Filters Analyse for Dashpay

Yang Hua yang.hua@insa-lyon.fr

1.Abstract

Lightweight dash client users are playing an important role nowadays. The lightweight Dash users or the SPV client users rely on one or more masternodes to check the transactions which are related to them.The SPV client are widely implemented in the constrained devices such as smartphones. Also, the SPV clients all need a specific probabilistic data structure which is highly space-efficient. For now, a Bloom filter is implemented. The Bloom filters not only reduce the space which each element takes for storage but also the time for querying each element in the server. However, a more space and time efficient filter or data structure is always required, especially after scaling the Dash network and when we need a fast, without delay notification with a notification server.

2.Background

In what follows, we briefly introduce the mechanism of the probabilistic filters from the very basic and currently used Bloom filter to the Cuckoo filter who is proved to be "*practically better than Bloom filter*"[1] and the mechanism of SIMD calculating for improving the performance of the data structure.

Bloom Filter: Bloom filter is invented by Bloom in 1970s. Please refer to [2]for further information about Bloom filters. Bloom filters are implemented in cryptocurrency for its time and space efficiency and the false positive rate for protecting the privacy of clients.

For the application of a Bloom filter, the user only need to type in two parameters: the **number of the elements at maximum** which will be hashed into the Bloom filter and **the false positive rate** expected.

However, for the best performance(It mostly refer to the space efficiency of the Bloom filter), the parameters of the Bloom filter are functions of the false positive and the number of the elements inserted in the Bloom filter.

In our case, the false positive rate will be around 5%, and the number of the elements in a Bloom filter will be a random number between 100 and 500.

Below , we will present how to set up the Bloom filter in function of the rate of the false positive and the elements inserted in the Bloom filter in our case.

p: the false positive rate n: number of the elements inserted in the Bloom filter m: the size of the Bloom filter k: the number of the hash functions needed p_q: the fraction of queries on existing items
Table 1: Notations used throughout the paper.

First, we calculate m the size of the Bloom filter in function of n and p,

$$P = 2^{-ln2\frac{m}{n}} \Rightarrow \ln p = \ln 2 \cdot (-\ln 2) \cdot \frac{m}{n} \Rightarrow m = -\frac{n \cdot lnp}{(ln2)^2}$$

Then, we calculate the number of the hash function needed:
we can prove that when m,n are all fixed and when we have

$$k = ln2 \cdot \frac{m}{n} = 0.7 \cdot \frac{m}{n}$$

the false positive rate is the lowest, the lowest false positive rate is:

$$P(error) = \left(1 - \frac{1}{2}\right)^k = 2^{-k} = 2^{-ln2\frac{m}{n}} \approx 0.6185^{\frac{m}{n}}$$

we also have:

$$P(error) = 2^{-k} \Rightarrow \log_2 P = -k \Rightarrow k = \log_2 \frac{1}{P} \Rightarrow \ln 2 \frac{m}{n} = \log_2 \frac{1}{P}$$

that means:

$$\frac{m}{n} = \frac{\log_2 \frac{1}{p}}{\ln 2}$$

so when the false positive rate is 5%, we will have

$$\frac{m}{n} \approx 6.2$$

that means if p = 5%, for every element we need 6 bits to store it.

For example, if we have a Bloom filter which will be inserted in 300 elements, then we will need at least 1870 bits and the number of the hash function will be 4.

we have tested differently implemented Bloom filters: the Bloom filter currently used in dashpay[3], the Bloom filter implemented in cpp[4].

Cuckoo Filter:
The Cuckoo Filter is a probabilistic data structure that supports fast set membership testing. It is very similar to a Bloom filter in that they both are very fast and space efficient.

Cuckoo filters are a new data structure, described in a paper in 2014 by Fan, Andersen, Kaminsky, and Mitzenmacher. Cuckoo filters improve upon the design of the Bloom filter by offering deletion, limited counting, and a bounded false positive probability, while still maintaining a similar space complexity. They use Cuckoo hashing[5] *to resolve collisions and are essentially a compact Cuckoo hash table.*[6]

The Cuckoo filter's biggest advantage against Bloom filter is that the element in the Cuckoo filter can be deleted in a complexity of O(1). In a standard Bloom filter, elements can't be deleted, for the counting Bloom filters, elements can be deleted while it's in a complexity of O(k).

