engineering.kablamo.com.au

Memcached vs Redis - More Different Than You Would Expect

10.10.2021 - By Ben Boyter

29-36 minutes

Important Note

Everything that follows should be only considered accurate at time of publishing. Both projects are not considered finished, and so changes that follow publication could invalidate some or all of the information below.

Redis vs Memcached

Redis vs Memcached. Another entry to the list of software holy wars with Emacs vs. Vi, Tabs vs. Spaces, IDE vs. Editor. I have heard a lot of people saying why they use one over the other, and a lot of wrong statements while doing it.

I kept getting questioned when I asked why people were making these choices, and as such started reading in depth into how they worked under the hood. The idea being to be able to intelligently talk in depth about them. I found it really interesting and so I kept reading and learning.

What follows is what I learnt so far. I turned it into a brownbag presentation some time ago, then took the core ideas of that into this blog post. From how they use memory to how they expire items, the differences are not as subtle as you would expect.

Obvious Differences

Let's start by discussing the obvious differences between Redis and memcached. The first and most obvious is that Redis supports multiple data types. Those being,

- strings
- lists
- sets
- hashes
- sorted sets
- bitmaps
- hyperloglogs

Redis also has some other functionality such as,

- disk persistence
- message broker
- lua scripting
- transactions
- pub/sub
- geospacial support

None of the the above are going to be covered in any detail. If you do need any of the above functionality, then the answer is use Redis, don't read any further and go for a walk outside.

So you want to use Redis primarily as a cache?

However if you want to use Redis primarially as a cache, then the differences are less obvious.

Both Redis and memcached provide atomic get/set/delete operations, automatic expiration of old items, sub millisecond latency, scale to available RAM and are clusterable. So are they the same?

Lets consider their history.

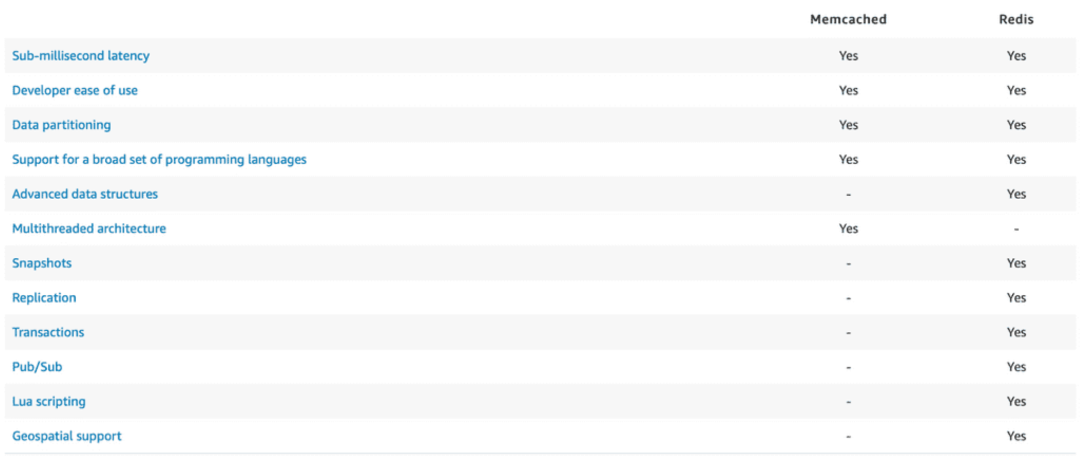
Memcached was created in 2003 and written in perl before being rewritten in C. Orginally created for livejournal it became one of the goto stack enhancements of the Web 2.0 era. It's still in use by very large web properties such as Youtube, Reddit, Facebook, Pinterest, Twitter, Wikipedia and more.

Its original design goal was to leverage unused RAM on web servers, pooling them together to provide a large high speed in-memory cache. It was especially useful for PHP websites allowing you to store state in a stateless programming envrionment without having to write to the database.

Redis was created in 2009, prototyped in TCL and then also rewritten in C. Originally it was written to speed up Salvatore's (the creator's) startup. It is also in use by very large web properties including GitHub, Instagram, Stackoverflow and Craigslist.

It's orginal design goal was to solve Salvatores problems and be a persistent in-memory datastructure store, hence why it supports so many other data types. It became an especially popular addition to the Ruby on Rails stack, probably due to excellent library support and integration.

Neither of the above really differenciates them, and neither does the AWS product page for ElastiCache.



	Memcached	Redis
Sub-millisecond latency	Yes	Yes
Developer ease of use	Yes	Yes
Data partitioning	Yes	Yes
Support for a broad set of programming languages	Yes	Yes
Advanced data structures	-	Yes
Multithreaded architecture	Yes	-
Snapshots	-	Yes
Replication	-	Yes
Transactions	-	Yes
Pub/Sub	-	Yes
Lua scripting	-	Yes
Geospatial support	-	Yes

If you pick software like your typical large enterprise company then the higher number of checkmarks makes Redis the clear choice from this page. However if I am only going to be caching things then is it still the better choice?

A quick scan of the internet produces all sorts of interesting arguments suggesting to use one over the other. A few choice arguments I have included below,

- Use Redis, its newer! (as if the code in memcached has rusted somehow)
- Memcached uses LRU! Its an inferior cache expiry algorithm.
- Memcached has less functionality! (probably not an issue if you use Redis as a cache only)
- bUt PhP tUtOrIaL's uSeS iT… (memcached that is... but yes, lets ignore tools based on percieved coolness)
- Redis is better supported, updated more often. (or maybe memcached is "finished" or has a narrower scope?)
- Memcached supports threads! So it scales!

Both memcached and redis have frequent commits and releases, so any arguments about frequency of updates are moot.

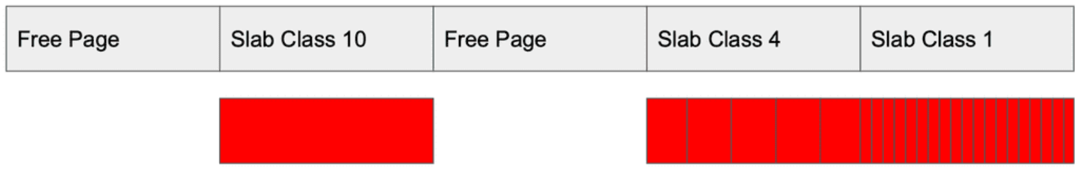
As mentioned for the basic get/set/delete operations both Redis and memcached provide atomic operations, O(1) time for any operation, have sub millisecond latency and scale in line with your systems RAM.

One other point of difference though is that Redis supports transactions, so you can bundle a few of those get/set/deletes into a single unit if you need.

Clearly we need to look harder to determine the differences between them and hopefully work out which one we should be using.

How Memcached Organises memory

Memcached organises memory into pages, slabs and chunks. When you start the service you define how much memory memcached is allowed to use to the nearest MB. Memcached on starting will allocate that much memory and then (by default) break it into 1 MB pages. These pages when first allocated are empty and called free pages.

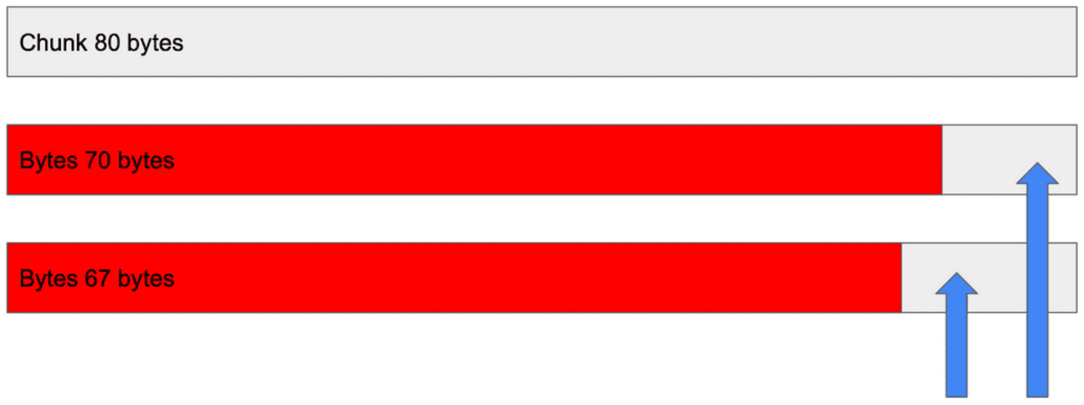


Once you add an item to memcached it looks at the size in bytes of what you are adding, and then assignes a free page a slab class. The slab class determines the number of chunks, containing items that go into the slab. By default the smallest chunk size is 80 bytes.

Say you add a 80 byte item into your freshly started memcached instance. It would pick the first free page, set its slab class to support 80 byte chunks, then put you item into the first chunk in that slab. So given the default 1 MB page size, by allocating its slab class to support 80 bytes chunks memcached would be able to store 13,107 80 byte items into it. If you then add another 80 byte item, it would go into the same slab as the previously added item.

If however you then add a 500 byte item, memcached would find a new free page, allocate a new slab size and store the item there. Futher items of the same size would fill the same slabs pages up to its limit.

There is a tradeoff that memcached makes when doing this, which is wasted cache space.



Say you wanted to store a 70 byte item in memcached. Given that 80 bytes is the smallest chunk size, when memcached stores the item there is an overhead of 10 bytes out of the 80 which remains unused. The memory is still taken, but adding a 10 byte item will not be allocated to that space, instead a new chunk will be used, with 70 unused bytes.

Its a design trade off that memcached makes in order to ensure that memory never fragments.

The 1 MB page/slab size does mean that by default the largest item you can store in memcached is 1 MB in size. While you can change this... there is a quote I read somewhere about memcached,

A wise person knows you can increase the page size, a wiser one asks how to reduce it.

You can actually view the slab/chunk allocations if you have memcached installed and command line access as follows.

```
slab class   1: chunk size  80 perslab 13107
slab class   2: chunk size   100 perslab 10485
slab class   3: chunk size   128 perslab  8192
slab class   4: chunk size   160 perslab  6553
slab class   5: chunk size   200 perslab  5242
slab class   6: chunk size   252 perslab  4161
slab class   7: chunk size   316 perslab  3318
slab class   8: chunk size   396 perslab  2647
slab class   9: chunk size   496 perslab  2114
```

Chunk sizes increase by a factor of 1.25 (rounded up to the next power of two). So given the smallest chunk size being 80 bytes means the next size is 100 bytes, and then 128 bytes.

