Probabilistic filters for Dash

Motivations

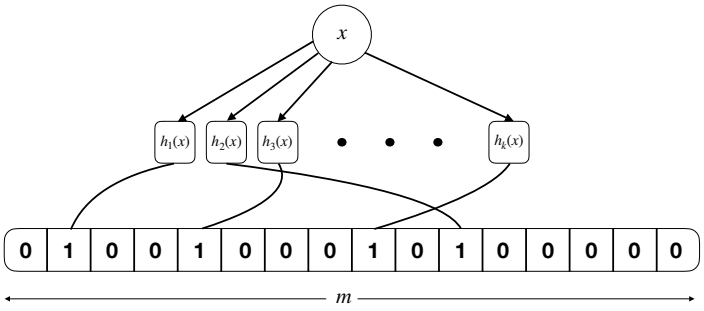
Cryptocurrencies such as Dash or Bitcoin relies on the blockchain technology where a full copy of the ledger is stored on nodes across the network. This ledger is very large and requires important computation power to process. Light clients such as smartphones cannot cope directly with the full ledger but instead request needed information to these nodes. This method is the *Simplified Payment Verification* or *SPV.*

For a node to know which piece of data to send to which client, these light clients send a *probabilistic filter* or *approximate membership set* to this node. This filter, which represent a set of elements potentially interesting for the client such as addresses or transactions hash for instance, is much smaller than a simple list containing these elements. Hence, this data structure reliefs the node memory storage and the network bandwidth which is often a scarce ressource.

During this study, we aimed at improving the existing performances of this filter, either by completely changing the data structure or by harnessing the capabilities of modern CPUs such as vectorised operations. According to our requirements, the improvements relate either to the speed of execution of a query to the filter, or to the size of the filter.

Improvements

The probabilistic filters implemented today in the Dash source code are **Bloom filters**. This type of filter was first introduced in 1970 by Burton Howard Bloom. It consists in a bit array, with all bits initially set to 0. An element is represented in this array by *k* bits set to 1, pseudo-randomly chosen by *k hash functions.* All the elements in the set to be represented will thus turn some bits to 1. Afterward, to check if a given element is a member of the represented set, the same k hash functions are applied to this element and the bits pointed by these k hash value are tested : if all the corresponding bits are set to 1, the element is *Probably inside* the set ; if one or more bits are set to 0, it is *Definitely not inside.* This data structure allows elements that were not inserted in the filter to lead to a positive answer : this is called a False positive.



Query speed

This Bloom filter is a smart and efficient data structure but is far from being optimal. An impressive amount of research has already be done and is still being done to find better alternatives that are either faster to query, that are more compact or that features any special properties. The study was conducted by first exploring these diverse alternatives , understand their mathematical grounds and identify which of those fit the better with our *Simplified Payment Verification* problem. After that, the chosen potentially better alternatives were modified, tuned or implemented from scratch.

Using profiling tools (e.g. Apple Instruments) we have able to determine two main factors that have an impact on the query speed : the time required to compute hash functions and the number of memory accesses performed by the CPU. The first factor has been tackled by replacing the

MurmurHash3 function by the faster Farmhash function. The memory access bottleneck has been addressed using different data structures such as the **Cuckoo filter** and the **Morton filter** or variants of the Bloom filter such as the **SIMD Blocked Bloom filter** and the One-memory-access Bloom filter. Tests were run on these filters with parameters that mimic real-life usage : filters representing 100 to 100,000 elements (addresses, hash of transactions, COutPoints), mostly negative queries, false positive rate of 0.01%. The obtained speedups are summarised in the following table :

Filters	Speed-up (Dash Bloom filter as reference)
Bloom filter using only 2 hash functions	3x - 4x
One-memory-access Bloom filter	3.5x - 4.5x
Morton filter	3.5x - 5x
Cuckoo filter	**6x - 7.5x**
SIMD blocked Bloom filter	**7x - 8x**

The SIMD blocked Bloom filter and the Cuckoo filter are by far the fastest filters tested, yielding query times up to 8 times shorter. As its name suggests, the SIMD blocked Bloom filter relies on SIMD (Single Instruction Multiple Data) set of instructions such as Intel AVX, which are supported by most recent CPUs but can be a paying option in Web Services providers (e.g. Amazon Web Services). This can be an obstacle to portability we can solve either by imposing specifications to the *masternodes* or implementing a slower non-SIMD blocked Bloom filter for CPUs that doesn't support AVX instructions.
The Morton filter is has first been introduced in 2018 so no reference implementation is available and no discussions about its performances were found. In my tests, with my implementation, Morton filter is slower than the two last options, whereas it was expected to be faster than the Cuckoo filter. Further work is needed to verify if we can reach these expected performances.

Space efficiency

If our goal is to minimise the size of the filters sent across the network, the Cuckoo filter and the Morton filter are an efficient alternative, reducing the size of the data structure by bringing it closer to its optimal value. Golomb coded sets is a more compact alternative, but comes at the cost of slow queries, linear in time with the number of elements represented by the set.

Filters	Bits per item
Bloom filter	$\approx 1.44 \times log_2(\frac{1}{\epsilon})$
Cuckoo filter	$\approx \frac{K + log_2(\frac{1}{\epsilon})}{0.95}$ $\quad K = 2 \text{ or } 3$
Morton filter	$\approx K' + \frac{log_2(\frac{1}{\epsilon})}{0.95}$ $\quad K' > 1.9$
Golomb coded sets	$\approx 1.5 + log_2(\frac{1}{\epsilon})$

ϵ = *false positive rate*

Conclusion

A variety of filters and variants exists to either improve speed or memory usage. I found out that the current fastest of these filters are the **Cuckoo filter** and the **SIMD blocked Bloom filter**. To address the memory usage issue, Golomb coded sets are near-optimal data structures but can not be used in situations where a server has to handle thousands of filters at once, each representing an important number of elements (e.g. a notification server), this is why a Cuckoo filter would be prefered.

Further work should be done to take full advantage of the **Morton filter** which is expected to be a better candidate for both query speed and size.

Another interesting line of research to ease the work of server handling the filters is the use of **Multidimensional Bloom filters**, or **Bloofi i**ntroduced by Crainiceanu and Lemire that allows multiple queries at once.