LOAD TESTING APACHE KAFKA ON AWS

[update: This was written before EC2 d2 instances were released, which I'm currently a fan of. I would generally recommend them over r3s.]

Notice my careful usage of the phrase "load testing" vs "benchmarking". Why is that? I think we've all learned by now that benchmark tests are often limited by many factors: sound testing theory, properly working tools and accounting for what's actually happening at a somewhat sub-atomic level (at least, relative to what you're testing). For instance, if we're comparing two hard I/O bound services on a striped RAID level where 'Service A' instigates significant levels of I/O inflation, and only because it's designed I/O profile disagrees with the configured stripe/chunk sizes, results in severely degraded performance. What are we really testing in this case if a storage knob can be easily spun to favor 'Service A'? The thing is that there's almost certainly a number of factors that aren't accounted for or even an awareness of (great example on a simple but significant mishaps in judging language performance).

I consider load testing a less canonical and more casual exercise of making measured generalizations regarding some interesting setup that I have in front of me. In my case I'm running Apache Kafka on EC2 and have somewhat large message sizes (2.5KB range vs the typical 180 byte server logs). I have a fairly controlled upstream message pipeline that imposes throughput limits (message rates before hitting Kafka), and I only have a need for ~4 hours retention in a primary topic(s).

But what performance characteristics should I expect in my particular configuration? What should I consider a severe latency spike? What's the impact of dropping a broker? The reason I'm interested in load testing is to form a collection of expectations so these questions are better understood in the reality of production infrastructure (this is taking the approach that I'm completely new to Kafka [which is true]).

To help get me there, I put together a little load testing tool (that's still a bit of a WIP). It's appropriately named after a gritty pirate weapon: Sangrenel. It works by generating random messages of configured byte sizes to bombard Kafka clusters with, and periodically spits out throughput and top 10% worst latency averages (it's a synchronous publisher and meters the latency from message send time to receiving an ack from a broker). Additionally,

it allows for fixed message rates so that Kafka brokers can be observed at specific message size/rate scenarios.

In spirit of the opening paragraph, consider that using this tool may not accurately characterize your own workloads (see repo for notes), nor does it define a standard by which Kafka should be measured. What it asserts is that the particular arrangements of *n* connections, producer threads and *x* message size that we are seeing *y* latency or *z* throughput. It allows us to compare varying arrangements and understand relative performance differences. What we're doing here is somewhat simple.

Some Initial Thoughts

After building up and tearing down too many Kafka clusters for a week, I found some variables would be interesting to share: the measured impact of varying partition counts, message rates or sizes, and so forth.

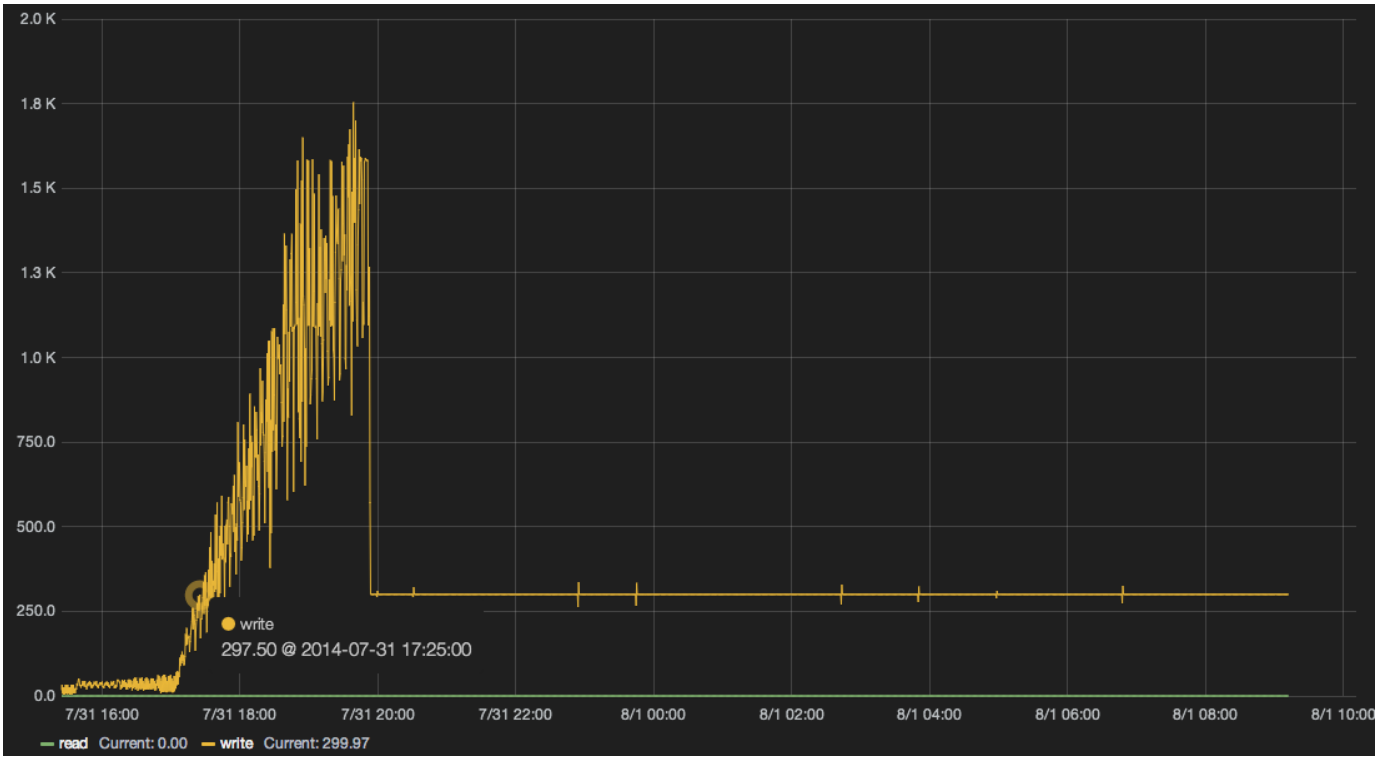
The testing setup, some background on my resource specs, Kafka performance