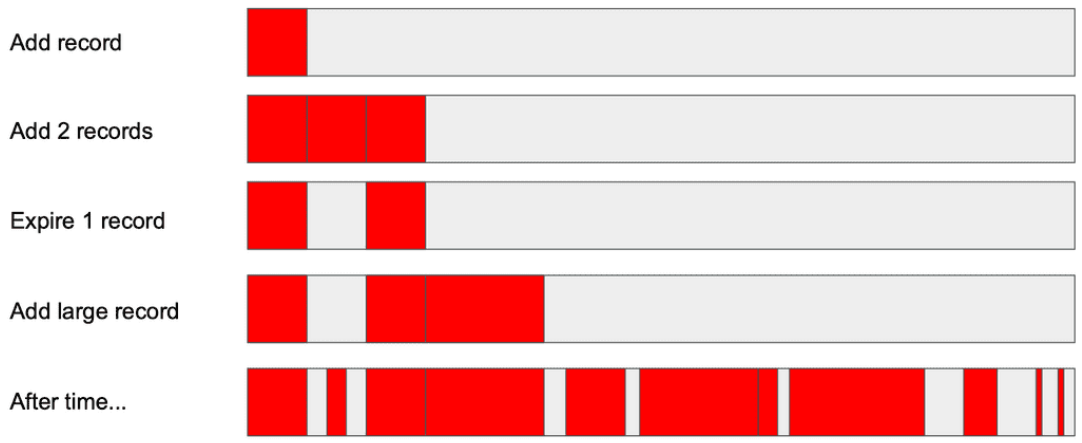
Historically memcached used to increase the chunk size by powers of two which due to the wasted chunk storage issue mentioned resulted in a 25% overhead on storage usage. The 1.25 factor produces a closer to 10% overhead. This was one of the improvements that Facebook added to the codebase due to their heavy memcached usage.

So what are the benefits and overheads of this technique? Well the first is that memory never fragments in memcached. As such there is no need to compact it, and as such there are no background processes needed to rearrange items store. In addition there is no need to ever clean memory, you can just overwrite the existing slab.

There are some negatives. The first being the memory overhead that wastes available space. It all comes down to a speed / memory trade off that memcached uses. Memcached can and does reassign slab classes. If the slab is emptied it can be reassigned with a new class.

How Redis Organises memory

Redis by contrast allocates memory on a per item stored basis. When an item comes into Redis, it allocates memory through a malloc call and stores the item in the space. Add another record? Same thing, malloc and store. Expiration of items has a free called on the allocated space.



As such after time you end up with "holes" in memory. This is a problem because those holes take up space and may not be able to be used by either Redis or the operating system, depending on the size of the items you are trying to store.

Remember when I wrote about how Redis was using malloc to assign memory? I lied. While Redis did use malloc at some point, these days Redis actually uses jemalloc. The reason for this is that jemalloc, while having lower peak performance has lower memory fragmentation helping to solve the framented memory issues that Redis experiences.

However memory fragmentation still occurs, and so Redis also has a memory defragmentor called defrag 2 which runs in the background cleaning up memory and closing the memory holes. Those old enough to remember running defrag in DOS will understand what is going on here, but in a nutshell it moves items in memory around to try and create a continuous block of used memory. This allows for efficient adds since they can be appended at the end, and allow Redis to release memory back to the operating system.

You can observe the fragmentation in Redis yourself. The below is taken from a local development machine. The relevant lines 21 and 22 are highlighted.

```
used_memory_rss:7688884224

used_memory_rss_human:7.16G

used_memory_peak:8413161288

used_memory_peak_human:7.84G

used_memory_peak_perc:99.69%

used_memory_overhead:38015880

used_memory_startup:782504

used_memory_dataset:8348759296

used_memory_dataset_perc:99.56%

total_system_memory:33536720896

total_system_memory_human:31.23G

used_memory_lua_human:37.00K

maxmemory_policy:allkeys-lfu

mem_fragmentation_ratio:0.92

mem_allocator:jemalloc-3.6.0

lazyfree_pending_objects:0
```

The interesting part in this case is mem_fragmentation_ratio:0.92, where a ratio greater than 1 indicates fragmentation is occuring. You can also see jemalloc is indeed being used mem_allocator:jemalloc-3.6.0.

Memory Organisation Meaning

So knowing how the difference between Redis and memcached in-memory usage, lets see what this means. Memcached slabs once assigned never change their size. This means it is possible to poison your memcached cluster and really waste memory. If you load your empty memcached cluster with lots of 1 MB items, then all of the slabs will be allocated to that size. Adding a 80 KB item once this happens will end up with your 80 KB item sitting in a 1 MB chunk wasting the majority of it's memory. Slabs can be reassigned if they are emptied, so your milage may vary depending on your TTL values and how you access items.

Memcached is also not able release memory to the OS it's running on. Redis can, if you purge all keys, or if enough items expire it will start releasing memory that the OS can use again. The benefits of this depend on if you are running multi-tennant which is less likely in 2021.

Memcached will never use more memory than allocated, while Redis potentially can, especially if you abuse the TTL values (to be covered in expiration below). Memcached will never fragment memory, while Redis will, although it's mitigated to an extent. It might be possible through some odd usage pattern to have highly fragmented memory, causing expiration of keys that might otherwise have been kept.

Finally memcached is limited to 1 MB per item (this is configureable but not advised) while Redis is limited to 512 MB per item stored, but this is also not advised.