Cuckoo filter also has stronger lookup performance because for one lookup process, the max 2 buckets are checked. [7]

3.Model

In this section, i will present my testing environment and the test model.

We put up a local nodejs server who will process hundreds of thousand of addresses at the same time.

The testing machine has a 4.2 GHz intel Core i7, and a 16GB 2400 Mhz ddr4 memory and avx2 capable.

We assume that each Bloom filter has 300 elements inserted in average, at least 100 elements and 500 elements at most. The sizes of the Bloom filters will be fixed during the test.

For a notification server, the most important performance is the lookup performance. In addition to it, the difference between space efficiency of differents filters(Cuckoo filter, Bloom filter, blocked Bloom filter, etc) can be ignored: according to the paper ***Cuckoo Filter: Practically Better Than Bloom ,*** the number of bits per item for storage is 12.6 for Cuckoo filter and 13 for Bloom filter and 13 for blocked Bloom filter. That means even we have 100k elements, the difference of the storage will only be 5kB which means nothing to one server in 2018 and even less compared with the performance of lookup.

The workload of the filters are characterized by the positive queries and the negative queries. The positive query refers to one query in which the element searched is in the filter and the negative query refers to one query in which the element searched is not in the filter. In our notification server, one Bloom filter has an average of 300 elements inserted and the total workload is varied from 100k to 500k. So the fraction of queries on existing items will vary from 0.3% to 0.06% in average which is quite close to zero.

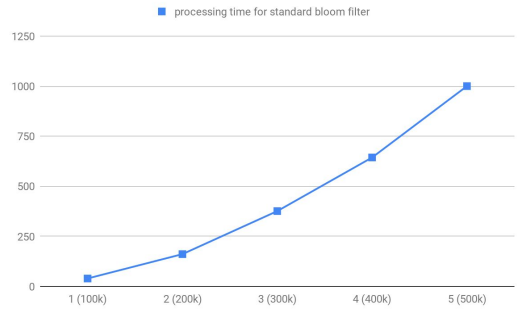
For accelerating each filters, we have tried to optimize the filters with different strategies, such as implementing a data structure in the server to avoid duplicated calculating.

For the implementation of the c++ native modules, unfortunately we chose the NAN to bind the c++ add-on to nodejs instead of N-api, which is believed to be more stable and easy to implement and it has no need for recompilation before using. However it is hard to tell the difference of the performance between NAN and N-api, the further improvement is only based on the maintenance of the NAN and N-api between versions.

4.lookup performance

Bloom filter

I firstly tested one Bloom filter implemented in javascript[8] in the server which is a model which is suitable for usage in Bitcoin Connection Bloom Filtering as part of BIP37 [9], using murmurhash3 and without any data structure for accelerating the looking up time.
the test results show as below:

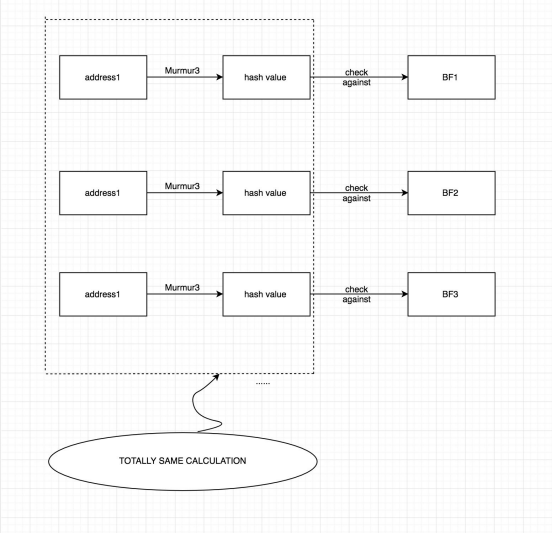


It is obvious that the function between number of the addresses and the time for looking up is not linear (in fact after curve fitting with the limited points, it turns out to be a fourth degree polynomial). So one solution to accelerate the processing time is necessary.

5. Workaround

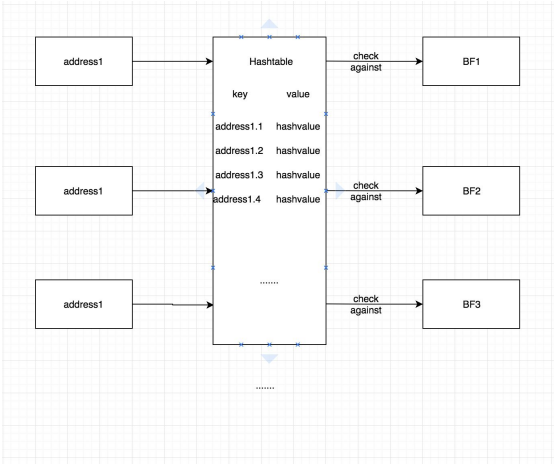
5.1 naive solution towards the lookup performance in Bloom filter

For every element, we need to hash the element through hash functions and check the index in all Bloom filters



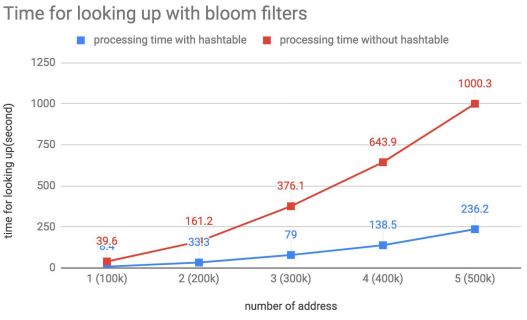
To avoid these duplicated calculations, i set up one hashtable to store every hash value with the address appended with the number of the hash function(in practical we have 4 hash functions in total so the number will be a number from 1 to 4)

after that , the process will be simplified as follows:



One of the most common workaround for reducing the processing time is to set up the data structure inside the server, although it will take more space for storing the data. We tried with one hash table with the search complexity O(n) and repeated the test.

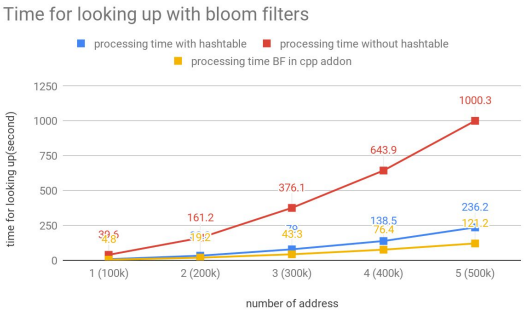
The results are showed in the chart below.



However, this solution has the following disadvantages:

1. the hashtable will cost more space. When the number of the elements in the server grows, the server will have to scale the hashtable and it may face the worst situation: rehashing all of the elements already in the hash table which will certainly cost more time.
2. the hash table will only accelerate the processing time for those who have already been inserted into the Bloom filters. For those first-hashed elements, the processing time will be all the same.In our test situation, the performance is increased greatly because all the elements would be queried several times(the number of times equals to the number of filters in total.)

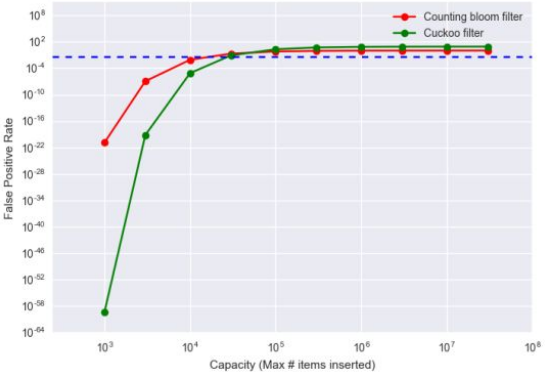
And then we managed to optimize the server from the very bottom. As we know, the nodejs is running on v8 engine who is implemented in c++, that's why we can use many c++ libraries while programming in nodejs. Also, using c++ add-on avoid many usage of v8 APIs, as a result, the processing time turns out to be much less if we use a c++ addon.
the results for processing with c++ add-on shows below.



