied with:

> *kafka-topics.sh—zookeeper zk-1.example.io:2181,zk-2.example.io:2181,zk-3.example.io:2181/clustername—alter—topic beispiel—config retention.bytes=6151645870*

And then later:

> *kafka-topics.sh—zookeeper zk-1.example.io:2181,zk-2.example.io:2181,zk-3.example.io:2181/clustername—alter—topic beispiel—delete-config retention.bytes*

…to remove the config. (At the end of the migration work, you’ll want to consider performing a “describe” on all topics and grep’ing for any mentions of “retention.bytes”.)

Unfortunately… You’ll need to do this with all of the topics that are large enough to cause trouble.

Lessons Learned and Future Improvements

Compression: We found that lz4 compressed better than lzjb with little additional CPU overhead, but didn’t attempt any other compression algorithms. It would be interesting to compare lz4 to gzip-6 and gzip-9 to see how they perform for compression and CPU overhead.

Encryption: ZFS, among many other things, provides encryption. Given the speed of the new cluster, this will be an easy task to complete.

Topic Manifest: Keeping a better mapping of services to clusters and topics would have been useful in knowing which specific teams to alert before each topic was migrated, and for muting or adjusting alert levels temporarily for those services.

. . .

Interested in the engineering challenges discussed here? Come help us out! Check out https://branch.io/careers/ and tell them Nordberg sent you.