Random notes on improving the Redis LRU algorithm

antirez 1018 days ago. 214893 views.

Redis is often used for caching, in a setup where a fixed maximum memory to use is specified. When new data arrives, we need make space by removing old data. The efficiency of Redis as a cache is related to how good decisions it makes about what data to evict: deleting data that is going to be needed soon is a poor strategy, while deleting data that is unlikely to be requested again is a good one.

In other terms every cache has an hits/misses ratio, which is, in qualitative terms, just the percentage of read queries that the cache is able to serve. Accesses to the keys of a cache are not distributed evenly among the data set in most workloads. Often a small percentage of keys get a very large percentage of all the accesses. Moreover the access pattern often changes over time, which means that as time passes certain keys that were very requested may no longer be accessed often, and conversely, keys that once were not popular may turn into the most accessed keys.

So in general what a cache should try to do is to retain the keys that have the highest probability of being accessed in the future. From the point of view of an eviction policy (the policy used to make space to allow new data to enter) this translat into the contrary: the key with the least probability of being accessed in the future should be removed from the data set. There is only one problem: Redis and other caches are not able to predict the future.

The LRU algorithm
===

While caches can’t predict the future, they can reason in the following way: keys that are likely to be requested again are keys that were recently requested often. Since usually access patterns don’t change very suddenly, this is an effective strategy. However the notion of “recently requested often” is more insidious that it may look at a first glance (we’ll return shortly on this). So this concept is simplified into an algorithm that is called LRU, which instead just tracks the *last tim a key was requested. Keys that are accessed with an higher frequency have a greater probability of being idle (not accessed) for a shorter time compared to keys that are rarely accessed.

