Dragonfly

Engineering

REDIS ANALYSIS - PART 2: SIMPLICITY

JANUARY 30, 2022 BY ROMAN GERSHMAN

Let's talk about the simplicity of Redis. Redis was initially designed as a simple store, and it seems that its APIs achieved this goal. Unfortunately, Redis's simple design makes it unreliable and difficult to manage in production.

So the question is - what simplicity means to you as a datastore user?

Before diving into Redis specifics, a disclosure: some of the problems I mention below do not show up for small workloads. For those of you, who have managed Redis instances larger than 12GB, please answer the following questions:

- There are eight Redis cache eviction policies. How well do you know them? Are you happy with either one of them?
- How confident are you when you need to change Redis settings, especially those that control its memory consumption? Can you guarantee what its peak memory usage will be?
- Have you ever needed to debug an unresponsive Redis instance or a its OOM crash? How easy was it?
- Have you ever observed connection overload events when

multiple clients connect to a redis instance?

Fourteen years after Redis's inception, we've established that the engineering community demands a simple, low-latency Redis-like API that compliments relational databases. However, the community is unlikely to settle with a fragile technology that is hard to manage. API simplicity does not mean that the foundation should not be solid.

I was fortunate to observe Redis and Memcached usage globally, so my opinion was shaped by looking at the whole range of workloads: I've seen how those "simple" design decisions in Redis caused a sub-optimal but manageable quirks with 4-16GB workloads, made it painful at 64GB scale, and caused frustration and a lack of trust with 100+ GB workloads.

Redis Caching

I mentioned cache policy already. It's one of more significant settings in Redis when used as a cache (a pretty popular use-case for Redis). In a perfect world, a regular user of Redis would love to have a magical cache that does the following:

- Reclaims expired items instead of growing in memory. btw, this requirement holds for non-cache scenarios as well.
- similarly, does not evict non-expired entries if there is a sinificant number of expired items that could be evicted instead.
- maximizes hit-ratio in a robust manner by keeping entries that are most likely to be hit in the future.

A normal user does not want to know what LFU or LRU is, or why Redis implements guesstimate of those heuristics or why it can not evict expired items efficiently. In reality, not only that the user is expected to know the internal implementation details of the Redis cache algorithms, he also needs to decide between eight "simple" options of maxmemory-policy setting.

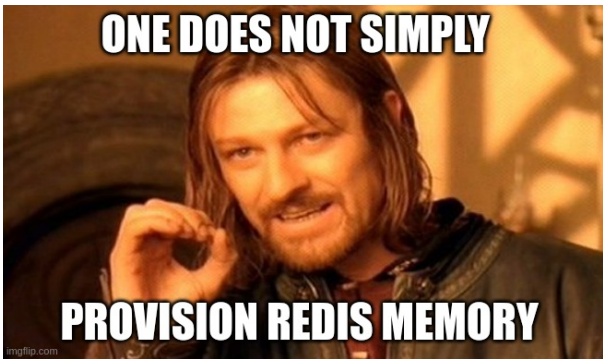
Persistence

If I had to pick a single design choice that looked “simple” at the time yet had a substantial negative impact on the reliability of the whole system and the complexity of other features - it would be the fork-based BGSAVE command. The BGSAVE algorithm allows Redis to produce a point-in-time snapshotting of in-memory data or sync with its secondary replica. Redis does it by forking the serialization routine into a child process. By doing so, this child process gets an immutable, point-in-time snapshot of its parent process memory from Linux for “free”. Redis relies on Linux property that does not copy the physical memory during fork() but uses lazy Copy-On-Write operation instead.

Using the OS to achieve consistent snapshot isolation looks like an elegant choice at first sight. However, there are some serious problems with this approach:

1. **Lack of back-pressure** When the Redis parent process mutates its entries, it in fact duplicates the Linux memory pages with CoW. CoW is transparent to the parent process; therefore the Redis memory component has a hard time estimating its actual memory usage. And even if it did, there is no efficient mechanism in Redis to “postpone” or stall the incoming writes - the execution thread must progress with the flow. Under the heavy writes, this will easily cause OOM crashes.

