Using Redis as an LRU cache

When Redis is used as a cache, often it is handy to let it automatically evict old data as you add new one. This behavior is very well known in the community of developers, since it is the default behavior of the popular *memcached* system.

LRU is actually only one of the supported eviction methods. This page covers the more general topic of the Redis `maxmemory` directive that is used in order to limit the memory usage to a fixed amount, and it also covers in depth the LRU algorithm used by Redis, that is actually an approximation of the exact LRU.

Starting with Redis version 4.0, a new LFU (Least Frequently Used) eviction policy was introduced. This is covered in a separated section of this documentation.

Maxmemory configuration directive

The `maxmemory` configuration directive is used in order to configure Redis to use a specified amount of memory for the data set. It is possible to set the configuration directive using the `redis.conf` file, or later using the CONFIG SET command at runtime.

For example in order to configure a memory limit of 100 megabytes, the following directive can be used inside the `redis.conf` file.

```
maxmemory 100mb
```

Setting `maxmemory` to zero results into no memory limits. This is the default behavior for 64 bit systems, while 32 bit systems use an implicit memory limit of 3GB.

When the specified amount of memory is reached, it is possible to select among different behaviors, called **policies**. Redis can just return errors for commands that could result in more memory being used, or it can evict some old data in order to return back to the specified limit every time new data is added.

Eviction policies

The exact behavior Redis follows when the `maxmemory` limit is reached is configured using the `maxmemory-policy` configuration directive.

The following policies are available:

- **noeviction**: return errors when the memory limit was reached and the client is trying to execute commands that could result in more memory to be used (most write commands, but DEL and a few more exceptions).
- **allkeys-lru**: evict keys by trying to remove the less recently used (LRU) keys first, in order to make space for the new data added.
- **volatile-lru**: evict keys by trying to remove the less recently used (LRU) keys first, but only among keys that have an **expire set**, in order to make space for the new data added.
- **allkeys-random**: evict keys randomly in order to make space for the new data added.
- **volatile-random**: evict keys randomly in order to make space for the new data added, but only evict keys with an **expire set**.
- **volatile-ttl**: evict keys with an **expire set**, and try to evict keys with a shorter time to live (TTL) first, in order to make space for the new data added.

The policies **volatile-lru**, **volatile-random** and **volatile-ttl** behave like **noeviction** if there are no keys to evict matching the prerequisites.

To pick the right eviction policy is important depending on the access pattern of your application, however you can reconfigure the policy at runtime while the application is running, and monitor the number of cache misses and hits using the Redis INFO output in order to tune your setup.

In general as a rule of thumb:

- Use the **allkeys-lru** policy when you expect a power-law distribution in the popularity of your requests, that is, you expect that a subset of elements will be accessed far more often than the rest. **This is a good pick if you are unsure**.
- Use the **allkeys-random** if you have a cyclic access where all the keys are scanned continuously, or when you expect the distribution to be uniform (all elements likely accessed with the same probability).
- Use the **volatile-ttl** if you want to be able to provide hints to Redis about what are good candidate for expiration by using different TTL values when you create your cache objects.

The **volatile-lru** and **volatile-random** policies are mainly useful when you want to use a single instance for both caching and to have a set of persistent keys. However it is usually a better idea to run two Redis instances to solve such a problem.

It is also worth to note that setting an expire to a key costs memory, so using a policy like **allkeys-lru** is more memory efficient since there is no need to set an expire for the key to be evicted under memory pressure.

How the eviction process works

It is important to understand that the eviction process works like this:

- A client runs a new command, resulting in more data added.
- Redis checks the memory usage, and if it is greater than the `maxmemory` limit , it evicts keys according to the policy.
- A new command is executed, and so forth.

So we continuously cross the boundaries of the memory limit, by going over it, and then by evicting keys to return back under the limits.

If a command results in a lot of memory being used (like a big set intersection stored into a new key) for some time the memory limit can be surpassed by a noticeable amount.