I've standardized on using EC2 R3 series (high memory) instances and general purpose SSD EBS volumes. Kafka exhibits relatively low CPU usage but greatly benefits from large amounts of memory available for the Linux page cache. Plus the Kafka authors were awesome enough to ship consumer reads using the `sendfile(2)` syscall; cached reads can pretty much saturate a Gb+ NIC while the CPUs practically sleep and the Kafka JVM heap remains remarkably stable. In testing I've found that the R3s large memory allotment relative to the somewhat low CPU allotment to be a non-issue. I've been unable to overly tax the CPUs before saturating the network in every scenario.

The second observation is that the advertised characteristics of Kafka are highly reflected in real world tests. Kafka has a lot of design effort behind it to make I/O as highly sequential as possible and allows the kernel handle the majority of the of the write scheduling. Because of this, the most prominent performance determinant boils down to pages being flushed to disk. The behavior is very predictable and clearly visible while watching the brokers (you'll see the disk sync at the same time as a latency spike at the producer). In fact you'll notice in the graphs below that spikes in latency and dips in throughput are not only in tandem, but with fairly consistence cadence (coupled with the broker flush intervals). The ability for the Kafka brokers to get sequential writes to disk should be considered the most significant performance factor.

Therefore, all of the cluster specs I've come up with use the GP SSD type EBS for the excellent throughput in burst intervals and a degree of guaranteed performance minimums over standard EBS. Larger setups are using the volumes in LVM2 stripes (and are EBS optimized instances).

Note: Keep in mind that the advertised AWS spec is 3 (≤ 16K I/O size) IOPS per provisioned GB with spikes to 3,000 IOPS per volume, regardless of size. Amazon's throttles are generally dead on; if you've really under-provisioned your storage where write flushes are almost constant, then don't test for a long enough period, you may not get throttled to the baseline 3 IOPS/GB and experience skewed results. Example of a 100GB volume under write saturation until being throttled to baseline of 300 IOPS:

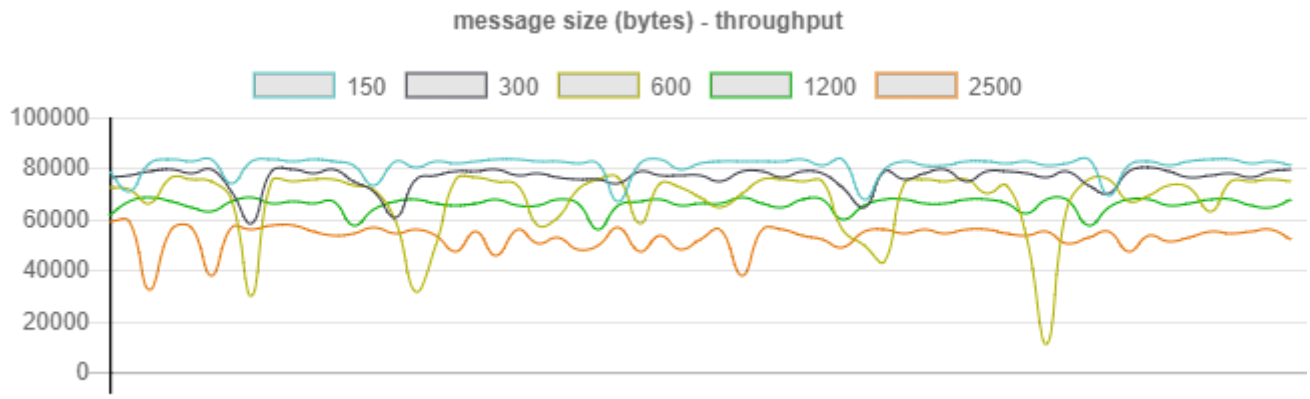


Lastly, Sangrenel is being run from one or more of my favorite EC2 instances: the c3.8xlarge / $1,200 a month of 32 core firepower (the repo notes why it's important to ensure you have sufficient juice to generate messages). **All tests in this writeup are run for a 5 minute duration**.