2. **Unbounded memory overhead** In the worst case, both child and parent processes could double Redis physical memory. Unfortunately, it does not stop there with replica syncs: the parent must also hold the replication log of mutations during the snapshot, which grows in memory until the sync completes. In addition, the parent dataset can grow beyond its initial size because the user continues adding items. These factors can contribute to RSS usage spiking wildly under different write loads or database sizes. All this makes it very hard to estimate the maximum memory usage for Redis.



3. **Linux large pages** Enabling large pages usually improves the database performance; however, with Redis and bgsave, huge pages would create high write amplification - a tiny write would cause 2MB or 1GB CoW. This would quickly cause 100% memory overhead and major latency spikes during writes.
4. **Bad interactions with other Redis features** Ask Redis maintainers how hard it was to implement TLS support in Redis 6. The seemingly unrelated feature had several problems due to how the TLS session interacted with the fork() call. As a result: TLS in Redis has mediocre performance.

So far, I've mentioned a few stability problems with Redis that are impossible to fix with the current Redis architecture. There is a long tail of additional problems that hurt Redis reliability, impact its resource usage or obscure its API guarantees. Here are some of them in random order:

- Unreliable replica syncs: if the master's replication buffer overflows during replica sync, the replica will retry the whole synchronization flow, possibly entering the infinite cycle of never-ending attempts.
- Unreliable timeouts in blocking commands: blocking commands can expire with timeouts much larger than specified.

- Uncontrolled freezes: commands like FLUSHDB will "stop the world" during their execution. This means Redis may completely stall the processing of ongoing requests for minutes or longer.
- LUA stalling: a similar problem with unresponsiveness exists when running bad LUA scripts. Moreover, even good scripts may cause significant delays to other concurrent requests due to the sequential nature of Redis execution.
- PUB/SUB is unreliable and prone to data loss when a SUB client disconnects.

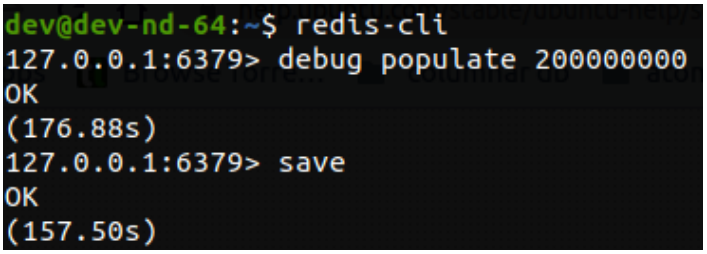
Fight against complexity

I think it's time to redesign the system that once challenged traditional databases but nowadays suffers from complexity itself.

Lately, I've been working on a novel design of a cache and dictionary data structures that could be the foundation stones for the next-generation memory store. The work is still in progress, but it already shows some promising results. The cache design is so novel that I think it deserves a blog post of its own. Therefore, today I will share just the basic characteristics of the underlying dictionary.

Below you can see a Redis 6.2 vs my experimental store (POC), both running on a dedicated 64-cpu n2d instance in GCP.

I run the same three commands on both servers: debug populate, save and flushdb. populate is interesting because it demonstrates the raw efficiency of the underlying engine, without bottlenecks like networking. save and flushdb are interesting because both are "stop the world" commands that must process the whole database and their performance directly affects database robustness.