Approximated LRU algorithm

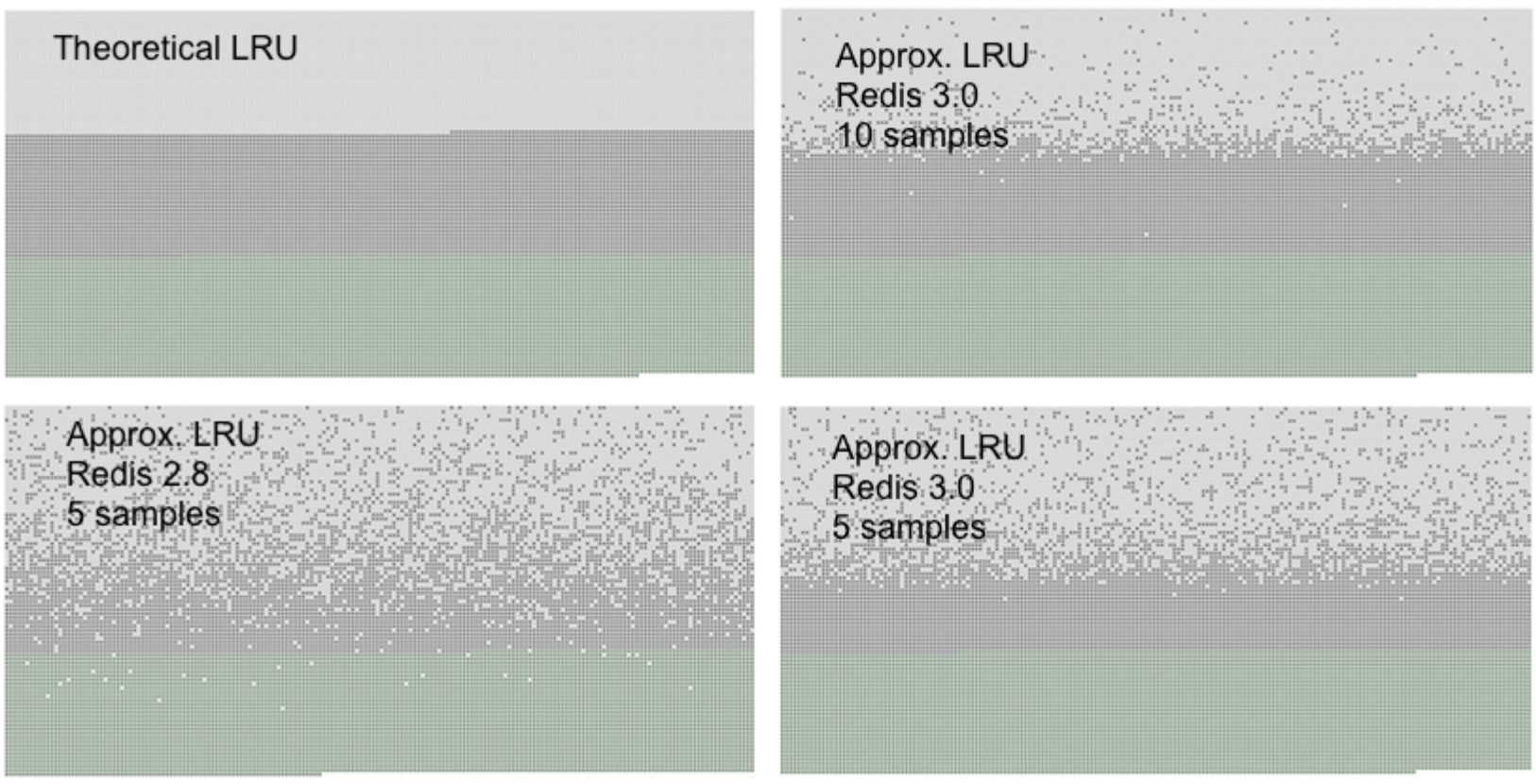
Redis LRU algorithm is not an exact implementation. This means that Redis is not able to pick the *best candidate* for eviction, that is, the access that was accessed the most in the past. Instead it will try to run an approximation of the LRU algorithm, by sampling a small number of keys, and evicting the one that is the best (with the oldest access time) among the sampled keys.

However since Redis 3.0 the algorithm was improved to also take a pool of good candidates for eviction. This improved the performance of the algorithm, making it able to approximate more closely the behavior of a real LRU algorithm.

What is important about the Redis LRU algorithm is that you **are able to tune** the precision of the algorithm by changing the number of samples to check for every eviction. This parameter is controlled by the following configuration directive:

```
maxmemory-samples 5
```

The reason why Redis does not use a true LRU implementation is because it costs more memory. However the approximation is virtually equivalent for the application using Redis. The following is a graphical comparison of how the LRU approximation used by Redis compares with true LRU.