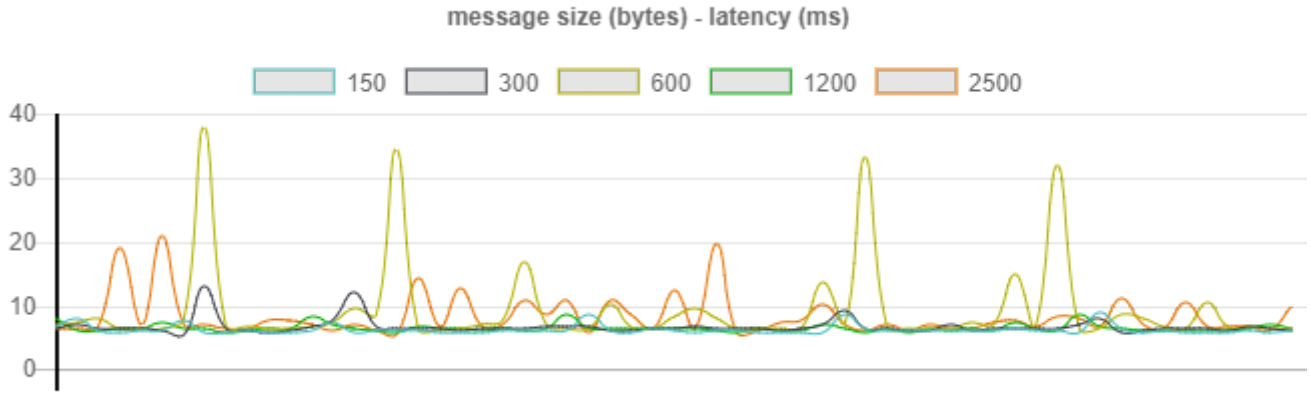
Anyway, off to the fun–

Message Size

One of the first things I was interested in is the impact of our large events in comparison to "normal event sizes". Here we're using a 3 node Kafka cluster made from R3.xlarge instances, each with 3x 250GB GP SSD volumes in an LVM2 stripe (stripe width 3 / 256K size). The load testing device is a single Sangrenel instance @ 32 workers and no message rate limit, firing at a topic with 3 partitions and a replication factor of 2:



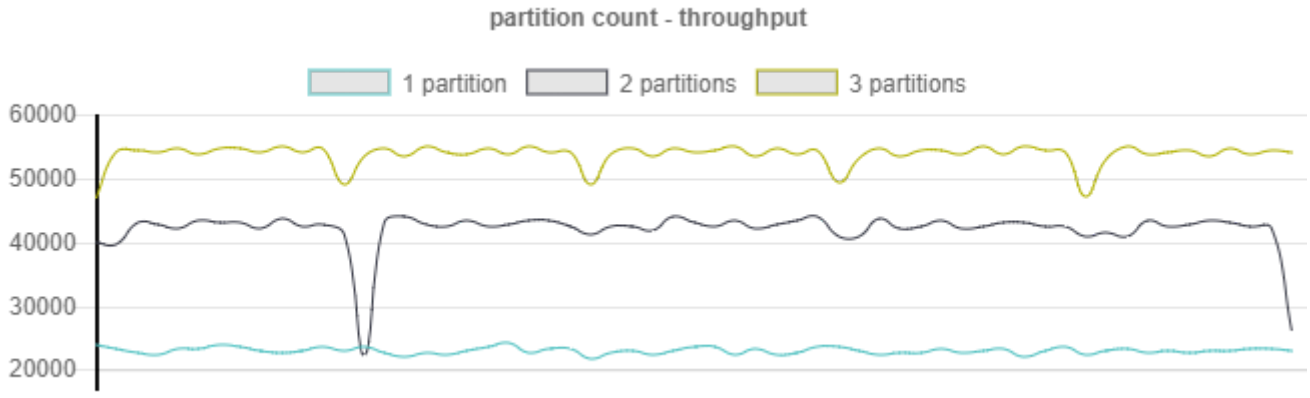
You can see that the 300 byte message sizes hover around 80K while 2500 byte (our reference size) sticks closer to the mid 50K range. Interestingly, the 600 byte size message stream had unusual spikes. I actually tested this as a repeatable phenomenon and left in the results. While I have theories, I haven't put in the time to figure that one out yet.



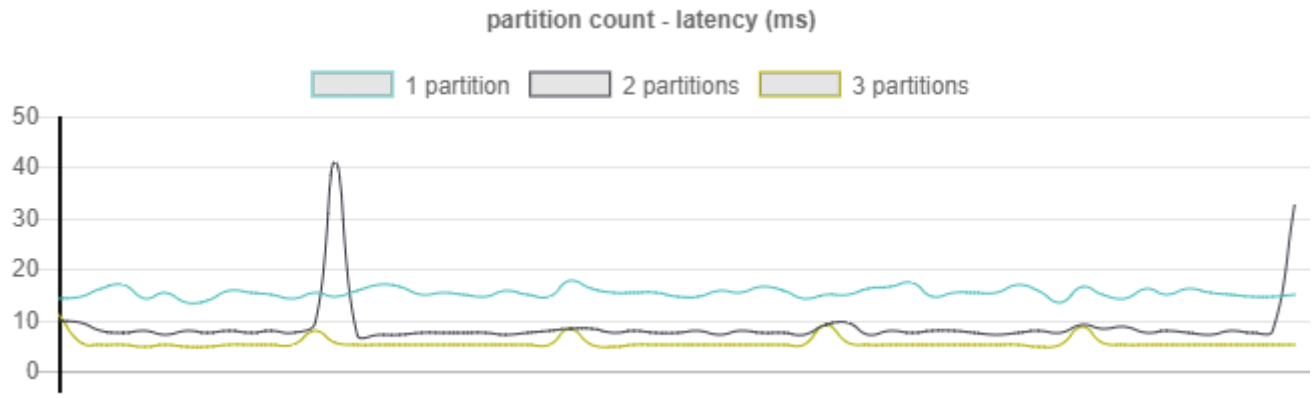
Keeping in mind that these are top 10% worst latency averages and that we have that strange 600 byte outlier, it seems this setup likes < 1200 byte message for single digit ms ack latency.

Number of Partitions

So what do we get with more or less partitions, anyway? Using the same cluster as the previous test but firing at 1, 2 and 3 partition / 1 replica topics with a fixed 3500 byte message size:



Going from 1 to 2 partitions is basically a linear increase. Going from 2 to 3 isnt, but still expresses a considerable gain.

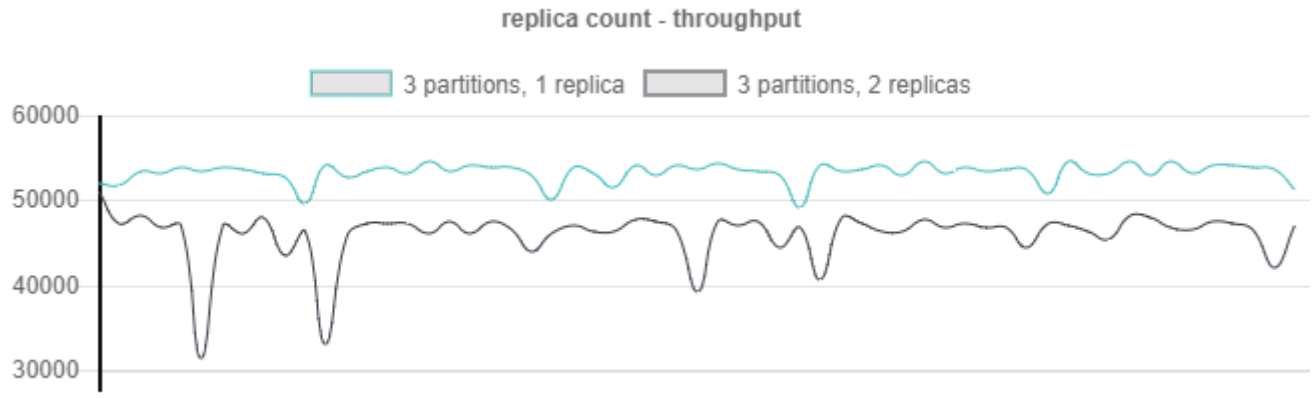


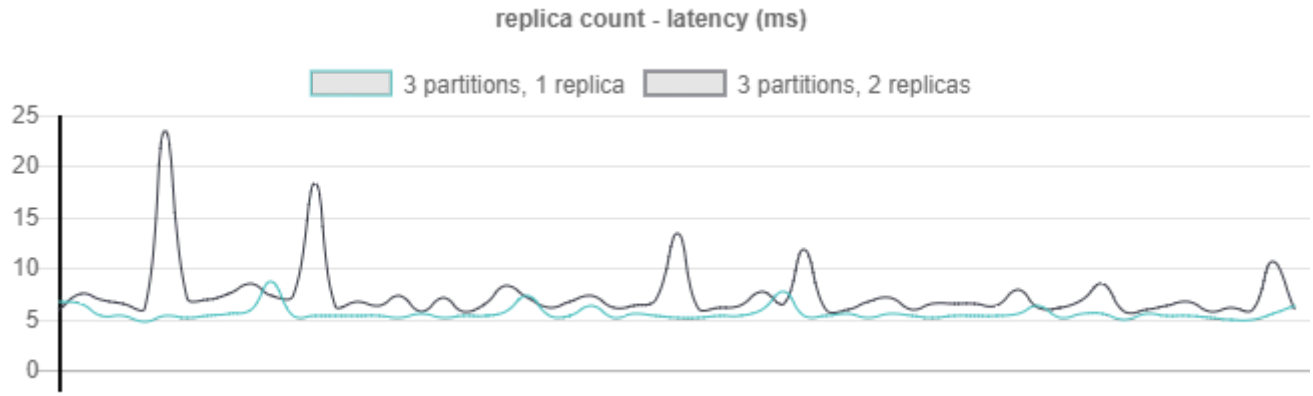
Another gain is lower and more consistent latency with increased partition counts. That first spike on the 2 partition setup happened to be unusually large and wasn't repeatably bad in subsequent tests.

Oh, and the 3500 byte size (versus my 2500 number) is just for fun. I went back and did some of these tests for this blog posts and not my production work.

Replication Cost

So what about those replicas? Same cluster, 3500 byte event size and 3 partition topics in both 1 and 2 replication factor (so 1 total copy of each partition compared to 1 primary and 1 replica copy):





As you'd expect, performance is worse with replication because you're essentially doubling the write volume per node. Additionally, you increase the the cluster traffic since each partition's data is being streamed to the other ISR nodes. **Really measure the network throughput** of whatever topology you choose and think about how much your producers/consumers are going to eat up as well. I dropped nodes from the ISR groups many times by starving intra-cluster bandwidth :(

Consumer Position Impact

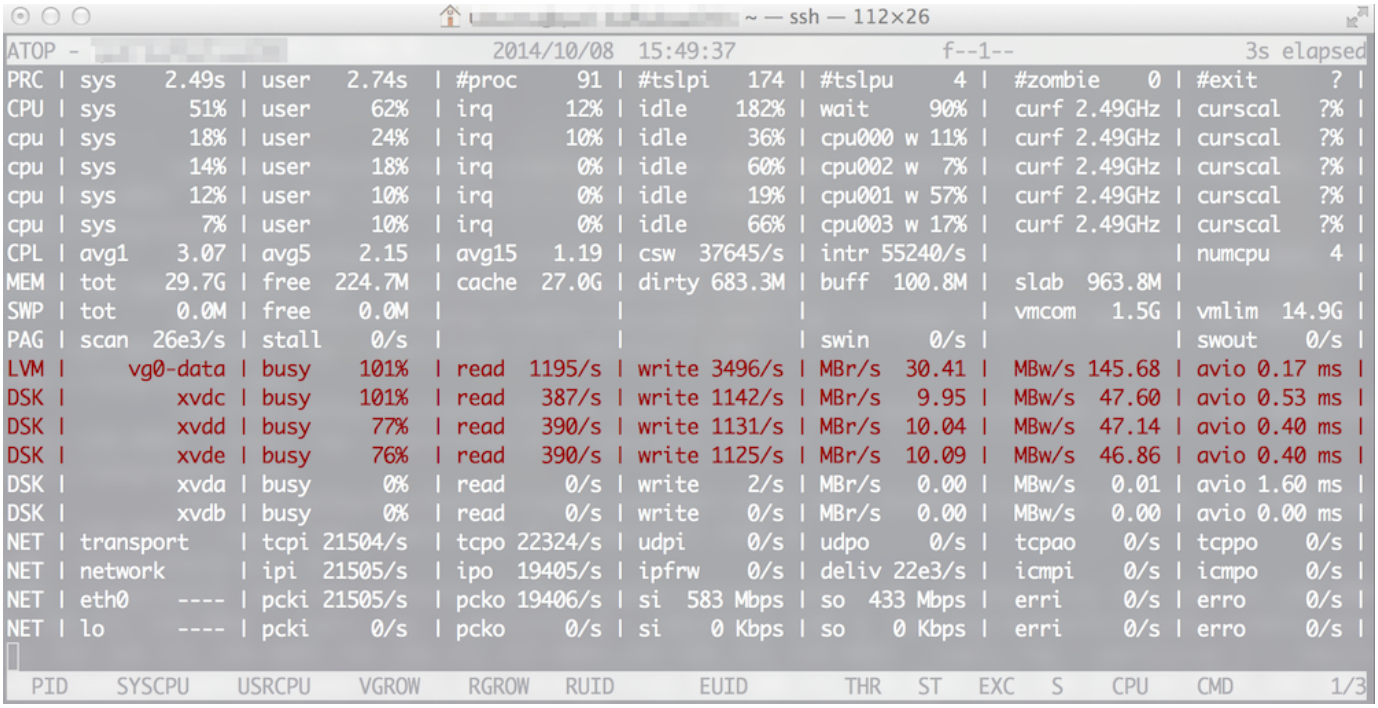
Here's a tricky area that I've alluded some discussion towards with my choice of high memory instances: **consumers can have a huge impact on cluster throughput**. One of the great features of Kafka, it's topic offsets, makes for an opportunity where a consumer can fetch a ton of data from disk. Taking into consideration that everything before this test expresses the importance of disk performance, let's get a visual of what happens when you start screwing around.

The following test is using the same cluster, a 3 partition / 2 replicas topic and a 2500 byte message stream. The 'latest offset' run has a consumer pulling the latest messages from each primary partition of the topic as they come in. This simulates up-to-date consumers. The 'oldest offset' run is the same situation except that the consumers are positioned at the very oldest messages in the topic and we're using a dataset significantly larger than the memory available on each node, meaning the disks will be hit. Hard.

Since the producers are waiting for acks from the brokers, the disk contention of lagging

consumers causes direct impact to the write throughput in addition to severely degraded latency stability.

The reason you don't see this impact with up to date consumers or even with replication is due to the aforementioned intelligent use of the page cache and sendfile(2) syscall. Neither operation will cause disk reads in normal circumstances, but fetching old / uncached data really does it up:



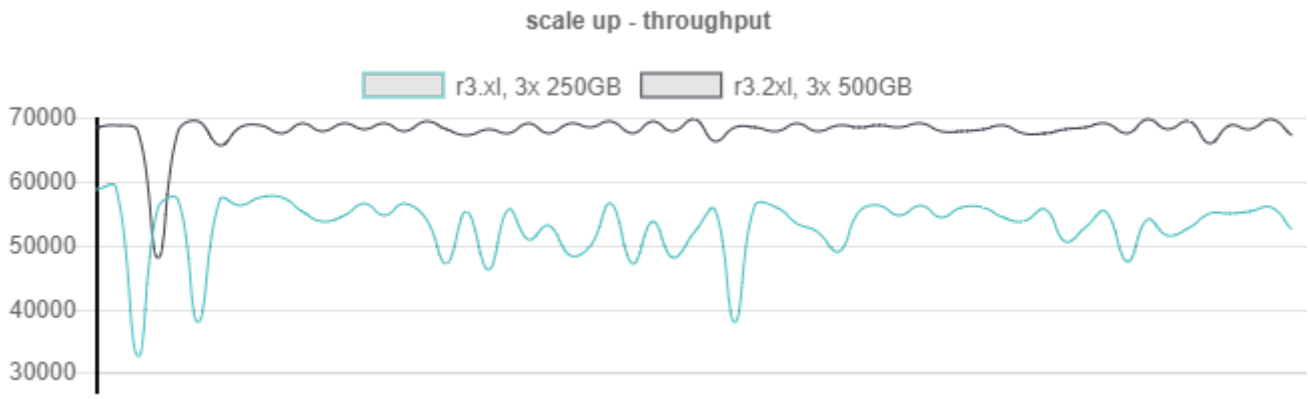
To make a long story short, really think about what retention period is necessary and how consumers may fetch old data and where from.

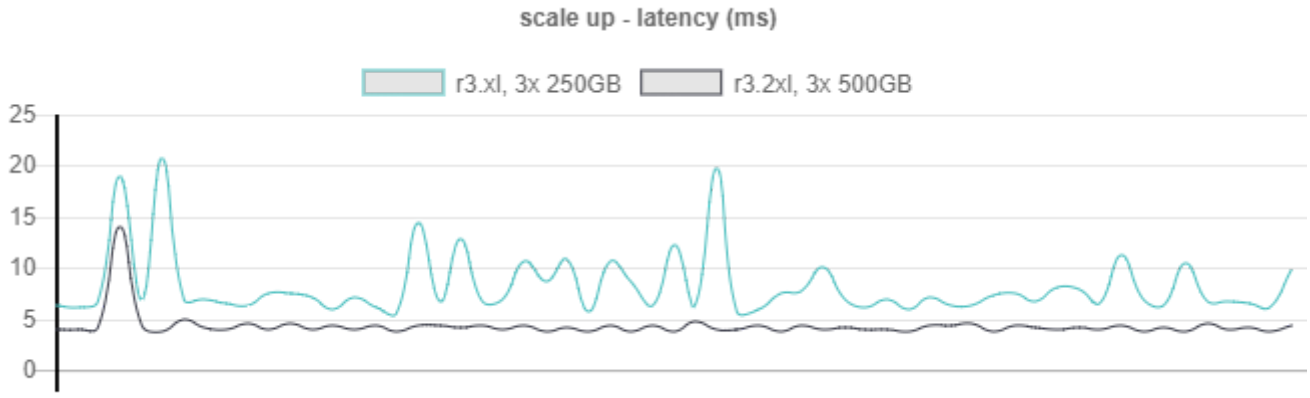
Scaling Up

So now that we've explored the characteristics of a particular Kafka cluster spec in varying circumstances, how much should we expect to gain by scaling up the broker resources?

A bump

Now we're going from a 3 node R3.xlarge each w/ 3x striped 250GB GP SSDs to a 3 node R3.2xlarge (EBS optimized) and 3x 500GB GP volumes in stripe (reminder that the larger size gives us a higher IOPS baseline, plus we're using a larger/dedicated storage channel). These tests are using a 3 partition / 2 replica topic and 2500 byte message sizes.