For instance this is a representation of four different keys accesses over time. Each “~” character is one second, while the “|” line at the end is the current instant.

```
~~~~~A~~~~~A~~~~~A~~~~A~~~~~A~~~~~A~~|
~~B~~B~~B~~B~~B~~B~~B~~B~~B~~B~~B~~B~|
~~~~~~~~~~C~~~~~~~~~C~~~~~~~~~C~~~~~~|
~~~~~D~~~~~~~~~~D~~~~~~~~~D~~~~~~~~~D|
```

Key A is accessed one time every 5 seconds, key B once every 2 seconds
and key C and D are both accessed every 10 seconds.

Given the high frequency of accesses of key B, it has among the lowest idle
times, which means its last access time is the second most recent among all the
four keys.

Similarly A and C idle time of 2 and 6 seconds well reflect the access
frequency of both those keys. However as you can see this trick does not
always work: key D is accessed every 10 seconds, however it has the most
recent access time of all the keys.

Still, in the long run, this algorithm works well enough. Usually keys
with a greater access frequency have a smaller idle time. The LRU
algorithm evicts the Least Recently Used key, which means the one with
the greatest idle time. It is simple to implement because all we need to
do is to track the last time a given key was accessed, or sometimes
this is not even needed: we may just have all the objects we want to
evict linked in a linked list. When an object is accessed we move it
to the top of the list. When we want to evict objects, we evict from
the tail of the list. Tada! Win.

LRU in Redis: the genesis
===

Initially Redis had no support for LRU eviction. It was added later, when memory efficiency was a big concern. By modifying a bit the Redis Object structure I was able to make 24 bits of space. There was no room for linking the objects in a linked lis (fat pointers!), moreover the implementation needed to be efficient, since the server performance should not drop too much because of the selection of the key to evict.

The 24 bits in the object are enough to store the least significant
bits of the current unix time in seconds. This representation, called

bits of the current unix time in seconds. This representation, called

"LRU clock" inside the source code of Redis, takes 194 days to overflow. Keys metadata are updated much often, so this was go
enough.

However there was another more complex problem to solve, how to select
the key with the greatest idle time in order to evict it? The Redis
key space is represented via a flat hash table. To add another data
structure to take this metadata was not an option, however since
LRU is itself an approximation of what we want to achieve, what
about approximating LRU itself?

The initial Redis algorithm was as simple as that: when there is to evict
a key, select 3 random keys, and evict the one with the highest
idle time. Basically we do random sampling over the key space and evict
the key that happens to be the better. Later this "3 random keys"
was turned into a configurable "N random keys" and the algorithm
speed was improved so that the default was raised to 5 keys sampling
without losing performances. Considering how naive it was, it worked
well, very well actually. If you think at it, you always never do
the best decision with this algorithm, but is very unlikely to do
a very bad decision too. If there is a subset of very frequently accessed
keys in the data set, out of 5 keys it is hard to be so unlucky to
only sample keys with a very short idle time.

However if you think at this algorithm *across* its executions, you
can see how we are trashing a lot of interesting data. Maybe when
sampling the N keys, we encounter a lot of good candidates, but
we then just evict the best, and start from scratch again the next
cycle.

First rule of Fight Club is: observe your algorithms with naked eyes
===

At some point I was in the middle of working at the upcoming Redis
3.0 release. Redis 2.8 was actively used as an LRU cache in multiple
environments, and people didn't complained too much about the
precision of the eviction in Redis, but it was clear that it could
be improved even without using a noticeable amount of additional CPU
time, and not a single bit of additional space.

However in order to improve something, you have to look at it. There
are different ways to look at LRU algorithms. You can write, for example,
tools that simulate different workloads, and check the hit/miss ratio
at the end. This is what I did, however the hit/miss ratio depends
a lot on the access pattern, so additionally to this information I
wrote an utility that actually displayed the algorithm quality in a
visual way.

The program was very simple: it added a given number of keys, then
accessed the keys sequentially so that each had a decreasing
idle time. Finally 50% more keys were added (the green ones in the
picture), so that half of the old keys needed to be evicted.

In a perfect LRU implementation no key from the new added keys are evicted, and the initial 50% of the old dataset is evicted

This is the representation produced by the program for different
versions of Redis and different settings:

http://redis.io/images/redisdoc/lru_comparison.png

When looking at the graph remember that the implementation we
discussed so far is the one of Redis 2.8. The improvement you
see in Redis 3.0 is explained in the next section.

LRU V2: don't trash away important information
===

With the new visual tool, I was able to try new approaches and
test them in a matter of minutes. The most obvious way to improve
the vanilla algorithm used by Redis was to accumulate the otherwise
trashed information in a "pool" of good candidates for eviction.

Basically when the N keys sampling was performed, it was used to
populate a larger pool of keys (just 16 keys by default).
This pool has the keys sorted by idle time, so new keys only enter
the pool when they have an idle time greater than one key in the
pool or when there is empty space in the pool.

This small change improved the performances of the algorithm
dramatically as you can see in the image I linked above and
the implementation was not so complex. A couple memmove() here
and there and a few profiling efforts, but I don't remember
major bugs in this area.

At the same time, a new redis-cli mode to test the LRU precision
was added (see the --lru-test option), so I had another way to
check the performances of the LRU code with a power-law access
pattern. This tool was used to validate with a different test that
the new algorithm worked better with a more real-world-ish workload.
It also uses pipelining and displays the accesses per second, so
can be used in order to benchmark different implementations, at least
to check obvious speed regressions.

Least Frequently Used
===

The reason I'm writing this blog post right now is because a couple
of days ago I worked at a partial reimplementation and to different
improvements to the Redis cache eviction code.

Everything started from an open issue: when you have multiple databases
with Redis 3.2, the algorithm evicts making local choices. So
if for example you have all keys with a small idle time in DB number 0,
and all keys with large idle time in DB number 1, Redis will evict
one key from each DB. A more rational choice is of course to start
evicting keys from DB number 1, and only later to evict the other keys.

This is usually not a big deal, when Redis is used as a cache it is
rarely used with different DBs, however this is how I started to work
at the eviction code again. Eventually I was able to modify the pool
to include the database ID, and to use a single pool for all the DBs
instead of using multiple pools. It was slower, but by profiling and
tuning the code, eventually it was faster than the original
implementation by around 20%.

However my curiosity for this subsystem of Redis was stimulated again
at that point, and I wanted to improve it. I spent a couple of days
trying to improve the LRU implementation: use a bigger pool maybe?
Account for the time that passes when selecting the best key?

After some time, and after refining my tools, I understood that the
LRU algorithm was limited by the amount of data sampled in the database
and was otherwise very good and hard to improve. This is, actually,
kinda evident from the image showing the different algorithms:
sampling 10 keys per cycle the algorithm was almost as accurate as
theoretical LRU.

Since the original algorithm was hard to improve, I started to test
new algorithms. If we rewind a bit to the start of the blog post, we
said that LRU is actually kinda a trick. What we really want is to
retain keys that have the maximum probability of being accessed in the
future, that are the keys *most frequently accessed*, not the ones with
the latest access.

The algorithm evicting the keys with the least number of accesses
is called LFU. It means Least Frequently Used, which is the feature of
the keys that it attempts to kill to make space for new keys.

In theory LFU is as simple as associating a counter to each key. At
every access the counter gets incremented, so that we know that a given
key is accessed more frequently than another key.

Well, there are at least a few more problems, not specific to Redis,
general issues of LFU implementations:

1. With LFU you cannot use the “move to head” linked list trick used for LRU in order to take elements sorted for eviction in simple way, since keys must be ordered by number of accesses in “perfect LFU”. Moving the accessed key to the right place can be problematic because there could be many keys with the same score, so the operation can be O(N) in the worst case, even if the key frequency counter changed just a little. Also as we’ll see in point “2” the accesses counter does not always change just a little, there are also sudden large changes.

2. LFU can’t really be as trivial as, just increment the access counter on each access. As we said, access patterns change over time, so a key with an high score needs to see its score reduced over time if nobody keeps accessing it. Our algorithm must be albe to adapt over time.

In Redis the first problems is not a problem: we can just use the trick used for LRU: random sampling with the pool of candidates. The second problem remains. So normally LFU implementations have some way in order to decrement, or halve the access counter from time to time.

Implementing LFU in 24 bits of space
===

LFU has its implementation peculiarities itself, however in Redis all we can use is our 24 bit LRU field in order to model LFU. To implement LFU in just 24 bits per objects is a bit more tricky.

What we need to do in 24 bits is:

1. Some kind of access frequency counter.
2. Enough information to decide when to halve the counter.

My solution was to split the 24 bits into two fields:

