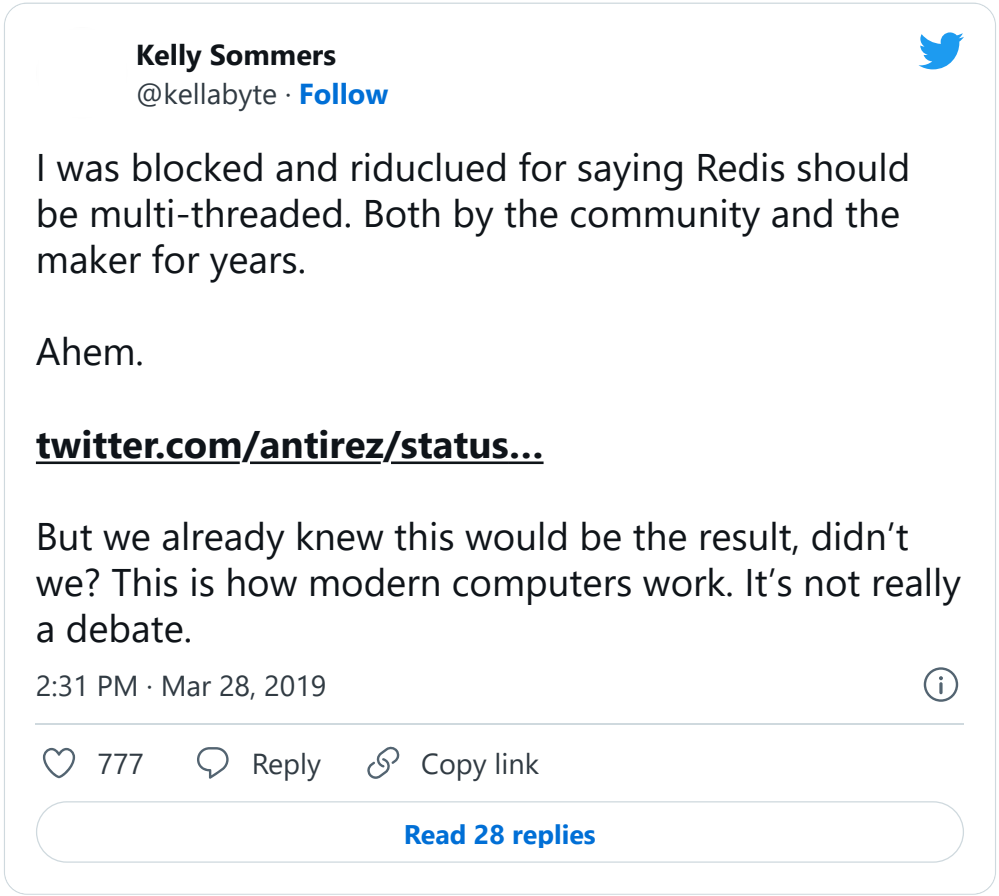
Engineering

REDIS ANALYSIS – PART 1: THREADING MODEL

DECEMBER 9, 2021 BY ROMAN GERSHMAN

Following my previous post, we are going start with the “hottest potato” - single-threaded vs multi-threaded argument.

This question in the context of Redis has been raised quite a few times before, sometimes sparkling pretty heated discussions. See, for example



and

.

Eventually, Salvatore has decided to allow the offloading of I/O processing onto additional threads. But as I said before, only a single designated thread handles the main Redis dictionary.

It seems, however, that was not enough for the community. Two and half years ago, a couple of talented folks from Canada decided to change the status quo and created a multi-threaded Redis fork called *KeyDb*. KeyDb allows multiple threads to access the Redis dictionary by protecting it with spinlocks. See their introductionary post about this.

In the previous post I mentioned the following arguments why Redis was built as single-threaded:

1. It preserves low tail latency by avoiding contention on locks.
2. Redis Cluster provides horizontal scale, which should be as good as

vertical scale if not better: N single-core machines are equivalent to a single process spanning N cores.
3. Pipelining gives you more throughput. A redis client can pipeline requests reaching ~1M qps, which is an order of magnitude higher than the regular traffic throughput.
4. Room for the upside on a single machine is limited anyway.
5. Single-threaded is simple, multi-threaded is complicated.