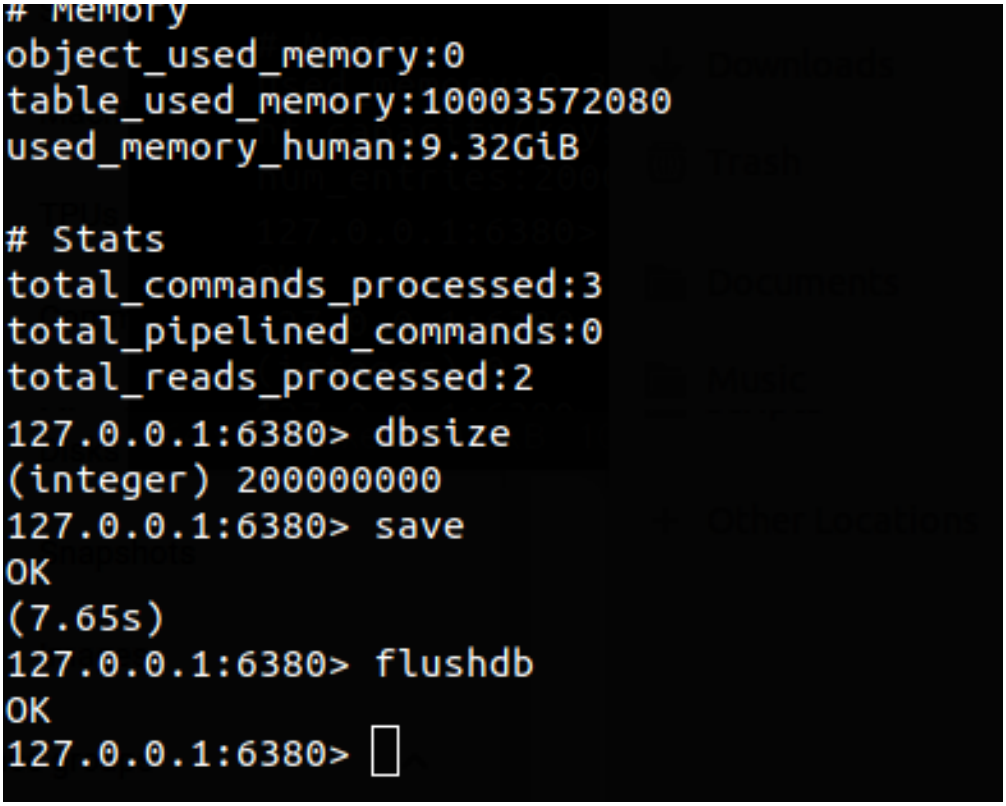


```
127.0.0.1:6379> info memory
# Memory
used_memory_human:16.89G
used_memory_rss:18380734464
used_memory_rss_human:17.12G
used_memory_peak_human:16.89G
used_memory_dataset:7992041088
127.0.0.1:6379> flushdb
OK
(176.89s)
127.0.0.1:6379>
```

As you can see it takes almost 180s to create 200M items in Redis. What's remarkable about this number is that it sets an upper bound on write throughput limit for Redis: no matter how big the Redis server is, it won't be able to go faster than ~1.1M records a second because records creation is always done entirely in a single thread. Saving 200M records takes 150s which shows how quickly a server can persist its data or replicate itself to a replica. Note the gap: those 200M items take up 17GB of RAM, hence Redis moves 17GB in 150s or 110MB/s (for comparison, the slowest hdd in AWS (sc1) reaches 250MB/s and in GCP (pd-standard) reaches 400MB/s). Finally, flushdb demonstrates that a simple "empty my store" operation may be cpu-intensive, and completely stall other requests for almost 3 minutes! Not so simple anymore.

This snapshot below is the POC store under development.

```
dev@dev-nd-64:~$ redis-cli -p 6380
127.0.0.1:6380> debug populate 200000000
OK
(3.52s)
127.0.0.1:6380> info
# Server
redis_version:6.2.0
redis_mode:standalone
arch_bits:64
multiplexing_api:iouring
atomicvar_api:atomic-builtin
tcp_port:6380
```



You can see that the same commands run an order of magnitude faster. The result is achieved in part because of the shared-nothing architecture that distributes operations across cpus but also due to novel dictionary design. Btw, you can see that flushdb operation was not timed - redis-cli does not show latencies of "fast" operations below 500ms. Hence, we can conclude that flushdb was at least 350 times faster in this case. If you compare used_memory_human metric, you can see that Redis required almost twice more RAM than the POC (16.9GB vs 9.5GB).

There is a lot more to cover. For example, Redis SAVE stalls the processing of all requests, similarly to FLUSHDB . In contrast, the new store runs it concurrently with the rest of the traffic while still producing a consistent point-in-time snapshot. In other words, it provides BGSAVE product experience with the reliability of SAVE.

If we combine the long tail of improvements that come from a new design, we will get a product that redefines what simplicity and ease of use mean for an in-memory database world.

Privacy Policy