```
          16 bits      8 bits
     +----------------+--------+
     + Last decr time | LOG_C  |
     +----------------+--------+
```

The 16 bit field is the last decrement time, so that Redis knows the last time the counter was decremented, while the 8 bit field is the actual access counter.

You are thinking that it’s pretty fast to overflow an 8 bit counter, right? Well, the trick is, instead of using just a counter, I used a logarithmic counter. This is the function that increments the counter during accesses to the keys:

```
  uint8_t LFULogIncr(uint8_t counter) {
      if (counter == 255) return 255;
      double r = (double)rand()/RAND_MAX;
      double baseval = counter - LFU_INIT_VAL;
      if (baseval < 0) baseval = 0;
      double p = 1.0/(baseval*server.lfu_log_factor+1);
      if (r < p) counter++;
      return counter;
  }
```

Basically the greater is the value of the counter, the less probable is that the counter will really be incremented: the code above computes a number ‘p’ between 0 and 1 which is smaller and smaller as the counter increases. Then it extracts a random number ‘r’ between 0 and 1 and only increments the counter if ‘r < p’ is true.

You can configure how much aggressively the counter is implemented via redis.conf parameters, but for instance, with the default settings, this is what happens:

After 100 hits the value of the counter is 10
After 1000 is 18

After 100k is 142
After 1 million hits it reaches the 255 limit and no longer increments

Now let’s see how this counter is decremented. The 16 bits are used in order to store the less significant bits of the UNIX time converted to minutes. As Redis performs random sampling scanning the key space in search of keys to populate the pool, all keys that are encountered are checked for decrement. If the last decrement was performed more than N minutes ago (with N configurable), the value of the counter is halved if it is an high value, or just decremented if it is a lower value (in the hope that we can better discriminate among keys with few accesses, given that our counter resolution is very small).

There is yet another problem, new keys need a chance to survive after all. In vanilla LFU a just added key has an access score of 0, so it is a very good candidate for eviction. In Redis new keys start with an LFU value of 5. This initial value is accounted in the increment and halving algorithms. Simulations show that with this change keys have some time in order to accumulate accesses: keys with a score less than 5 will be preferred (non active keys for a long time).

Code and performances
===

The implementation described above can be found in the “unstable” branch of Redis. My initial tests show that it outperforms LRU in power-law access patterns, while using the same amount of memory per key, however real world access patterns may be different: time and space locality of accesses may change in very different ways, so I’ll be very happy to learn from real world use cases how LFU is performing, and how the two parameters that you can tune in the Redis LFU implementation change the performances for different workloads.

Also an OBJECT FREQ subcommand was added in order to report the frequency counter for a given key, this can be both useful in order to observe an application access pattern, and in order to debug the LFU implementation.

Note that switching at runtime between LRU and LFU policies will have the effect to start with almost random eviction, since the metadata accumulated in the 24 bits counter does not match the meaning of the newly selected policy. However over time it adapts again.

There are probably many improvements possible.

Ben Manes pointed me to this interesting paper, describing an algorithm called TinyLRU (http://arxiv.org/pdf/1512.00727.pdf).

The paper contains a very neat idea: instead of remembering the access frequency of the current objects, let’s (probabilistically) remember the access frequency of all the objects seen so far, this way we can even refuse new keys if, from the name, we believe they are likely to get little accesses, so that no eviction is needed at all, if evicting a key means to lower the hits/misses ratio.

My feeling is that this technique, while very interesting for plain GET/SET LFU caches, is not applicable to the data structure server nature of Redis: users expect the key to exist after being created at least for a few milliseconds. Refusing the key creation at all seems semantically wrong for Redis.

However Redis maintains LFU informations when a key is overwritten, so for example after a:

    SET oldkey some_new_value

The 24 bit LFU counter is copied to the new object associated to the old key.

The new eviction code of Redis unstable contains other good news:

1. Policies are now “cross DB”. In the past Redis made local choices as explained at the start of this blog post. Now this is fixed for all the policies, not just LRU.

2. The volatile-ttl eviction policy, which is the one that evicts based on the remaining time to live of keys with an expire set, now uses the pool like the other policies.

3. Performances are better by reusing SDS objects in the pool of keys.

This post ended a lot longer than I expected it to be, but I hope it offered a few insights on the new stuff and the improvements to the old things we already had. Redis, more than a “solution” to solve a specific problem, is a generic tool. It’s up to the sensible developer to apply it in the right way. Many people use Redis as a caching solution, so improvements this area are always investigated from time to time.

Hacker News comments: https://news.ycombinator.com/item?id=12185534

: