This page is a work in progress. Currently it is just a list of things you should check if you have problems with memory.

Special encoding of small aggregate data types

Since Redis 2.2 many data types are optimized to use less space up to a certain size. Hashes, Lists, Sets composed of just integers, and Sorted Sets, when smaller than a given number of elements, and up to a maximum element size, are encoded in a very memory efficient way that uses *up to 10 times less memory* (with 5 time less memory used being the average saving).

This is completely transparent from the point of view of the user and API. Since this is a CPU / memory trade off it is possible to tune the maximum number of elements and maximum element size for special encoded types using the following redis.conf directives.

```
hash-max-ziplist-entries 512
hash-max-ziplist-value 64
zset-max-ziplist-entries 128
zset-max-ziplist-value 64
set-max-intset-entries 512
```

If a specially encoded value overflows the configured max size, Redis will automatically convert it into normal encoding. This operation is very fast for small values, but if you change the setting in order to use specially encoded values for much larger aggregate types the suggestion is to run some benchmarks and tests to check the conversion time.

Using 32 bit instances

Redis compiled with 32 bit target uses a lot less memory per key, since pointers are small, but such an instance will be limited to 4 GB of maximum memory usage. To compile Redis as 32 bit binary use *make 32bit*. RDB and AOF files are compatible between 32 bit and 64 bit instances (and between little and big endian of course) so you can switch from 32 to 64 bit, or the contrary, without problems.

Bit and byte level operations

Redis 2.2 introduced new bit and byte level operations: GETRANGE, SETRANGE, GETBIT and SETBIT. Using these commands you can treat the Redis string type as a random access array. For instance if you have an application where users are identified by a unique progressive integer number, you can use a bitmap in order to save information about the

subscription of users in a mailing list, setting the bit for subscribed and clearing it for unsubscribed, or the other way around. With 100 million users this data will take just 12 megabytes of RAM in a Redis instance. You can do the same using GETRANGE and SETRANGE in order to store one byte of information for each user. This is just an example but it is actually possible to model a number of problems in very little space with these new primitives.

Use hashes when possible

Small hashes are encoded in a very small space, so you should try representing your data using hashes every time it is possible. For instance if you have objects representing users in a web application, instead of using different keys for name, surname, email, password, use a single hash with all the required fields.

If you want to know more about this, read the next section.

Using hashes to abstract a very memory efficient plain key-value store on top of Redis

I understand the title of this section is a bit scary, but I'm going to explain in details what this is about.

Basically it is possible to model a plain key-value store using Redis where values can just be just strings, that is not just more memory efficient than Redis plain keys but also much more memory efficient than memcached.

Let's start with some fact: a few keys use a lot more memory than a single key containing a hash with a few fields. How is this possible? We use a trick. In theory in order to guarantee that we perform lookups in constant time (also known as O(1) in big O notation) there is the need to use a data structure with a constant time complexity in the average case, like a hash table.

But many times hashes contain just a few fields. When hashes are small we can instead just encode them in an O(N) data structure, like a linear array with length-prefixed key value pairs. Since we do this only when N is small, the amortized time for HGET and HSET commands is still O(1): the hash will be converted into a real hash table as soon as the number of elements it contains grows too large (you can configure the limit in redis.conf).

This does not only work well from the point of view of time complexity, but also from the point of view of constant times, since a linear array of key value pairs happens to play very well with the CPU cache (it has a better cache locality than a hash table).

However since hash fields and values are not (always) represented as full featured Redis objects, hash fields can't have an associated time to live (expire) like a real key, and can only contain a string. But we are okay with this, this was the intention anyway when the hash data type API was designed (we trust simplicity more than features, so nested data structures are not allowed, as expires of single fields are not allowed).

So hashes are memory efficient. This is very useful when using hashes to represent objects or to model other problems when there are group of related fields. But what about if we have a plain key value business?

Imagine we want to use Redis as a cache for many small objects, that can be JSON encoded objects, small HTML fragments, simple key -> boolean values and so forth. Basically anything is a string -> string map with small keys and values.

Now let's assume the objects we want to cache are numbered, like:

- object:102393
- object:1234
- object:5

This is what we can do. Every time we perform a SET operation to set a new value, we actually split the key into two parts, one part used as a key, and the other part used as the field name for the hash. For instance the object named "object:1234" is actually split into:

- a Key named object:12
- a Field named 34

So we use all the characters but the last two for the key, and the final two characters for the hash field name. To set our key we use the following command:

```
HSET object:12 34 somevalue
```

As you can see every hash will end containing 100 fields, that is an optimal compromise between CPU and memory saved.

There is another very important thing to note, with this schema every hash will have more or less 100 fields regardless of the number of objects we cached. This is since our objects will always end with a number, and not a random string. In some way the final number can be considered as a form of implicit pre-sharding.

What about small numbers? Like object:2? We handle this case using just "object:" as a key name, and the whole number as the hash field name. So object:2 and object:10 will both end inside the key "object:", but one as field name "2" and one as "10".

How much memory do we save this way?

I used the following Ruby program to test how this works:

```
require 'rubygems'
require 'redis'

UseOptimization = true

def hash_get_key_field(key)
    s = key.split(":")
    if s[1].length > 2
        {:key => s[0]+":"+s[1][0..-3], :field => s[1][-2..-1]}
    else
        {:key => s[0]+":", :field => s[1]}
    end
end

def hash_set(r,key,value)
    kf = hash_get_key_field(key)
    r.hset(kf[:key],kf[:field],value)
end

def hash_get(r,key,value)
    kf = hash_get_key_field(key)
    r.hget(kf[:key],kf[:field],value)
end

r = Redis.new
(0..100000).each{|id|
    key = "object:#{id}"
    if UseOptimization
        hash_set(r,key,"val")
    else
        r.set(key,"val")
    end
}
```

This is the result against a 64 bit instance of Redis 2.2:

- UseOptimization set to true: 1.7 MB of used memory
- UseOptimization set to false; 11 MB of used memory

This is an order of magnitude, I think this makes Redis more or less the most memory efficient plain key value store out there.

WARNING: for this to work, make sure that in your redis.conf you have something like this:

```
hash-max-zipmap-entries 256
```

Also remember to set the following field accordingly to the maximum size of your keys and values:

```
hash-max-zipmap-value 1024
```

Every time a hash exceeds the number of elements or element size specified it will be converted into a real hash table, and the memory saving will be lost.

You may ask, why don't you do this implicitly in the normal key space so that I don't have to care? There are two reasons: one is that we tend to make tradeoffs explicit, and this is a clear tradeoff between many things: CPU, memory, max element size. The second is that the top level key space must support a lot of interesting things like expires, LRU data, and so forth so it is not practical to do this in a general way.

But the Redis Way is that the user must understand how things work so that he is able to pick the best compromise, and to understand how the system will behave exactly.

Memory allocation

To store user keys, Redis allocates at most as much memory as the `maxmemory` setting enables (however there are small extra allocations possible).

The exact value can be set in the configuration file or set later via CONFIG SET (see Using memory as an LRU cache for more info). There are a few things that should be noted about how Redis manages memory:

- Redis will not always free up (return) memory to the OS when keys are removed. This is not something special about Redis, but it is how most malloc() implementations work. For example if you fill an instance with 5GB worth of data, and then remove the equivalent of 2GB of data, the Resident Set Size (also known as the RSS, which is the number of memory pages consumed by the process) will probably still be around 5GB, even if Redis will claim that the user memory is around 3GB. This happens because the underlying allocator can't easily release the memory. For example often most of the removed keys were allocated in the same pages as the other keys that still exist.
- The previous point means that you need to provision memory based on your **peak memory usage**. If your workload from time to time requires 10GB, even if most of the times 5GB could do, you need to provision for 10GB.
- However allocators are smart and are able to reuse free chunks of memory, so after you freed 2GB of your 5GB data set, when you start adding more keys again, you'll see

the RSS (Resident Set Size) stay steady and not grow more, as you add up to 2GB of additional keys. The allocator is basically trying to reuse the 2GB of memory previously (logically) freed.

- Because of all this, the fragmentation ratio is not reliable when you had a memory usage that at peak is much larger than the currently used memory. The fragmentation is calculated as the physical memory actually used (the RSS value) divided by the amount of memory currently in use (as the sum of all the allocations performed by Redis). Because the RSS reflects the peak memory, when the (virtually) used memory is low since a lot of keys / values were freed, but the RSS is high, the ratio `RSS / mem_used` will be very high.

If `maxmemory` is not set Redis will keep allocating memory as it finds fit and thus it can (gradually) eat up all your free memory. Therefore it is generally advisable to configure some limit. You may also want to set `maxmemory-policy` to `noeviction` (which is *not* the default value in some older versions of Redis).

It makes Redis return an out of memory error for write commands if and when it reaches the limit - which in turn may result in errors in the application but will not render the whole machine dead because of memory starvation.

Work in progress

Work in progress... more tips will be added soon.