I think there are great benefits of having multi-threaded databases, and I will try to challenge these arguments. But before we continue forward, let's agree on a few terms:

A vertically scalable system is a system that can scale *almost* linearly with its performance metrics (aka requests per second) as a function of available CPUs on a single server. *Horizontally scalable system* is a distributed system that can run on multiple nodes/processes and scale *almost* linearly with a number of nodes.

I assert the following claims:

1. A vertically scalable system is more cost-efficient than its equivalent horizontally scalable configuration until it reaches its physical limits on a single server.
2. In order to reach the full potential of the modern hardware, and preserve the low latency property, a system should be designed as a shared-nothing architecture, i.e. avoid locking and contention, along with the original philosophy of Redis.

I will try to prove (1). I base (2) on the empirical evidence gathered in the research community and on lessons learned from other well-designed systems like ScyllaDb or Twitter's Pelikan.

Vertical scale vs Horizontal scale

Imagine you are in the business of selling sugar. You need to choose between renting a warehouse that can hold 10,000kg of sugar vs 10 warehouses that can hold 1000kg each. The price of smaller warehouses is precisely a tenth of the bigger one. It seems that there is no difference, right? However, if you choose to rent

10 smaller warehouses, you will see that after some time, some, but not all of them, will be nearly empty, and you need to send trucks to fill them up. Why? Because the randomness of nature dictates that you have almost zero chance that all warehouses are depleted at precisely the same rate. So you need to spend resources to fill some of the warehouses. Then you will spend resources to fill others and so on. Moreover, you need to staff those warehouses: you might need 2.5 people per place on average, but you will need to round it up and hire 3 folks in every place. There will be high-pressure days when your workers will work like crazy, while during other days, they will do nothing. Let's do some math to understand how to model those inefficiencies.

Let's assume that $X_1 \dots X_n$ are independent random variables. Then the expected mean and variance of their sum is the sum of their means and variances:

$$\mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i]$$

$$Var\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} Var[X_i]$$

These formulas are basic rules in probability theory, see here, for example. Specifically, for independent random variables with the same mean μ and standard deviation σ, we get:

$$\mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \mu = n * \mu$$

$$Var\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} Var[X_i] = \sum_{i=1}^{n} \sigma^2 = n * \sigma^2$$

The last equation can be rewritten as
$StdDev\left[\sum_{i=1}^{n} X_i\right] = \sqrt{n} * \sigma.$

These equations prove the so-called square root staffing law in queueing theory, which explains why provisioning a bigger system

is more efficient than provisioning a few smaller ones with equivalent capacity.

Indeed, when we provision a system that handles the load distributed as (μ, σ), we usually take an additional margin, say, twice the standard deviation, to cope with the intrinsic stochastic variability of that load. With n warehouses, we can model their load as n independent variables distributing as (μ, σ), therefore, in order to cope with $n * \mu$ effective load, we will need to staff $2n\sigma$ additional resources. However, a single warehouse that handles the load $(n\mu, \sqrt{n}\sigma)$ needs only $2\sqrt{n}\sigma$ resources to cover the same margin. The bigger warehouse is, the larger the difference between $\sqrt{n}\sigma$ and $n\sigma$.

There are quite a few articles on the internet and lots of academic research in this area. See this post, for example.

Obviously, the vertical scale is equivalent to renting a bigger warehouse, and the horizontal scale is equivalent to provisioning multiple smaller places.

Let's switch back to the memory-store example. Suppose we provision 9 nodes that are expected to host 12GB of data *on average* with the standard deviation - 2GB. We would take servers with 12GB+2*2GB=16GB capacity, in total 144GB with 36GB margin. With a single server, however, we would need 108 + sqrt(9) * 4 = 120GB. We just reduced the overall cost of the system by 17%. And if 17% seems not much, we can compare a single server with arbitrary many n servers of $\frac{1}{n}$ capacity. With n large enough, the standard deviation becomes the significant factor compared to the average load. For example, 108 nodes that host 108GB overall, would need to sustain load $L(\mu = 1, \sigma = 0.577)$, thus we would need to provision 108*(1 + 2*0.577) = 232GB which is 93% higher than the single server cost.