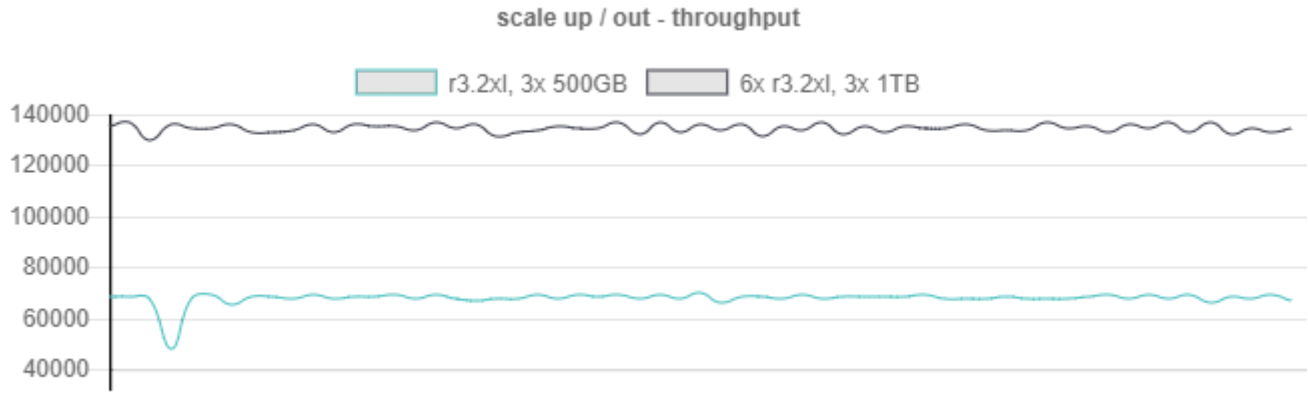


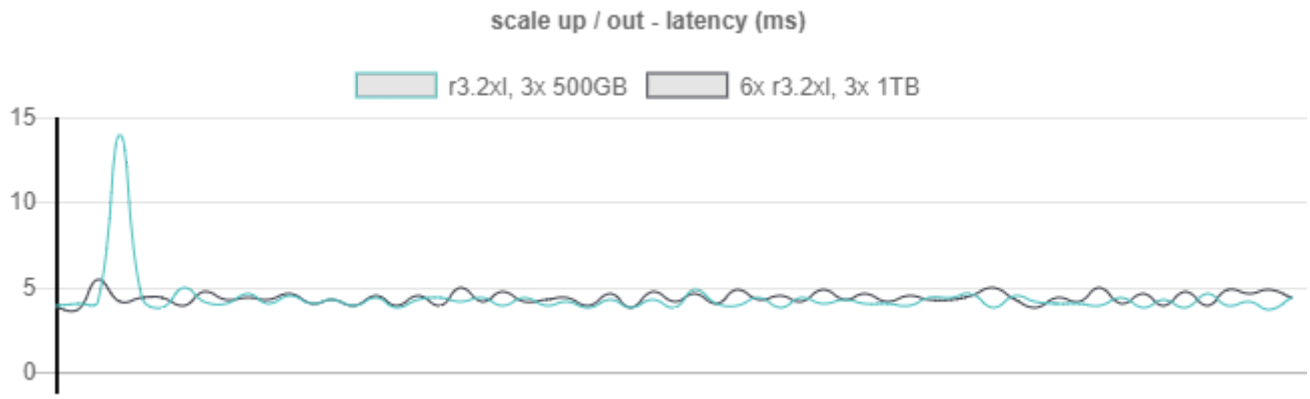
The larger spec actually maxed out the single 32 core Sangrenel box while exhibiting a higher throughput ceiling and lower / more stable latency readings.

By the time we're already bringing in a second c3.8xlarge, why don't we just scale up everything?

Just throwing hardware at it

Now we're comparing the 3 node R3.2xl / 3x 500GB GP stripe per node to a 6 node R3.2xlarge cluster, each node with 3x 1TB GP SSDs in stripe. We're also running 2 C3.8xlarge Sangrenel boxes, each configured to 32 workers and a 2500 byte message size. The larger cluster allows us to span even more primary partitions on physically separate nodes: 6 partitions / 2 replicas for the large cluster and 3 partitions / 2 replicas for the samller cluster.

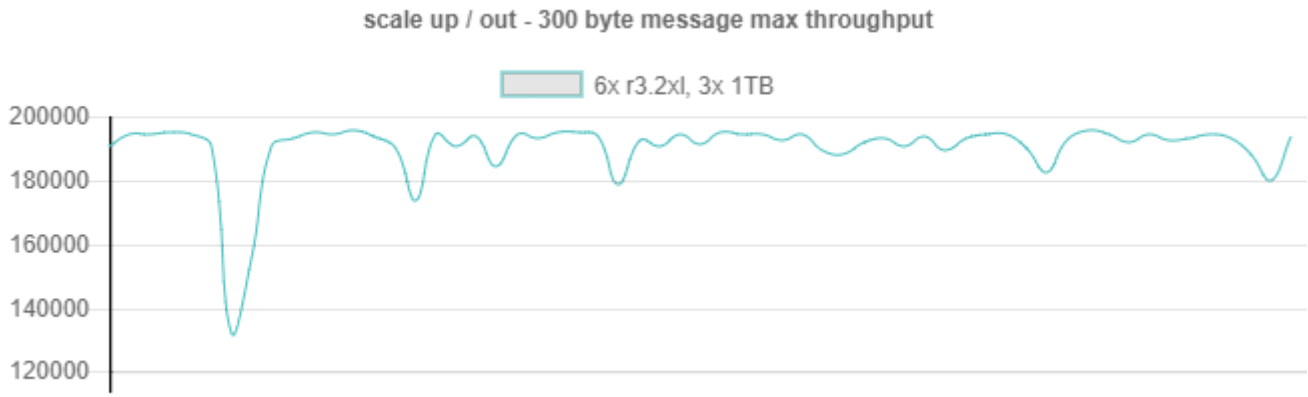


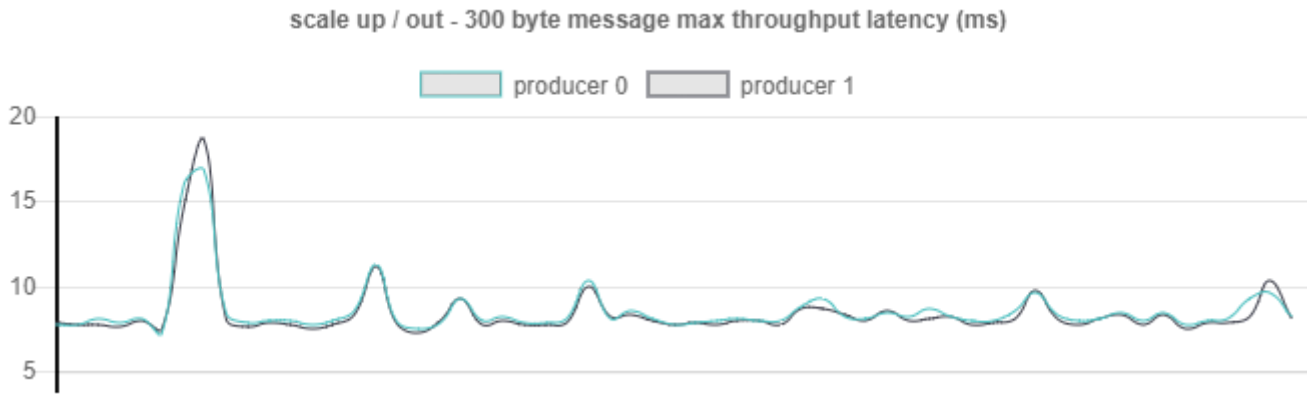


Both setups exhibit great single digit ms latency with stability, but the 6 node setup lets us fire a ton more data through it– at roughly linear scale, an impressive 135K message rate at roughly a 2.6Gb/s ingestion rate.

