Fork me on GitHub

简体中文

Bloom Filters by Example

A Bloom filter is a data structure designed to tell you, rapidly and memory-efficiently, whether an element is present in a set.

The price paid for this efficiency is that a Bloom filter is a **probabilistic data structure**: it tells us that the element either *definitely is not* in the set or *may be* in the set.

The base data structure of a Bloom filter is a **Bit Vector**. Here's a small one we'll use to demonstrate:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Each empty cell in that table represents a bit, and the number below it its index. To add an element to the Bloom filter, we simply hash it a few times and set the bits in the bit vector at the index of those hashes to 1.

It's easier to see what that means than explain it, so enter some strings and see how the bit vector changes. Fnv and Murmur are two simple hash functions:

Enter a string:
add to bloom filter

fnv:
murmur:

Your set: []

When you add a string, you can see that the bits at the index given by the hashes are set to 1. I've used the color green to show the newly added ones, but any colored cell is simply a 1.

To test for membership, you simply hash the string with the same hash functions, then see if those values are set in the bit vector. If they aren't, you know that the element isn't in the set. If they are, you only know that it *might* be, because another element or some combination of other elements could have set the same bits. Again, let's demonstrate:

Test an element for membership:

fnv:
murmur:

Is the element in the set? no

Probability of a false positive: 0%

And that's the basics of a bloom filter!

Advanced Topics

Before I write a bit more about Bloom filters, a disclaimer: I've never used them in production. Don't take my word for it. All I intend to do is give you general ideas and pointers to where you can find out more.

In the following text, we will refer to a Bloom filter with *k* hashes, *m* bits in the filter, and *n* elements that have been inserted.

Hash Functions

The hash functions used in a Bloom filter should be **independent** and **uniformly distributed**. They should also be as fast as possible (cryptographic hashes such as sha1, though widely used therefore are not very good choices).

Examples of fast, simple hashes that are independent enough[3] include murmur, the fnv series of hashes, and HashMix.

To see the difference that a faster-than-cryptographic hash function can make, check out this story of a ~800% speedup when switching a bloom filter implementation from md5 to murmur.

In a short survey of bloom filter implementations:

- Chromium uses HashMix. (also, here's a short description of how they use bloom filters)
- python-bloomfilter uses cryptographic hashes
- Plan9 uses a simple hash as proposed in Mitzenmacher 2005
- Sdroege Bloom filter uses fnv1a (included just because I wanted to show one that uses fnv.)
- Squid uses MD5

How big should I make my Bloom filter?

It's a nice property of Bloom filters that you can modify the false positive rate of your filter. A larger filter will have less false positives, and a smaller one more.

Your false positive rate will be approximately $(1-e^{-kn/m})^k$, so you can just plug the number *n* of elements you expect to insert, and try various values of *k* and *m* to configure your filter

for your application.[2]

This leads to an obvious question:

How many hash functions should I use?

The more hash functions you have, the slower your bloom filter, and the quicker it fills up. If you have too few, however, you may suffer too many false positives.

Since you have to pick *k* when you create the filter, you'll have to ballpark what range you expect *n* to be in. Once you have that, you still have to choose a potential *m* (the number of bits) and *k* (the number of hash functions).

It seems a difficult optimization problem, but fortunately, given an *m* and an *n*, we have a function to choose the optimal value of *k*: *(m/n)ln(2)* [2], [3]

So, to choose the size of a bloom filter, we:

1. Choose a ballpark value for *n*
2. Choose a value for *m*
3. Calculate the optimal value of *k*
4. Calculate the error rate for our chosen values of *n*, *m*, and *k*. If it's unacceptable, return to step 2 and change m; otherwise we're done.

How fast and space efficient is a Bloom filter?

Given a Bloom filter with *m* bits and *k* hashing functions, both insertion and membership testing are *O(k)*. That is, each time you want to add an element to the set or check set membership, you just need to run the element through the *k* hash functions and add it to the set or check those bits.

The space advantages are more difficult to sum up; again it depends on the error rate you're willing to tolerate. It also depends on the potential range of the elements to be inserted; if it is very limited, a deterministic bit vector can do better. If you can't even ballpark estimate the number of elements to be inserted, you may be better off with a hash table or a scalable Bloom filter[4].

What can I use them for?

I'll link you to wiki instead of copying what they say. C. Titus Brown also has an excellent talk on an application of Bloom filters to bioinformatics.

References

1: Network Applications of Bloom Filters: A Survey, Broder and Mitzenmacher. An excellent overview.

2: Wikipedia, which has an excellent and comprehensive page on Bloom filters

3: Less Hashing, Same Performance, Kirsch and Mitzenmacher

4: Scalable Bloom Filters, Almeida et al