5.2 Better solutions

Cuckoo Filter

In the paper **Cuckoo Filter: Practically Better Than Bloom,** the compared the lookup performance with different filters (e.g simd-blocked Bloom filters Cuckoo filters and standard Bloom filters etc.) with different fraction of queries on existing items. And in the case when we have the fraction close to 0, we see that blocked Bloom filters perform much better than other filters and the Cuckoo filters too.
However, the false negative is very low compared to the Bloom filters when the elements inside the filter(less than 1%)[10],



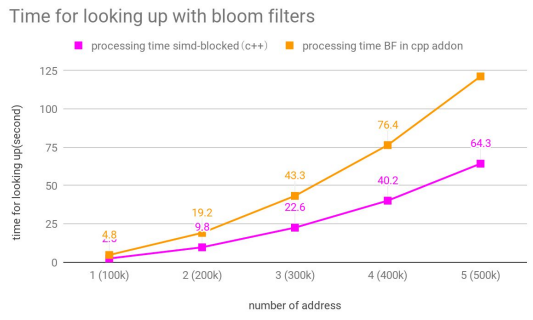
In addition to that, the false positive rate of Cuckoo filter rests between 0 and 5% no matter how many elements are being inserted. It seems to be a good news for common usage of probabilistic filters, while it may cause some problems in its

usage in cryptocurrency. The current used Bloom filters permit the SPV users to increase the the false positive rate to protect their privacy. But for the Cuckoo filters, the limited false positive rate means limited privacy especially the new SPV users, who has Bloom filters with little capacities. Their false positive rates tend to be 0 which means 0 privacy.Even it is proved that the current using Blooming filter may face severe privacy leakage[11], the 0 false positive rate is not welcomed.

Simd-blocked Bloom filter

SIMD (Single instruction, multiple data) [12]instructions bring us the data level parallelism, but not concurrency, which is perfectly suitable for the notification server for Dash: we have tons of addresses coming at the same time and they are all hashed and checked against the filters simultaneously. With the SIMD blocked Bloom filter, a number of the addresses are loaded at the same time, not one after another. Besides, one instruction is operated on a number of data. The application fo SIMD instructions is no doubt a great improvement for Bloom filters.

we tested the simd-blocked Bloom filter to compare with the other Bloom filters.(To make it clear in the chart, we just present it with only the fastest one among the ones tested: standard Bloom filter in cpp addon)



6. Conclusion

Certainly, the current used Bloom filter can be improved in several aspects. Cuckoo filter has a great advantage compared with Bloom filters in lookup performance, but it may not be the best choice considering the need of the SPV users for protecting their privacy.

Fortunately, SIMD-blocked Bloom filter performs very well dealing with large number of data in a notification server in our test situation(fraction of the existing elements close to 0) which is quite similar to the real world situation.

Between SIMD-block Bloom filter, standard Bloom filter and Cuckoo filter, the best workaround for the currently used dash is SIMD-block Bloom filter, which can be improved by the actual used Bloom filter.

7. References

[1]B. Fan, D. G. Andersen, M. Kaminsky† , M. D. Mitzenmacher.
Cuckoo Filter: Practically Better Than Bloom

[2]Bloom filter:
https://en.wikipedia.org/wiki/Bloom_filter

[3]dashpay
https://github.com/dashpay/dash

[4]Bloom-filter-cpp
https://github.com/bbondy/Bloom-filter-cpp

[5]Cuckoo hashing:

https://en.wikipedia.org/wiki/Cuckoo_hashing

[6]Cuckoo filter:
https://brilliant.org/wiki/Cuckoo-filter

[7]Cuckoo filter vs Bloom filter
https://bdupras.github.io/filter-tutorial/

[8]Bloom filter in javascript
https://www.npmjs.com/package/Bloom-filter

[9]BIP37
https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki

[10]Julius. *Probabilistic Data Structure Showdown: Cuckoo Filters vs. Bloom Filters*
https://blog.fastforwardlabs.com/2016/11/23/probabilistic-data-structure-showdown-Cuckoo.html

[11]A. Gervais , G. O. Karame , D.Gruber , S. Capkun
On the Privacy Provisions of Bloom Filters in Lightweight Bitcoin Clients

[12]Tuomas Tonteri. *A practical guide to SSE SIMD with C++*
http://sci.tuomastonteri.fi/programming/sse