Reverse Engineering Redis Memory Usage

WORK IN PROGRESS

It is useful to understand how much memory is used by a particular key=value pair. Redis does not expose any way to find the memory used by an object. So, we use indirect methods to arrive at the memory usage.

Redis' dump.rdb file is a serialized version of the entire Redis database. We can parse this dump file and arrive at approximate numbers for each key=value pair.

The dump file contains objects in several encodings. Some of these encodings are "raw" - which means in-memory layout is the same as the on-disk layout. It is trivial to figure out the size of objects in raw encoding. Ziplists, Zipmaps and Intsets follow this approach.

Other objects are encoded differently when they are serialized. So, we use some heuristics to arrive at the in-memory size of these objects. Strings, Linked Lists, Hash Tables and Skip Lists fall in this category.

Pointers

The sizeof a C pointer depends on the architecture. On 32 bit systems, a pointer is 4 bytes. On 64 bit systems, a pointer is 8 bytes. Several calculations below depend on `sizeof_pointer`

Strings

A String in Redis has 3 fields - `length`, `free`, and a character buffer. `length` and `free` are ints, so they use 4 bytes each. The size of the character buffer is `length + free + 1`, because the last character is the null character.
When we parse the dump file, we know the length of the string. But we cannot figure out the free bytes, because that value changes at run time. We ignore the free bytes. It may possible to come up with a heuristic to say 'x% of total bytes are usually free'.

So, our formula is : `sizeof string = length of string + 9`
References

1. http://redis.io/topics/internals-sds
2. https://github.com/antirez/redis/blob/unstable/src/sds.h

Hash Table

A Hashtable is made of 3 structs - `dict`, `dictht` and `dictEntry`

```
typedef struct dict {
    dictType *type;
    void *privdata;
    dictht ht[2];
    int rehashidx; /* rehashing not in progress if rehashidx == -1 */
    int iterators; /* number of iterators currently running */
} dict;
/*Each dict has 2 dictht*/
typedef struct dictht {
    dictEntry **table;
    unsigned long size;
    unsigned long sizemask;
    unsigned long used;
} dictht;
typedef struct dictEntry {
    void *key;
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
    } v;
    struct dictEntry *next;
} dictEntry;
```

Overhead of creating a dictionary is `2 * (3 unsigned longs + 1 pointer) + 2 ints + 2 pointers`, which is equal to `56 + 4*sizeof_pointer`.
Each entry in the dictionary is represented by a `dictEntry`, which is `3*sizeof_pointer`. Additionally, each dictEntry is also stored in the field `table`, which costs an additional pointer. `table` actually is rounded *up* to the nearest power of two.
Putting it together, if we have have a dictionary of `n` elements, the approximate memory usage is 56 + 4*sizeof_pointer* + *next_power(n)sizeof+pointer* + *3n*sizeof_pointer
In addition to this, each key is a string, and its size is not included. The value can be any redis object, and its size is also not included.

Linked Lists

Redis uses double linked lists. The relevant structures are pasted below

```
typedef struct listNode {
    struct listNode *prev;
    struct listNode *next;
    void *value;
} listNode;

typedef struct list {
    listNode *head;
    listNode *tail;
    void *(*dup)(void *ptr);
    void (*free)(void *ptr);
    int (*match)(void *ptr, void *key);
    unsigned long len;
} list;
```

Each Linked List has 5 pointers and 1 unsigned long. So overhead of a linked list is `5*sizeof_pointer + 8` bytes.
Each node in the linked list has 3 pointers. So the overhead of every element in the list is `3*sizeof_pointer` bytes.
References :

1. https://github.com/antirez/redis/blob/unstable/src/adlist.h