Cache Expiration

So lets take a brief tour though some of the more well known cache expiration techniques, Least Frequently Used (LFU) and Least Recently Used (LRU). This is important as both Redis and memcached use cache expiration algorithms in order to know what data needs to be removed from the cache when it is full.

Memcached uses LRU. Redis is actually configurable and so you can use LRU, LFU, random exiction, TTL... you have some options here, but LRU or LFU is what most people tend to use in my experience.

The problem with LRU as an algorithm can be explained fairly simply. Consider a service that checks the cache, and if the value is not there hits the backend to fetch the value. With a cache size of 3 what happens when a stream of requests are made?

A, B, C, A, A, A, A, A, A, A, A, A, A, A, B, C, D

The above represents our key gets in order as they come in, starting with A and followed by B, then C and so forth.

Lets see what happens to the cache when this stream is requested. Starting with an empty cache.

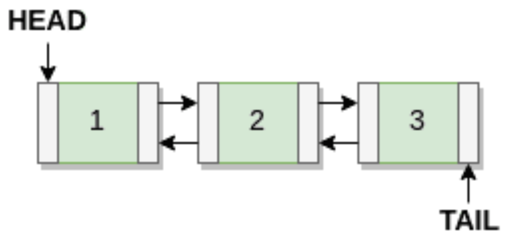
We start with an empty cache, then populate A then B and C, before moving A to the head as there was a stream of them which bumped its access to the head of the cache.

We then finish the stream of A's, and start to pull back B, C and D. The cache looks like the following, after the stream of requests is finished.

Notice that despite A being the most requested item, we evicted it once the final D value was requested. Which was probably not the optimal item to expire as we know it was accessed a lot previously. LFU is designed to deal with this, by keeping a count of how often the item was requested and using that to influence what is expired. Since A was requested so many times it would not be evicted, at least until its usage count is reduced enough though a decay process.

How Memcached expires items

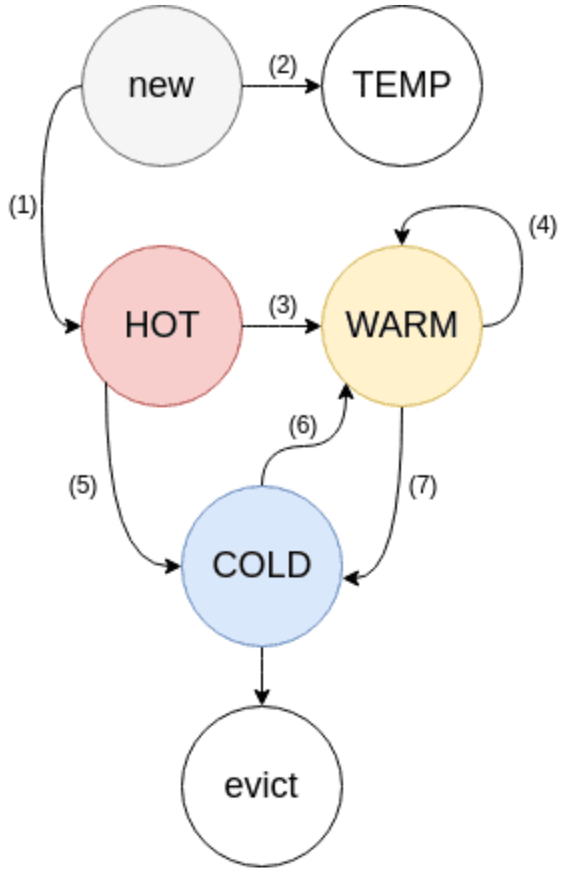
As mentioned memcached uses LRU, and used to implement a perfect LRU implementation, meaning the one you would implement in your COM-SCI 101 course. The implementation itself is a doubly linked list where items are bumped to the head when accessed, added or updated.



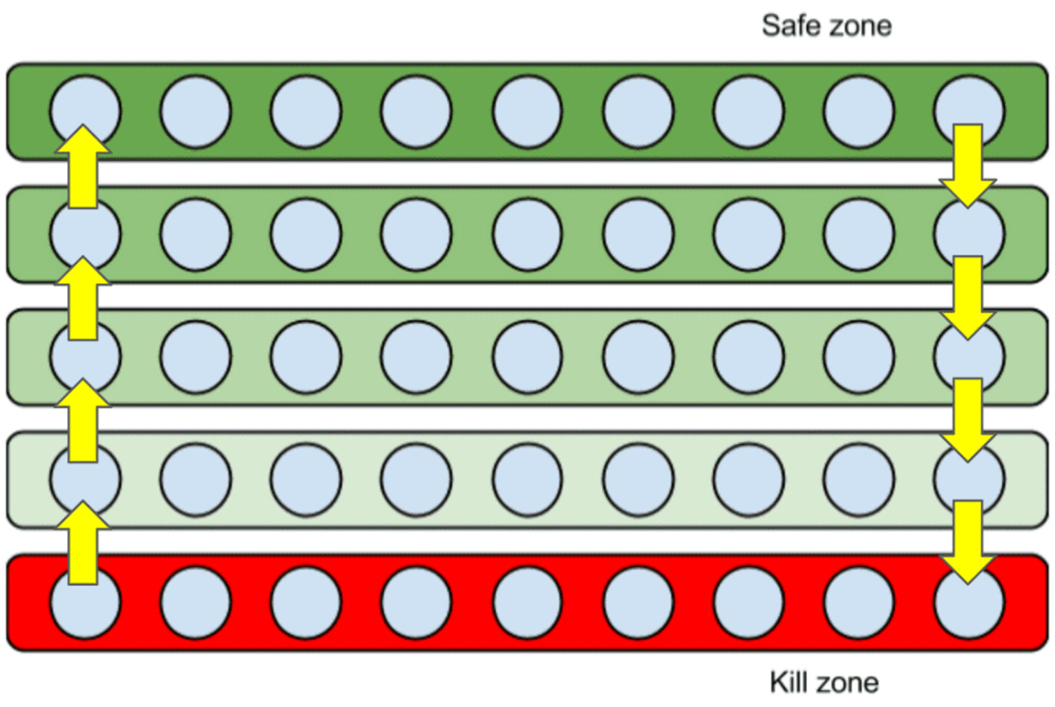
Items are removed when something new is added, or if fetched and its TTL (time to live) age says it should expire. The linked list itself has a mutex lock for any modifications meaning while memcached is multithreaded, there is an upper limit on the number of CPU cores it can use effectively due to mutex contention. One of the first optimisations made in memcached was that items are only bumped every 60 seconds when requested or or updated. This prevents hot (frequently requested or updated) items from causing large amounts of mutex contention.

With this design the upper limit of scale was about 8 worker threads. Which is a bit of a problem when you can buy a 128 thread CPU for your desktop.

Also as we previously discovered LRU isn't an optimal solution to the expiration problem. So in 2020 the memcached developers changed how it works. The below diagram illustrates the change.



Memcached now uses a modern LRU implementation which they describe in detail on their blog. Its actually similar in approach to the Varnish massive storage engine approach which I think has a much clearer diagram, when you add some arrows to demonstate item movement.



Memcached now has multiple doubly linked lists, where some are considered to be hot, making them "safe" as items in those lists will not be expired. When they move to the end of the list they drop to a warm list, and eventually to a cold list. Items in the cold list that reach the end are evicted and potentially killable. Items at the head of the cold or warm list can be bumped to to a warmer list with everything flowing down when this happens.

As such items when used move up and down the cache layers becoming safer or more likely to be evicted. Items that get bursts of activity move up in the cache, becoming safe for a while, and often long enough that should they get burts requests again still be in the cache. Infreqently accessed items drop out.

Its a bit more nuanced than the above due to the TTL (time to live) of items, meaning frequently accessed items with a short TTL can cause the eviction of long TTL items that are infreqently accessed, but this is less likely to happen than under the more traditional LRU implementation.

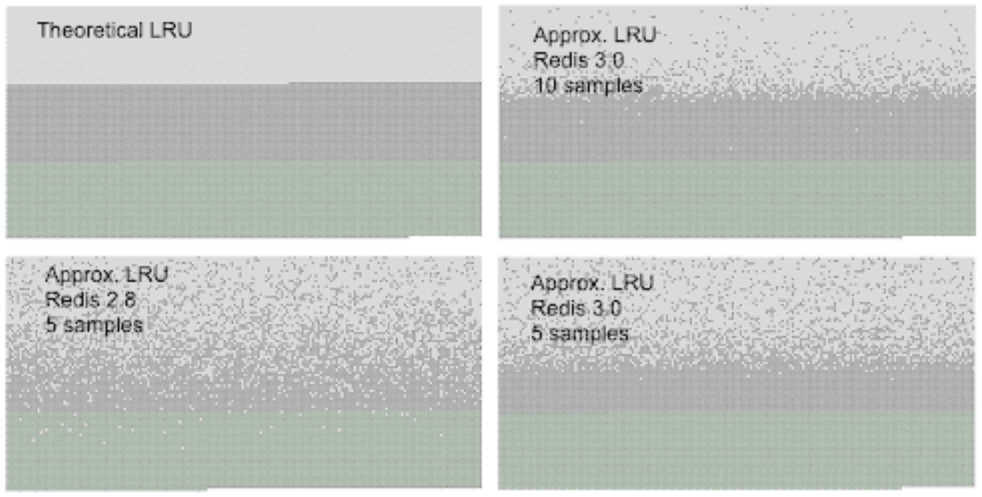
It also means that memcached needs a background thread called the LRU crawler, which is scanning the lists, expiring items where their TTL has ended and readjusting the cache appropiately. Because of the list split it also improves the mutex contetion issue and the memcached developers state it should now scale to 48 threads. It's worth nothing that this thread count estimate a while ago, and it is highly likely to have improved.

How Redis expires items LRU edition

When using Redis in LRU mode the most important thing to note is that it is not your textbook perfect implementation of LRU. Instead Redis implements an approximated LRU algorithm. The reason Redis does this is to save on memory usage.

The way it works is that Redis selects a random number of keys (by default 5) and evicts the oldest. Since Redis 3.0 it improves this algorithm by keeping a pool of good candidates from the previous runs which are mixed into the random selection. While an approximation, by keeping a pool and picking 5 keys it is actually able to achieve close to a theoretical perfect LRU eviction.

The following image from the Redis description about LRU illustrates how well it works.



- The light gray band are objects that were evicted.
- The gray band are objects that were not evicted.

- The green band are objects that were added.

  Don't try to read too much into it. In short you want the images to look like the theoretical image in the top left.

  By increasing the samples to 10 you can see how the graphical approximation gets close to the perfect implementation. You can configure the number of samples by changing the Redis config maxmemory-samples 5 value, where higher values will use more CPU but produce a closer to perfect result.

  For those wondering if picking random values is a good idea or not I suggest you go read the post Caches: LRU v. random on Dan Luu's blog. I actually have a quote about this I like to use.

  Good enough for Dan Luu? Good enough for you.

How Redis expires items LFU

It's actually also approximated. Not as interesting as LRU in my opinion. It just bumps a value when things are accessed and lowers the value over time through a configurable decay time.

The configurable vales you can tweak are,

I am not going to cover Redis LFU in detail here. For generalised or generic situations LRU is generally not as effective as LFU. If you don't know your access patterns pick LFU, at least until you have better information.

So which expiry algorithm is better?

Technically Redis can still expire active elements. The laws of probability suggest that this probably does happen from time to time, but it's also pretty unlikely. Memcached by contrast will only expire unused items over time. Its still LRU but with the main flaw of LRU mitigated. Memcached can expire items with long TTL in favor of keeping more frequently accessed items with a short TTL although in practice this is less a problem with its modern LRU implementation.

In practice both work very well, and their actual performance is not in line with what classical LRU or LFU algorithms suggest due to the tweaks they have added. Simply saying that one uses LFU and is therefore better than one with LRU is just not an argument that should be used to suggest using one tool over another, unless you have some hard data proving so for your specific case.

Memcached Scaling

Memcached is multi-threaded, both for background processes and event processing. This means you can scale up your memcached instance giving it more CPU cores for improved performance. As previously mentioned before 2020 8 cores was the effective limit of scaling for memcached, but this is now at 48 cores due to the new LRU design.

Memcached can scale out into a cluster by design from the moment it was created. Clusters in memcached are interesting beasts, as they can scale out as much as you like, although you may end up having to write your own client libary to do so.

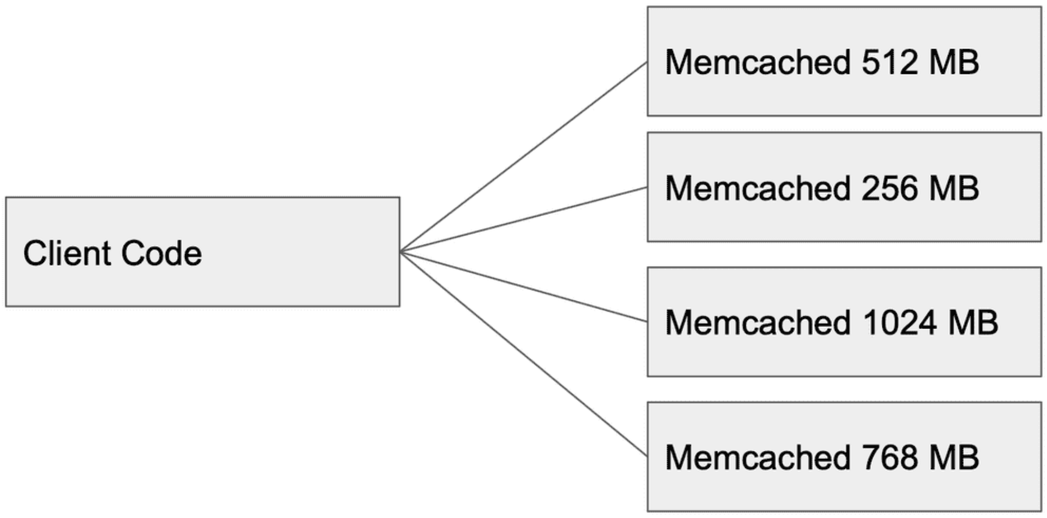
Redis Scaling

Redis is mostly single threaded, with some background tasks such as the defragger running in the background, but for adds/updates /deletes events hit a core event loop. As a result the only effective way to scale Redis is by running multiple instances on the same box, although this impacts memory efficency.

Memcached can however scale out to a cluster. Cluster modes scale to 16,384 nodes, but it is reccomended to stick to less than 1,000.

How Memcached clusters

Memcached clusters use a shared nothing approach. Memcached clusters are in reality just multiple independent memcached services, and they can even have individualised config settings or cache sizes! In order to use the cluster you need to configure the client libaries to be aware of each instance in the cluster, although there are middleware cluster management tools out there which can achieve this for you.



Where this is interesting though is that it means there is no real limit on the number of instances you can have in the cluster. It all comes down to how you distribute items into it, and since that can be in client code you have the ability to do this however you want.

How Memcached distributes keys

However most memcached clients have clustering built in. So how do they do it? How does the client know for a key which instance it should reside on?

So if you ask someone how to achive this the usual first answer is to hash the key, and the apply the modulus operator to the result based on the number of instances you have in your cluster. The result is which instance you should look for or store the key on.

Note you can avoid using the modulus operator with some bit fiddling if that becomes a CPU bottleneck.

In pseudocode it looks like the following.

```
$server_target = inthash($key) % $servercount;
```

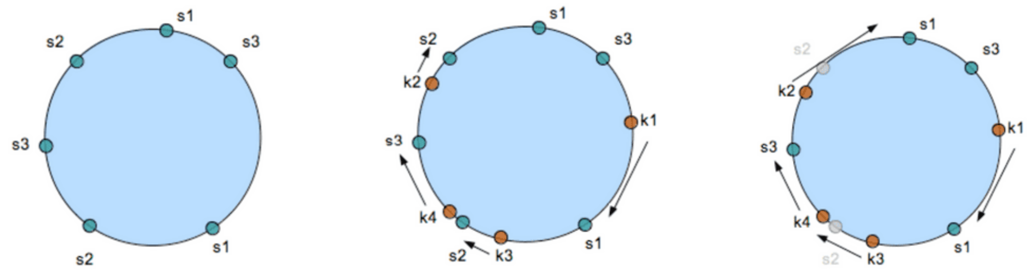
The problem with the above is if you ever change the value of server count, by adding a new server to scale out, or detecting that an instance is unhealthy and remove it from your rotation you end up expiring almost all of the key lookups, as they suddenly produce a new hash value. In effect dumping your whole cache.

So whats the solution? Memcached solves it though consistent hashing, where you don't have to worry if your instance count rises or falls.

Consistant hashing uses a counter like a clock called a continuum. With the continuum mapping to an integer in a range. So assuming your continuum works with integers from 0 to 65,535, numbers over 65,535 would wrap around to 0 before incrementing again. In effect 0 sits at the 12 o'clock position, and 65,535 just to the left of it. 32,767 would be at 6 o'clock and 16,383 at 3 o'clock.

Each instance then has one or more hashes assigned to it, with those hashes mapped to an intger and the server sitting at that position on the continuum. You can add more hashes to each instance to assist with distribution of keys. When a key is added, it has the same hash algorithm applied, producing a number from 0 to 65,535. If that number happens to land on a server space it can be stored. If not, you walk continuum increasing its value until you find a server and store the key there. The same process applies for fetching keys.

If an instance is removed from the pool, then removing it from the continuum means when we walk forward we get the next server in line, which is then populated with the value. The result is that you can add and remove instances from your cluser without expiring all of your keys.

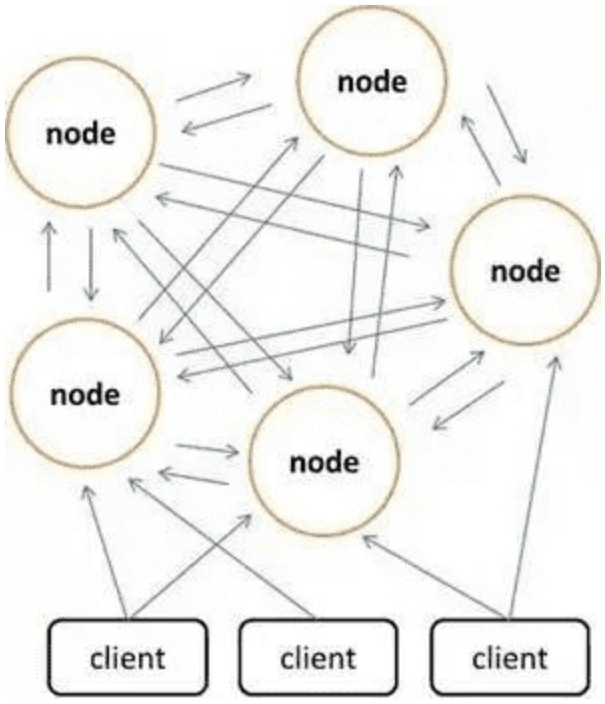


The above visualisation is taken from this blog post on memcached internals, which sadly has fallen off the face of the internet. It shows 3 memcached instances with 2 hashes each in green with key lookups in red, and then what happens when a instance is removed.

What the above means however is you can increase the size of your continuum to as large a number as your programming language supports. 65,535 not enough servers for you? Change it to 4,294,967,295 which ought to be enough memcached servers for anybody.

How Redis clusters

Redis clusters with multiple leader nodes which have some followers. Nodes don't proxy commands sent to them, instead they redirect to the correct node. Clients are expected to remember the mapping so they can hit the node correctly next time. While this is optional from an implementation point of view as the cluster will not enforce this, the client is not considered redis compliant unless it does.



The cluster itself is a full mesh cluster meaning every node is aware of every other node. Because full mesh networks can be come incredibly chatty when communicating with high node counts, it uses a gossip protocol in order for information to spread over the network. So a new node will announce itself to a few other nodes, and they will pass the information along until everyone is aware of the newcomer.

Redis clusters are actually reasonably complicated and worthy of their own blog post. The redis cluser spec page gives a pretty good overview though and is worth looking at if you want more information.

In all seriousness though, use a managed service if you really need a Redis cluster. It will save you a lot of time and effort standing up and maintaining it.

How Redis distributes keys

When items are added to Redis it takes a CRC16 of the key, and then uses 14 bits of the result. This gives 16,384 possible values that Redis can use to store the item into the cluster. This is the reason for its maximum cluster value. This also means there is a hard limit of the number of nodes, although the officially reccomended maximum node count is 1,000 nodes. Each leader handles a portion of the total number of values, which are called hash slots, and the cluster when reconfigring itself knows where each value should be stored.

When reconfiguring because a new node was added or removed then hash slots are moved between leaders, along with their values.

Redis also has special "hash key" tags. These allow you to use different keys to map to the same hash value, so

both hash to the same value. This is a special thing in Redis that memcached does not support, although it is possible if you wrote or modified your client libary. With your own client library you could also have multiple clusters of Redis similar to memcached, by implemeting your own continuum on top.

Redis vs Memcached. Which one should I use?