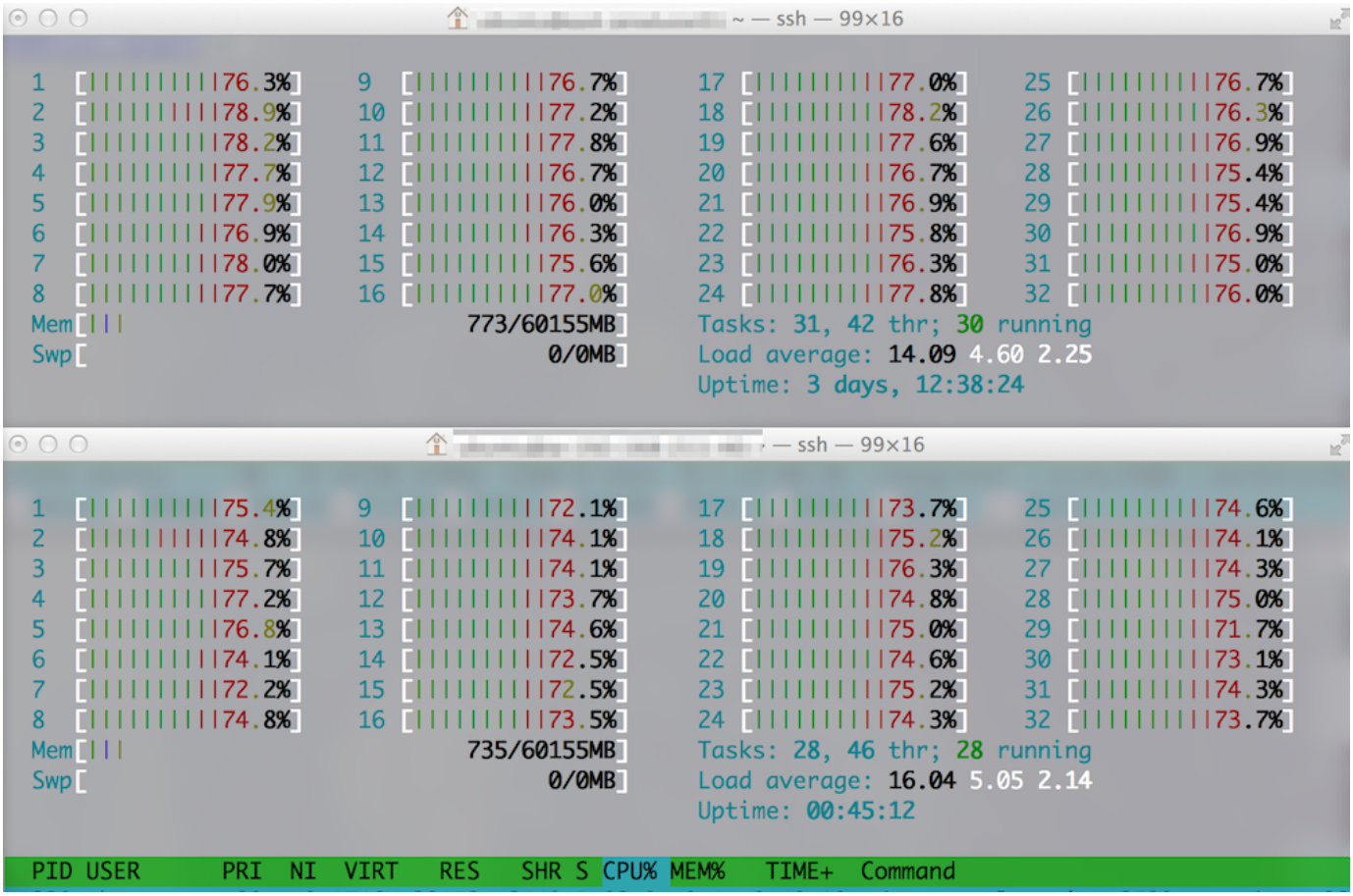
Small messages - max throughput

While I had this up, I figured it would be interesting to capture some numbers on smaller message sizes just for fun. I let two C3.8xlarge Sangrenel boxes rip at full speed with 32 workers and 300 byte message sizes. I also separated the latency per Sangrenel box here.





So, settling at nearly 200K messages/sec. with replicas, broker acks and mostly stable latency is pretty impressive. Equally awesome was 2x C3.8xlarge instances burning up 64 cores:



Closing Thoughts

At least in this limited performance testing, Kafka is one of those systems that I pretty much enjoy every bit of: nothing positive about this software is an accident.

Cheers to the excellent work and articles written by people like Jay Kreps, Neha Narkhede, Jun Rao, and all of the Kafka committers for making software that does a lot of things well.

Back to posts