Memory optimisation: Custom Set

Introduction

A few months ago I was confronted at work with the following use case: “data analysis of some selected clients”. It seems easy, but not if there are **billions of data** and there can be **10 million of clients** to analyse and it needs to run **as fast as possible**!

In this post I’ll describe the solution I found but most importantly, how I came to this solution.

The problem

I have two files:

- one with billions of data concerning clients (hundreds of Gigabytes to Terabytes)
- another that contains the mobile numbers of the clients I am allowed to analyze. This file contains at max 10 million of numbers.

How can I efficiently filter the data in terms of CPU and memory consumption?

My first idea

I didn’t want to sort the files because the best sorting algorithm is in O(n *Log(n)) time complexity and (again) there are billions of data. This why a thought of using a Set<String> in memory and more precisely a HashSet<String> with the following idea:

- Divide the data file into multi parts and give each part to a process
- For each process, load the Set<String> in memory from the client file
- For each process, read each line of the part file

- If the mobile number of the data is in the Set, analyze it
- else drop it

On paper that rocks because a HashSet has a time cost in O(1) for checking the existence of an element. But here is another problem: what is the memory consumption for storing millions of strings ?

A (French) mobile number looks like 33X XX XX XX XX and I needed to store 10 million of Strings with 11 caracteres. On a file in UTF8 encoding one number only takes 13 bytes (with the end of line caracteres) and 10 millions take less than 125 MegaBytes. But in java a String takes really more space. I found the following formula on a blog, I'm not sure if it's true but from my tests it's more true than the approximetly 13 bytes I was expecting:

Minimum String memory usage (bytes) = 8 * (int) ((((no chars) * 2) + 45) / 8)

I ended up with **1.4 Gigabytes** in memory. Instead of trying to optimize the memory usage of a string I tried another way.

My second idea

A French mobile number looks like 33X XX XX XX XX. Since it always begins with 33 there is need to keep it. So I had to find a way to store 9 digits while being still able to use them time efficiently. And the winner is … an integer:

The integer in java is a 32bit signed int so it can store up to 9 digits (the 10 th digit can only goes up to 2 but I don't need it).

10 million integers take only 38 Mbytes, not bad? But using a HashSet<Integer> also comes at a price: with a default load factor of 0.75, the memory used by a HashSet only is approximately **900 Mbytes**. Not that good anymore but I can live with that. The filtering with this HashSet worked well on my eclipse environment but there was a problem on the production environment (which is also used for testing): it was very slow!

The difference between my environment and the production was : the full program in production runs on a Hadoop cluster with an OpenJDK whereas my

eclipse runs with Oracle's Hotspot. I was suspecting three possibilities:

- the mobile numbers were too skewed and it wasn't good for the standard HashSet
- the garbage collector was overworking because some parts of the heap were full or/and the JVM was badly configured (the only thing I was sure was that the heap space was 3 Gigabytes)
- there was some side effect with hadoop

Since I couldn't login on the production servers and I was just starting working with Hadoop, I tried to have a quick look at the implementation of the HashMap (an HashSet uses an HashMap). But instead of wasting time to resolve the problem, I look another way because I wasn't happy with the memory consumption of this solution anyway.

My last idea

I wanted total control and comprehension of the data structure to use so I created a my own Set. My objectives where:

- using the least possible memory,
- insert() and get() as fast as possible.

I only needed to know if a mobile phone was present or not so a bit could do the trick. I thought of using a bits array where the indexes are the mobile numbers and the values their presence.

For example: array_of_bits[the number i'm looking for] == 0 means the mobile number is not present.

Without more optimizations:

- it would have cost 799 999 999 (95 Megabytes) since the maximum possible number is(33)7 99 99 99 99
- and the time costs of get() and insert() would have been immediat

But java doesn't provide bits, I first used bytes which contains 8 bits. With the following formula one can used a bytes array like a bits array:

array_of_bytes[the number/8 + the mobile%8] == 0 means the mobile number is not present.

Bytes are signed numbers so the last bit is very difficult to handle. Char are the only unsigned “number”, this is why I decided to use chars instead of bytes. The only problem with chars is that whether you use 2 bits or 16 bits it will still costs you 16 bits in memory. Though I only needed a table of 799999999 bits for the set, using chars forced me to use 800000000 bits ((ceiling(799999999/16)*16)) which is not a big deal.
I optimized the space consumption using the following idea:

- I only need to store number from 6 00 00 00 00 to 799 999 999 so I could transform those numbers inside the Set into 0 00 00 00 00 to 1 99 99 99 99.
- ceiling(199999999/16)*16 = 200 000 000 bits

This Set using char only costs **24 Megabytes** and has very fast insert() and get(). Sometimes it is worth reinventing the wheel.

Here is the resulting implementation the Set: