Memory Optimization for Redis

While RedisInsight can monitor the health of your databases, here are some tips to increase memory efficiency in Redis.

To get the best performance out of your databases, make sure you are using the latest version of Redis.

Developer best practices

Avoid dynamic Lua script

Refrain from generating dynamic scripts, which can cause your Lua cache to grow and get out of control. Memory is consumed as we have scripts loaded. The memory consumption is because of the following factors.

1. Memory used by the server.lua_scripts dictionary holding original text
2. memory used internally by Lua to keep the compiled byte-code. So If you have to use dynamic scripting, then just use plain EVAL, as there's no point in loading them first.

Things to keep in mind if using dynamic Lua scripts

1. Remember to track your Lua memory consumption and flush the cache periodically with a SCRIPT FLUSH.
2. Do not hardcode and/or programmatically generate key names in your Lua scripts because it makes them useless in a clustered Redis setup.

Switch to 32-bits

Redis gives you the following statistics for a 64-bit machine.

1. An empty instance uses ~ 3MB of memory.
2. 1 Million small Keys -> String Value pairs use ~ 85MB of memory.
3. 1 Million Keys -> Hash value, representing an object with 5 fields, use ~ 160 MB of memory.

64-bit has more memory available as compared to a 32-bit machine. But if you are sure that your data size does not exceed 3 GB then storing in 32 bits is a good option.

64-bit systems use considerably more memory than 32-bit systems to store the same keys, especially if the keys and values are small. This is because small keys are allocated full 64 bits resulting in the wastage of the unused bits.

Advantages

Switching to 32-bit from 64-bit machine can substantialy reduce the cost of the machine used and can optimize the usage of memory.

Trade offs

For the 32-bit Redis variant, any key name larger than 32 bits requires the key to span to multiple bytes, thereby increasing the memory usage.

When to avoid switching to 32 bit

If your data size is expected to increase more than 3 GB then you should avoid switching.

Reclaim expired keys memory faster

When you set an expiry on a key, redis does not expire it at that instant. Instead, it uses arandomized algorithm to find out keys that should be expired. Since this algorithm is random, there are chances that the keys are not expired. This means that redis consumes memory to hold keys that have already expired. The moment the key is accessed, it is deleted.

If there are only a few keys that have expired and redis hasn't deleted them - it is fine. It's only a problem when a large number of keys haven't expired.

How to detect if memory is not reclaimed after expiry

1. Run the `INFO` command and find the total_memory_used and sum of all the keys for all the databases.
2. Then take a Redis Dump(RDB) and find out the total memory and total number of keys.

Looking at the difference you can clearly point out that lot of memory is still not reclaimed for the keys that have expired.

How to reclaim expired keys memory faster

You can follow one of these three steps to reclaim the memory:

1. Restart your redis-server
2. Increase memorysamples in redis conf. (default is 5, max is 10) so that expired keys are reclaimed faster.
3. You can set up a cron job that runs the scan command after an interval which helps in reclaiming the memory of the expired keys.
4. Alternatively, Increasing the expiry of keys also helps.

Trade offs

If we increase the memorysamples config, it expires the keys faster, but it costs more CPU cycles, which increases latency of commands. Secondly, increasing the expiry of keys helps but that requires significant changes to application logic.

Use better serializer

Redis does not have any specific data type to store the serialized objects, they are stored as byte array in Redis. If we are using regular means of serializing our java,python and PHP objects, they can be of larger size which impacts the memory consumption and latency.

Which serializers to use

Instead of default serializer of your programming language (java serialzed objects, python pickle, PHP serialize etc), switch to a better library. There are various libraries like - Protocol Buffers, MessagePack etc.

MessagePack

MessagePack is an efficient binary serialization format. It lets you exchange data among multiple languages like JSON. But it's faster and smaller. Small integers are encoded into a single byte, and typical short strings require only one extra byte in addition to the strings themselves.

As said by Salvatore Sanfilippo, creater of Redis

```
Redis scripting has support for MessagePack because it is a fast and compact serialization format with a simple to implement specification. I liked it so much that I implemented a MessagePack C extension for Lua just to include it into Redis.
```

Protocol buffers

Protocol buffers, usually referred as Protobuf, is a protocol developed by Google to allow serialization and deserialization of

structured data. Google developed it with the goal to provide a better way, compared to XML, to make systems communicate. So they focused on making it simpler, smaller, faster and more maintainable then XML.

Data modeling recommendations

Combine smaller strings to hashes

Strings data type has an overhead of about 90 bytes on a 64 bit machine. In other words, calling set foo bar uses about 96 bytes, of which 90 bytes is overhead. You should use the String data type only if:

1. The value is at least greater than 100 bytes
2. You are storing encoded data in the string - JSON encoded or Protocol buffer
3. You are using the string data type as an array or a bitset

If you are not doing any of the above, then use **Hashes**.

How to convert strings to hashes

Suppose we have to store the number of comments on the posts of a user, we can have a key names like `user:{userId}:post:{postId}:comments`.

This way we have a key per post for each user. So now if we need to find the total number of comments for whole application we can do

```
Redis::mget("user:{$userId}:post:1", "user:{$userId}:post:2", ...);
```

For converting this to Hash you can do something like this-

```
Redis::hmset("user:{$userId}:comments", "post:1", 20, "post:2", 50);
```

This builds a Redis hash with two fields post:1 and post:2 holding the values 20 and 50.

Advantages

Combining small strings to Hashes reduces the memory used and in return save a cost.

Hashes can be encoded to use memory efficiently, so Redis makers recommend that we use hashes whenever possible since “a few keys use a lot more memory than a single key containing a hash with a few fields”, a key represents a Redis Object holds a lot more information than just its value, on the other hand a hash field only hold the value assigned, thus why it’s much more efficient.

Trade offs

Performance comes with a cost. By converting the strings to hash, we conserve memory because it saves only the string value and no extra information like: `idle time`, `expiration`, `object reference count`, and `encoding` related to it. But if we want the key with the expiration value, we can’t associate it with a hash structure as expiration is not available.

When to avoid combining strings to hashes

The decision depends on the number of strings, if it less than 1 million and the memory consumption is not high, the conversion is not effected much and there is no point in increasing the complexity of code.

But if the strings are more than 1 million and the memory consumption is high then this approach should definitely be followed.

Convert hashtable to ziplist for hashes

Hashes have two types of encoding- HashTable and Ziplist. The decision of storing in which of the data structures in done based on the two configurations Redis provides - `hash-max-ziplist-entries` and `hash-max-ziplist-values`.

By default the redis conf has these settings as:

- hash-max-ziplist-entries = 512
- hash-max-ziplist-values = 64

So if any value for a key exceeds the two configurations it is stored automatically as a Hashtable instead of a Ziplist. It is observed that HashTable consumes almost double the memory as compared to Ziplist so in order to save your memory you can increase the two configurations and convert your hashtables to ziplist.

Why ziplist uses less memory

The ziplist implementation in Redis achieves itssmall memory size by storing only three pieces of data per entry; the first is the length of the previous entry, second is the length of current entry and third is the stored data. Therefore, ziplists consumes less memory.

Trade offs

This brevity comes at a cost because more time is required for changing the size and retrieving the entry. Hence, there is an increase in latency and possibly increase in CPU utilization on your redis server.

> **Note:** Similarly, for sorted sets can also be converted to ziplist, but the only differnce is that zset-max-ziplist-entries is 128 which is less that what is there for hashes.

Switch from set to intset

Sets that contain only integers are extremely efficient memory wise. If your set contains strings, try to use integers by mapping string identifiers to integers.

You can either use enums in your programming language, or you can use a redis hash data structure to map values to integers. Once you switch to integers, Redis uses the IntSet encoding internally.

This encoding is extremely memory efficient. By default, the value of set-max-intset-entries is 512, but you can set this value in redis.conf.

Trade offs

By increasing the value of set-max-intset-entries, latency increases in set operations, and CPU utilization is also increased on your redis server. You can check this by running this command before and after making this change.

```
Run `info commandstats`
```

Use smaller keys

Redis keys can play a devil in increasing the memory consumption for your Redis instances. In general, you should always prefer descriptive keys but if you have a large dataset having millions of keys then these large keys can eat a lot of your money.

How to convert to smaller keys

In a well written application, switching to shorter keys usually involves updating a few constant strings in the application code.

You have to identify, all the big keys in your Redis Instance and shorten it by removing extra characters from it. You can achieve this in two ways:

1. You can identify the big keys in your Redis Instance by using RedisInsight. This gives you details about all the keys and a way to sort your data based on the length of keys.
2. Alternatively, you can run the command `redis-cli --bigkeys`

Advantage of using RedisInsight is that it gives you the big keys from the whole dataset whereas the big keys commands run over a certain set of records and return the big keys from that set, hence it is difficult to identify the big keys from the whole dataset using big keys.