So far, I have talked about economy of scale and why pooling resources is more efficient than employing multiple independent capacities. There are additional factors that increase the total cost

of ownership for multi-node system: I believe that any experienced devop would agree that managing a fleet of N servers is more complicated than managing a single server just because of moving parts: The chance that at least one of the servers will fail is approximately N time bigger than for a single machine. In addition, the horizontally scalable system might impose additional restrictions on how the system is used. Specifically, with Redis - Redis Cluster does not allow transactions or multi-key operations covering multiple nodes, it lacks multi-database support, and it can not issue consistent management operations like flushdb, save across the cluster.

The goal of this section is not to persuade you that vertical scale is strictly better than horizontal scale - obviously, the vertical scale is bounded by the physical limits of its host and can not always be chosen. However, when there is a choice - it can be the much simpler and most cost-efficient alternative to splitting your workloads across separate nodes.

Shared-nothing architecture

"Hardware on which modern workloads must run is remarkably different from the hardware on which current programming paradigms depend, and for which current software infrastructure is designed." - From Scylla blog.

Share-nothing architecture is a methodology to build a system in such way that its state is partitioned among multiple processors, and each processor can execute its work independently from others. Some prominent examples of shared-nothing architecture: a) Map-Reduce is a distributed processing framework that processes huge amounts of data over multiple independent workers. b) Redis Cluster is comprised of independent Redis nodes, where each one of them can perform without relying on others. c) ScyllaDb is a Cassandra clone that partitions its server database over CPUs in such a way that each CPU thread consistently handles the same partition. See more info about their open-sourced Seastar framework d) Similarly, Envoy is a

prominent proxy server that uses thread-local storage to minimize contention between multiple threads inside its process.

Shared-nothing architecture can be designed over multiple nodes like with (a) and (b) or within a single process like with (c) and (d).

In our case, I believe that memory store can be designed as a multi-threaded, single-process system that utilizes the underlying CPUs using shared-nothing architecture. In other words, every cpu thread will handle its dictionary partition shard. Other threads can not access directly the data-structures they do not own. If a thread needs to write or read from a dictionary managed by another thread, it achieves it by sending a message to the owner via a dedicated message bus. This architecture is not novel - it appears a lot in technical papers, and became mainstream thanks to ScyllaDb design.

Any mature database needs to perform operations equivalent to Redis flushdb, save, load etc. It also needs to perform resize or compaction operations periodically. With the single-threaded architecture, it means that a single CPU is involved in processing all this data which heavily reduces database resilience. This brings us to another significant benefit for shared-nothing architecture with the thread-per-core threading model, which is often neglected. Modern servers maintain a bounded ratio of CPU vs memory. Say, in AWS, r5 family has 1:8 ratio, m5 family has 1:4. Similarly, in GCP n2 family maintains ratios between 1-8 GB per vcpu. Therefore, with the thread-per-core model, each thread handles at most K GB of workload, which means that a database stays resilient whether it's 8GB or 860 GB on a single machine.

Benchmarks

This section addresses arguments (3) and (4) from the beginning of the post, namely how much upside we can bring by employing shared-nothing architecture using modern cloud servers. For that, I will use a toy redis-like memory store called midi-redis. midi-

redis is similar to rust tokio mini-redis - i.e. it's built to demonstrate the capabilities of the underlying IO library by implementing a subset of redis/memcached commands.

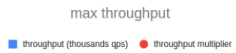
Specifically, midi-redis supports GET, SET, PING, and DEBUG POPULATE commands. It also has complimentary support for memcached GET and SET commands and pipeline mode for both protocols.

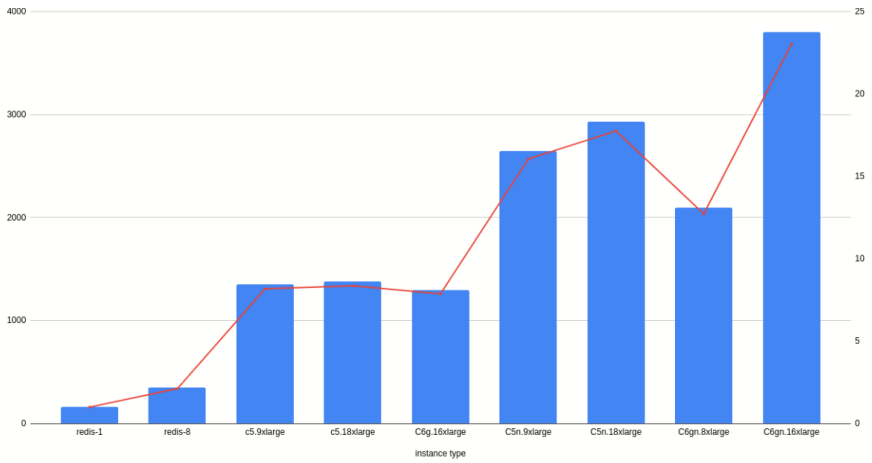
The underlying I/O library that powers midi-redis is helio that uses linux io-uring api underneath. helio is specially designed for shared-nothing architectures and it is the evolution of my previous library GAIA.