The test to generate the above graphs filled a Redis server with a given number of keys. The keys were accessed from the first to the last, so that the first keys are the best candidates for eviction using an LRU algorithm. Later more 50% of keys are added, in order to force half of the old keys to be evicted.

You can see three kind of dots in the graphs, forming three distinct bands.

- 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.

In a theoretical LRU implementation we expect that, among the old keys, the first half will be expired. The Redis LRU algorithm will instead only *probabilistically* expire the older keys.

As you can see Redis 3.0 does a better job with 5 samples compared to Redis 2.8, however most objects that are among the latest accessed are still retained by Redis 2.8. Using a sample size of 10 in Redis 3.0 the approximation is very close to the theoretical performance of Redis 3.0.

Note that LRU is just a model to predict how likely a given key will be accessed in the future. Moreover, if your data access pattern closely resembles the power law, most of the accesses will be in the set of keys that the LRU approximated algorithm will be able to handle well.

In simulations we found that using a power law access pattern, the difference between true LRU and Redis approximation were minimal or non-existent.

However you can raise the sample size to 10 at the cost of some additional CPU usage in order to closely approximate true LRU, and check if this makes a difference in your cache misses rate.

To experiment in production with different values for the sample size by using the `CONFIG SET maxmemory-samples <count>` command, is very simple.

The new LFU mode

Starting with Redis 4.0, a new Least Frequently Used eviction mode is available. This mode may work better (provide a better hits/misses ratio) in certain cases, since using LFU Redis will try to track the frequency of access of items, so that the ones used rarely are evicted while the one used often have an higher chance of remaining in memory.

If you think at LRU, an item that was recently accessed but is actually almost never requested, will not get expired, so the risk is to evict a key that has an higher chance to be requested in the future. LFU does not have this problem, and in general should adapt better to different access patterns.

To configure the LFU mode, the following policies are available:

- `volatile-lfu` Evict using approximated LFU among the keys with an expire set.
- `allkeys-lfu` Evict any key using approximated LFU.

LFU is approximated like LRU: it uses a probabilistic counter, called a Morris counter in order to estimate the object access frequency using just a few bits per object, combined with a decay period so that the counter is reduced over time: at some point we no longer want to consider keys as frequently accessed, even if they were in the past, so that the algorithm can adapt to a shift in the access pattern.

Those informations are sampled similarly to what happens for LRU (as explained in the previous section of this documentation) in order to select a candidate for eviction.

However unlike LRU, LFU has certain tunable parameters: for instance, how fast should a frequent item lower in rank if it gets no longer accessed? It is also possible to tune the Morris counters range in order to better adapt the algorithm to specific use cases.

By default Redis 4.0 is configured to:

- Saturate the counter at, around, one million requests.
- Decay the counter every one minute.

Those should be reasonable values and were tested experimental, but the user may want to play with these configuration settings in order to pick optimal values.

Instructions about how to tune these parameters can be found inside the example `redis.conf` file in the source distribution, but briefly, they are:

```
lfu-log-factor 10
lfu-decay-time 1
```

The decay time is the obvious one, it is the amount of minutes a counter should be decayed, when sampled and found to be older than that value. A special value of `0` means: always decay the counter every time is scanned, and is rarely useful.

The counter *logarithm factor* changes how many hits are needed in order to saturate the frequency counter, which is just in the range 0-255. The higher the factor, the more accesses are needed in order to reach the maximum. The lower the factor, the better is the resolution of the counter for low accesses, according to the following table:

```
+--------+------------+------------+------------+------------+------------+
| factor | 100 hits   | 1000 hits  | 100K hits  | 1M hits    | 10M hits   |
+--------+------------+------------+------------+------------+------------+
| 0      | 104        | 255        | 255        | 255        | 255        |
+--------+------------+------------+------------+------------+------------+
| 1      | 18         | 49         | 255        | 255        | 255        |
+--------+------------+------------+------------+------------+------------+
| 10     | 10         | 18         | 142        | 255        | 255        |
+--------+------------+------------+------------+------------+------------+
| 100    | 8          | 11         | 49         | 143        | 255        |
+--------+------------+------------+------------+------------+------------+
```

So basically the factor is a trade off between better distinguishing items with low accesses VS distinguishing items with high accesses. More informations are available in the example `redis.conf` file self documenting comments.

Since LFU is a new feature, we'll appreciate any feedback about how it performs in your use case compared to LRU.