Advantages

Let's take an example: Suppose you have 100,000,000 keys name like

```
my-descriptive-large-keyname (28 characters)
```

Now if you shorten the key name like

```
my-des-lg-kn (12 characters)
```

You save 16 characters by shortening your key i.e. 16 bytes which lets you save **1,000,000,000*16 = 1.6GB of RAM Memory !**

Trade offs

Large Keys were more descriptive then shortened keys, hence when reading through your database you may find the keys less relatable, but the memory and cost savings are much efficient as compared to this pain.

Convert to a list instead of hash

A Redis Hash stores field names and values. If you have thousands of small hash objects with similar field names, the memory used by field names adds up. To prevent this, consider using a list instead of a hash. The field names become indexes into the list.

While this may save memory, you should only use this approach if you have thousands of hashes, and if each of those hashes have similar fields. Compressed Field Names are another way to reduce memory used by field names.

Let's take an example. Suppose you want to set user details in Redis. You do something like this:

```
hmset user:123 id 123 firstname Bob lastname Lee
location CA twitter bob_lee
```

Now, Redis 2.6 stores this internally as a Zip List; you can confirm by running debug object user:123 and look at the encoding field. In this encoding, key value pairs are stored sequentially, so the user object we created above would roughly look like this `["firstname", "Bob", "lastname", "Lee", "location", "CA", "twitter", "bob_lee"]`

Now, if you create a second user, the keys are duplicated. If you have a million users, well, its a big waste repeating the keys again. To get around this, we can borrow a concept from Python - NamedTuples.

How does NamedTuple work

A NamedTuple is simply a read-only list, but with some magic to make that list look like a dictionary. Your application needs to maintain a mapping from field names to indexes,like `"firstname" => 0, "lastname" => 1 and so on.`

Then, you simply create a list instead of a hash, like-`lpush user:123 Bob Lee CA bob_lee`. With the right abstractions in your application, you can save significant memory.

Trade offs

The only tradeoffs are related to code complexity. Redis internally uses the same encoding (ziplist) for small hashes and small lists, so there is no performance impact when you switch to a list. However, if you have more than 512 fields in your hash, this approach is not recommended.

When to avoid converting hash to list

Avoid converting hashes to lists when:

1. When your hash contains fewer than 50,000 field-value pairs.
2. The size of your hash values are not consistent (for instance, when some hashes contain only a few field-value pairs while others contain many).

Shard big hashes to small hashes

If you have a hash with large number of key, value pairs, and if each key, value pair is small enough - break it into smaller hashes to save memory. To shard a HASH table, we need to choose a method of partitioning our data.

Hashes themselves have keys which can be used for partitioning the keys into different shards. The number of shards are determined by the total number of keys we want to store and the shard size. Using this and the hash value we can determine the shard ID in which the key resides.

How sharding happens

- **Numeric Keys** - For Numeric keys, keys are assigned to a shard ID based on their numeric key value (keeping numerically similar keys in the same shard).
- **Non-Numeric Keys** - For Non-Numeric keys, CRC32 checksum is used. CRC32 is used in this case because it returns a simple integer without additional work and is fast to calculate (much faster than MD5 or SHA1 hashes).

Things to keep in mind

You should be consistent about the `total no. of expected elements` and the `shard size` while sharding because these two information are required to keep the number of shards down. Ideally, you should not change the values as this changes the number of shards.

If you were to change any one the values, you should have a process for moving your data from the old datashards to the new data shards (this is generally known as resharding).

Trade offs

The only trade off of converting big hashes to small hashes is that it increase the complexity in your code.

Switch to bloom filter or hyperloglog

Unique items can be difficult to count. Usually this means storing every unique item then recalling this information somehow.

Redis sets support this with a single command; however, storing every unique item you want want to count may use a prohibitive amount of memory. If this is the case, consider using a HyperLogLog instead. A HyperLogLog is a probabilistic data structure for counting unique items in a set. HyperLogLogs trade off perfect accuracy for less memory usage.

Bloom filters help when your set contains a high number of elements and you use the set to determine existence or to eliminate duplicates.

Bloom filters aren't natively supported, but you can find several solutions on top of redis. If you are only using the set to count number of unique elements - like unique ip addresses, unique pages visited by a user etc - then switching to hyperloglog saves significant memory.

Trade offs

Following are the Trade Offs of using HyperLogLog:

1. The results that are achieved from HyperLogLog are not 100% accurate, they have an approximate standard error of 0.81%.
2. Hyperloglog only tells you the unique count. It cannot tell you the elements in the set.

For example, if you want to maintain how many unique ipaddresses made an API call today. HyperLogLog tells you`46966 unique`

`IPs for today.`

But if you want `Show me those 46966 IP Addresses` — it cannot show you. For that, you need to maintain all the IP Addresses in a set

Data compression methods

Compress field names

Redis Hash consists of Fields and their values. Like values, field name also consumes memory, so it is required to keep in mind while assigning field names. If you have a large number of hashes with similar field names, the memory adds up significantly. To reduce memory usage, you can use smaller field names.

What do we mean by compress field names

Referring to the previous example in convert hashes to list, we had a hash having user details.

```
hmset user:123 id 123 firstname Bob lastname Lee
location CA twitter bob_lee
```

In this case- firstname, lastname, location, twitter are all field names which could have been shortened to: `fn, ln, loc, etc.` By doing this, you could have saved some memory that was been used by the field names.

Compress values

Redis and clients are typically IO bound and the IO costs are typically at least 2 orders of magnitude in respect to the rest of the request/reply sequence. Redis by default does not compress any value that is stored in it, hence it becomes important to compress your data before storing in Redis. This helps in reducing the payload which in return gives you higher throughput, lower latency and higher savings in your cost.

How to compress strings

There are several compression algorithms to choose from, each with it's own tradeoffs.

1. Snappy aims for high speed and reasonable compression.
2. LZO compresses fast and decompresses faster.
3. Others such as Gzip are more widely available.

GZIP compression uses more CPU resources than Snappy or LZO, but provides a higher compression ratio. GZip is often a good choice for cold data, which is accessed infrequently. Snappy or LZO are a better choice for hot data, which is accessed frequently.

Compressing strings requires code changes. Some libraries can transparently compress objects, you would only need to configure your library. In other cases, you might have to compress the data manually.

Advantages

Compressing strings can save you anywhere between 30-50% memory. By compressing strings, you also reduce the network bandwidth between your application and redis databases.

Trade offs

Compressing/Decompressing requires your application to do extra work. This tradeoff is usually worth it. If you are concerned about additional CPU load, switch to a faster algorithm like snappy or LZO.

When to avoid compression

Compression should not be followed blindly, there are times when compression does not help you reduce your memory, rather increases your CPU utilization. There are few cases when compression should be avoided:

1. For shorter Strings it's likely a waste of time. Short strings generally don't compress much, so the gain would be too small.
2. When the data isn't well structured, compression should be avoided. JSON and XML are good at compression as they have repeated characters and tags.

Enable compression for list

List is just a link list of arrays, where none of the arrays are compressed. By default, redis does not compress elements inside a list. However, if you use long lists, and mostly access elements from the head and tail only, then you can enable compression.

We have two configurations: List-max-ziplist-size: 8kb(default) List-compression-depth: 0(default)

A configuration change in redis.conf `list-compression-depth=1` helps you achieve compression.

What is compression-depth

Compression depth is the number of list nodes from each end of the list to leave untouched before we start compressing inner nodes.

Example:

1. List-compression-depth=1 compresses every list node except the head and tail of the list.
2. List-compression-depth=2 never compresses the head or head->next or the tail or tail->prev.
3. List-compression-depth=3 starts compression after the head->next->next and before the tail->prev->prev, etc.

Trade offs

For small values (for example 40 bytes per list entry here), compression has minimal performance impact. When using 40 byte values with a max ziplist size of 8k, that's around 200 individual elements per ziplist. You only pay the extra "compression overhead" cost when a new ziplist gets created (in this case, once every 200 inserts).

For larger values (for example 1024 bytes per list entry here), compression does have a noticeable performance impact, but Redis is still operating at over 150,000 operations per second for all good values of ziplist size (-2). When using 1024 byte values with a max ziplist size of 8k, that works out to 7 elements per ziplist. In this case, you pay the extra compression overhead once every seven inserts. That's why the performance is slightly less in the 1024 byte element case.