I performed benchmarking of midi-redis on a variety of instances on the AWS cloud. The point is not to evaluate midi-redis but to show the true potential of the hardware vs what Redis gives us on this hardware. Please, treat these numbers as directional only - I run each load-test only once, so I believe there is some variance that could affect numbers in +-15% range. redis-1 bar in the graph represents Redis 6 with the io-threads=1 configuration, and redis-8 denotes io-threads=8 configuration. Redis results were similar on all instance types, therefore I used the redis-1 run on c5.18xlarge as my relative baseline for all other runs.

I used read-only traffic to minimize the influence of memory allocations and database resizes on the results. Also, I run "debug populate 10000000" command before each run to fill a server under test with data. Finally, I run memtier_benchmark from 3 client machines in the same zone with the following configuration: --key-prefix=key: --key-pattern=P:P --ratio 0:1 -n 2000000. The number of threads and connections for memtier_benchmark were chosen to maximize the throughput of the server under test and were different for each instance type.

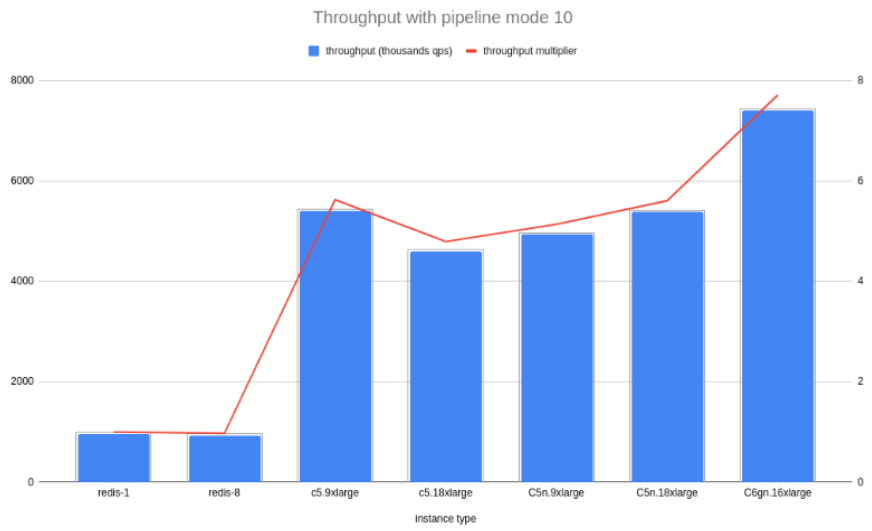
My first graph shows throughput for regular traffic without pipelined requests:





You can see that midi-redis running most network-capable AWS instance c6gn.16xlarge has the throughput that is >20 times higher than redis-1 and >10 times higher than redis-8.

My next graph shows the throughput of instances with pipeline mode when memtier_benchmark sends bursts of 10 requests at once (--pipeline 10):



Here, c6gn.16xlarge has seven times more throughput reaching a staggering 7.4M qps. Interestingly, redis-8 is a bit slower than redis-1 because Redis main thread becomes the bottleneck for pipelined traffic. And redis-8 spends additional cpu for coordination with its io-threads. midi-redis on the other hand, splits its dictionary between all its threads, reduces their communication to a minimum, and scales its performance much better.

I do not know if a factor of 20 or a factor of 7 sounds impressive to you, but please remember that Redis 6 is the product of a decade of development and optimizations. Even 5%, 10% of incremental

improvement is significant here. By changing the foundation, we allow potential 2000% upside for the non-pipeline case. Moreover, with time we will benefit from additional tailwinds from hardware advancements - with better networking and more cpus we will see even higher rates.

Privacy Policy