So given all that has been discussed, for a key value cache which should I use? Redis or Memcached? I am sure anyone reading this is looking for a clear answer one way or the other, and sadly I will dissapoint you. The answer as always is that it depends.

There are a few rules you can probably follow though.

Reasons to use memcached,

- If you need to be able to scale up by throwing more CPU at the problem
- If you are really *hammering* the cache
- If you are caching lots of very small values, ideally all the same size
- If you are running the cluster yourself, memcached is much easier

to setup and maintain

Otherwise I would suggest using Redis and use the other things it brings, such as its lovely data types, streams and such.

However there is an unmentioned secret about memcached and redis. Which is that both are network caches.

So... network caches are actually slow

For some defintion of slow. I have seen people make the following claim from time to time.

"Use Redis/memcached and your application will be fast!"

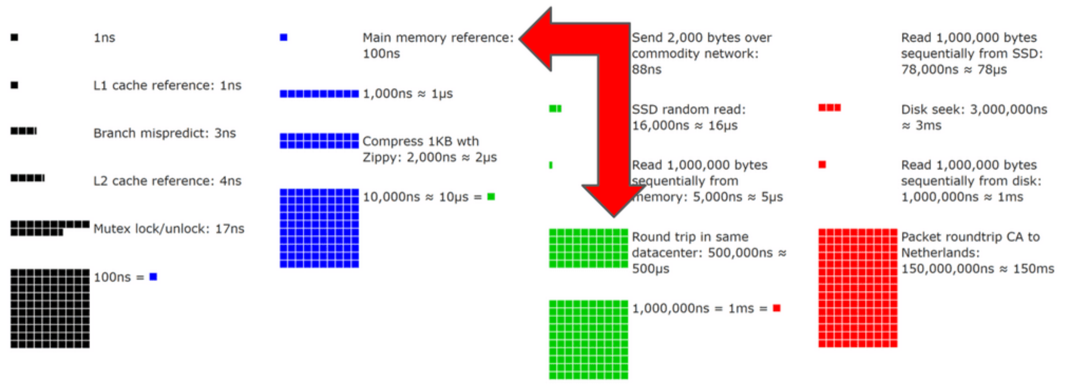
Which may be true but seems to ignore that you still have a network call to make (or socket if using local memcached). In either case both are a lot slower than accessing RAM directly.

So whats a programmer to do?

In my humble opinion Stack overflow got this right by using a L1/L2 caching strategy, which is covered in depth on this Stack Overflow blog.

In a nutshell, if you can you should use memory your application can call directly as a level 1 (L1) cache, and if the value is not there reach out to your level 2 (L2) Redis/memcached cluster cache before going back to your source of truth. If the value was in your L1 cache the difference in access time is 100 ns vs 1 ms which is an order of magnitude faster.

Here is a nice visualisation by Colin Scott which demonstrates the difference.



This approach has the additional benefit of allowing application restarts without hitting your backend data stores any harder than need to be, while keeping things as fast as possible. Lastly, its pretty hard to denial of service an application served from local memory (have you ever tried to overload something served behind varnish?). So consider it a cheap form of protection in what should be a small part of a multi layered defence strategy.

Anyway by now you know more about the internals of memcached and Redis than concievably anyone really should/want to know. Congraulations if you made it this far. If you do however have a slow friday coming up, consider looking though following links, some of which go into even more detail than everything I have written, and formed the basis of a lot of the content above.

- https://www.adayinthelifeof.nl/2011/02/06/memcache-internals/
- https://xuri.me/2017/10/07/memcache-internals.html
- https://nosql.mypopescu.com/post/13506116892/memcached-internals-memory-allocation-eviction
- https://www.igvita.com/2008/04/22/mysql-conf-memcached-internals/
- https://memcached.org/blog/modern-lru/
- https://redis.io/topics/lru-cache
- https://www.reddit.com/r/redis/comments/3tcfuz

/why_lots_of_memory_management_problems_just/

- https://redis.io/topics/ARM
- https://redis.io/topics/lru-cache
- https://memcached.org/blog/modern-lru/
- https://info.varnish-software.com/blog/introducing-varnish-massive-storage-engine
- https://web.archive.org/web/20210328200630/https://www.adayinthelifeof.nl/2011/02/06/memcache-internals/
- https://holmeshe.me/understanding-memcached-source-code-X-consistent-hashing/
- https://www.infoworld.com/article/3063161/why-redis-beats-memcached-for-caching.html
- https://www.imaginarycloud.com/blog/Redis-vs-memcached/
- https://software.intel.com/content/www/us/en/develop/articles/enhancing-the-scalability-of-memcached.html
- https://stackoverflow.com/questions/17759560/what-is-the-difference-between-lru-and-lfu
- https://github.com/memcached/memcached/wiki/ConfiguringServer#threading
- https://redis.io/topics/cluster-spec
- https://alibaba-cloud.medium.com/redis-vs-memcached-in-memory-data-storage-systems-3395279b0941
- https://www.imaginarycloud.com/blog/redis-vs-memcached/
- https://aws.amazon.com/elasticache/redis-vs-memcached/
- https://www.baeldung.com/memcached-vs-redis
- https://stackoverflow.com/questions/10558465/memcached-vs-

redis

- https://www.mikeperham.com/2009/06/22/slabs-pages-chunks-and-memcached/
- https://www.loginradius.com/blog/async/memcach-memory-management/