Bloom filters

Posted on September 4, 2013, 5:08 am, by Alexander Sandler, under Blog.

Bloom filter is a data structure that contains set of elements. Unlike regular data structures it cannot contain data that is associated with certain key. Neither it can contain keys themselves. The only type of information it can contain is whether certain key belongs to a set or not.

You must be wondering what it is useful for. Here's typical scenario for using bloom filter. Lets say you have large data structure and you often have to check if particular member is in the data structure. For example, lets say you have large binary tree and you often query the tree if it contains some element.

Since actual data is in large binary tree, checking if some element in the tree takes logarithmic time. For a tree that contains one million objects, that would be 20 levels tall tree. To optimize access time, you may want to check if certain entry is in the data structure before you search the tree. Lets say when adding a new entry, you may want to check if that entry already in the tree. When deleting, you may also verify that element you're trying to delete is there.

> I came across bloom filters through disk based data structures. Accessing some data structure in logarithmic time may be good enough even for a very large data structures. But not when data structure is on disk. For just any large data structure that lies on disk, I/O operations is a scarce resource. So, reducing number I/O operation can be a game changer.

Representing large data structures is where bloom filters shines. When you insert an element into your data structure, you first insert it into bloom filter. Bloom filter resides in memory and so its easy to check before you check your main data structure. This is why it is called filter – it actually filters out unnecessary look-ups on main data structure.

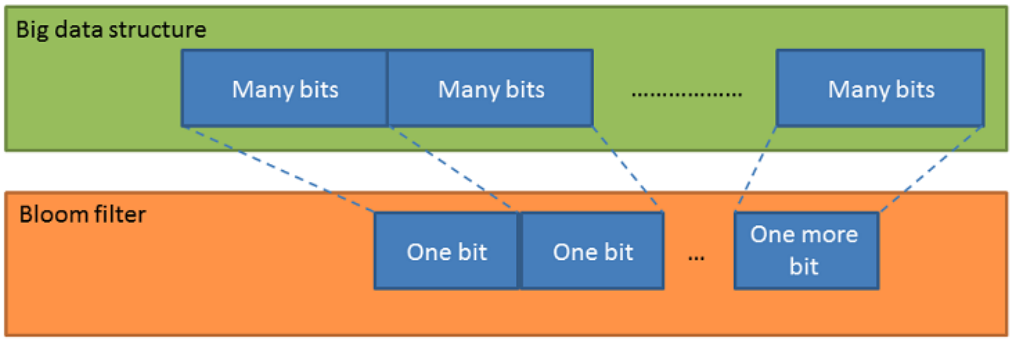
Another interesting property of the bloom filter is that it often consumes less space per element than element's key. In fact it can perform fairly well with just 10 bits per element it stores (more on this later). However, this does not come for free. Query on bloom filter can return false positive result on some of the queries. This is why it is called probabilistic data structure. Probability of false positive result can be controlled and calculated.

So, the way to use bloom filter is this. If bloom filter says entry is not in the set, then it is for

sure not in the set. If bloom filter says entry is in the set, then check large data structure for the entry. Most likely it will be there.

Now lets talk about structure of the bloom filter. Bloom filter is just a set of bits. Each bit represents a sub-set of elements in the element universe. Am I talking like a mathematician here? Oh well… if you have a bunch of possible values for your object, one bit of bloom filter represents not just one, but some of them.

When one of the elements present in the set, bit that represents it set to one. This also means that if we query about some other element that by chance represented by the same bit, we will get a positive result, despite the element may not be in the set – this is where probabilistic property of bloom filters come from.



Controlling probabilities

First of all, lets see what happens when you have large number of entries per bloom filter bucket. Lets say your bloom filter represents n entries in the large data structure. Lets also assume that bloom filter has m bits.

> There is an interesting problem in probability theory called birthdays problem. The problem asks following question: how many people should work in one company so that probability that two of them have birthday on the same day is above 50%. Interestingly enough, the answer is approximately square root of all possibilities. There are 365 options for birthdays, so the answer is 19.
>
> This problem often called paradox because the right answer is not intuitive. One would expect the answer to be much bigger.

Birthdays paradox applies here as well. It is enough to insert $\sqrt{n}$ entries into bloom filter for probability of single bit representing two or more elements to be more than 50%. This is a serious problem when working with bloom filter.

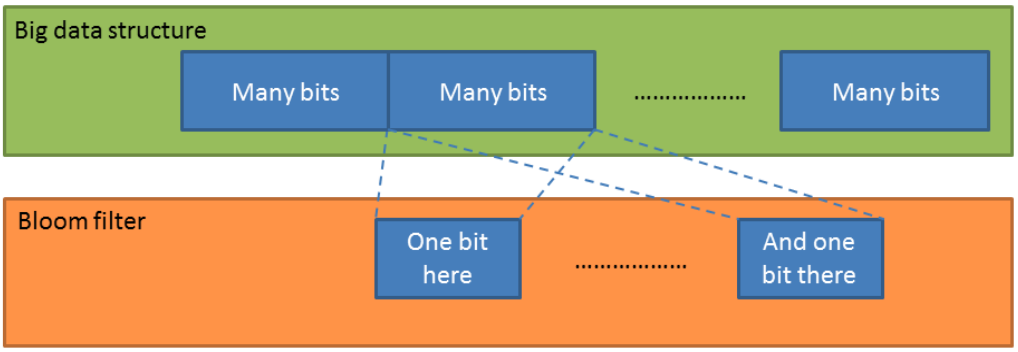
Luckily there is a way out. For simplicity, lets say we just 10 bits in the bloom filter. We added one element into bloom filter, so one of the bits is on. Now, as you remember, when querying for element that is in the filter, we will get 100% accurate result. It is when we query for element that is not in the filter where we get false positives. Lets see when we get false positive for element that is not in the filter.

With just 1 element inside of 10 bit long bloom filter, probability of getting false positive result is $\frac{1}{10}$. This is so because we assume equal distribution of the hash function that convert

element into right bit in the bloom filter. So, one element inside, means one bit is set and querying for different element can accidentally hit this bit with probability of $\frac{1}{10}$.

Now lets see how we make this probability smaller. Lets say that when inserting element into bloom filter, instead of setting one bit, we will set two bits. So now, to query if element is in the bloom, we will check two bits. Its important that for every entry, it will be two different bits. If we find that both bits set, we will conclude that element is in the filter.

In this case, probability of false positive will be $\frac{2}{10} \cdot \frac{1}{9} = \frac{1}{45}$. This is because probability of hitting first set bit is $\frac{2}{10}$ and probability if hitting second set bit is $\frac{1}{9}$. So, we clearly see that probability of hitting drops when we use several bits.



However, we cannot increase number of bits per element forever. After all, our entire bloom filter is 10 bits so using 10 bits for single element will make false positive probability of 100% after inserting just one element. In fact, jump to 100% chance of false positive is gradual. Using more bits per element decreases number of elements we can safely insert into bloom filter. Lets see how.

With just one bit per element, probability of hitting false positive answer for querying is $\frac{1}{10} = 10\%$. With two bits inside, probability of hitting false positive when querying for third element is $\frac{2}{10} = \frac{1}{5} = 20\%$. This is obviously lower than $\frac{1}{10}$.

With two bits per element, probability of hitting false positive is $\frac{2}{10} \cdot \frac{1}{9} = \frac{1}{45}$. Adding second element and querying for third, makes probability of false positive $\frac{4}{10} \cdot \frac{3}{9} + \frac{3}{10} \cdot \frac{2}{9} + \frac{2}{10} \cdot \frac{1}{9} = \frac{2}{9}$. So, its clear that probability of false positive when querying for third element decreased from $\frac{1}{5}$ to $\frac{2}{9}$ simply because we're using one more bit per element.

In fact, as we will see in later, bloom filter is a function of four parameters: overall number of bits in the bloom, number of bits per element, probability of false positive and how many elements are in the bloom.

Fine tuning bloom filter

Like I said, bloom filter is a function of four different variables. Lets name them:

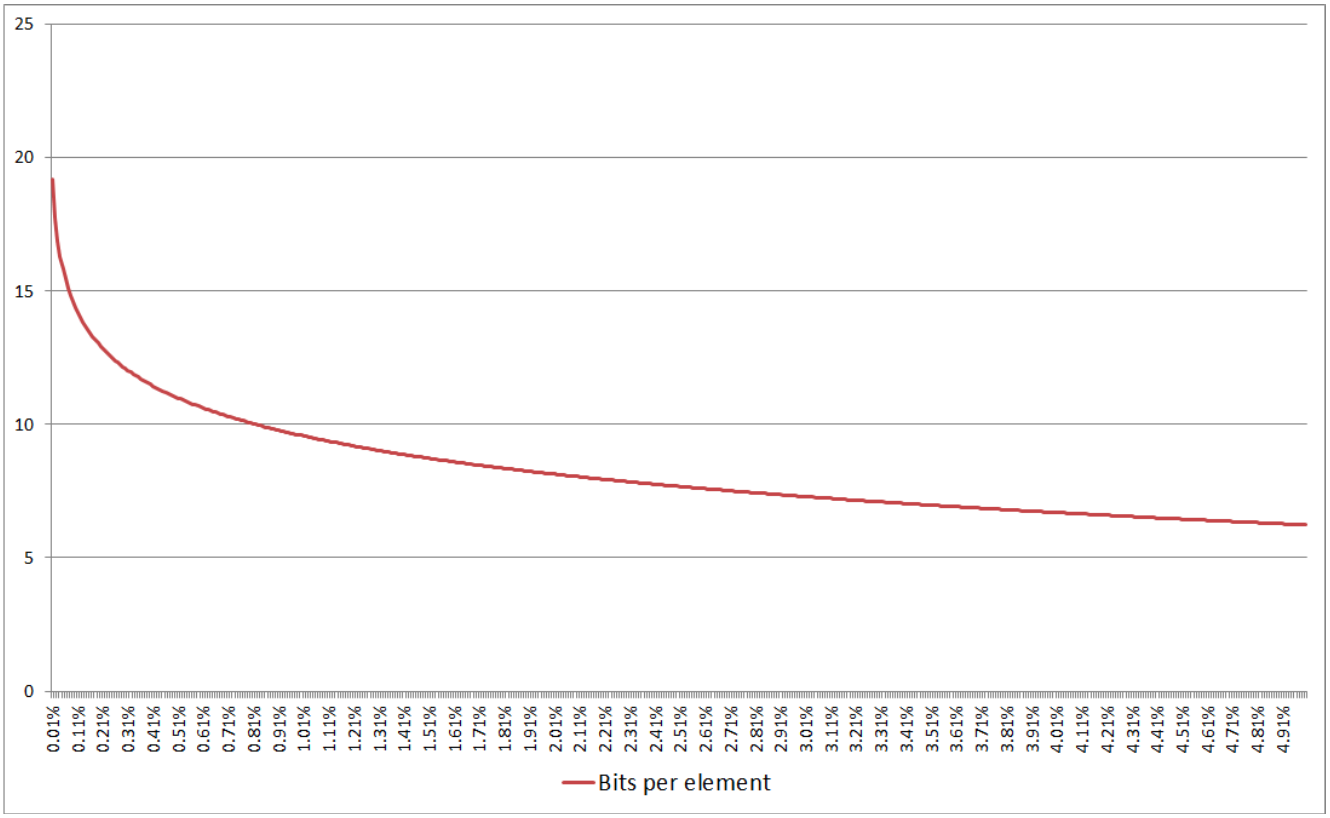
- m – number of bits in bloom filter
- k – number of bits that will represent single entry
- n – number of entries in the bloom filter when calculating probability of false positive
- p – probability of false positive answer

As we just saw, calculating probabilities for large bloom filter can be quiet complicated task. Luckily there are some approximations that can help us to get a clue of what's going on.

Normally, when calculating various properties of the bloom filter, we can start with making an assumption on maximum number of elements in the filter (n). Also, you can decide what is the probability of false positive you can tolerate. With this in hand, lets see how we choose m . I'll spare you the task of calculating probabilities and how they affect our m and give you the final formula:

$$m = -\frac{n \ln p}{\ln 2^2} = n \cdot -\frac{\ln p}{\ln 2^2}$$

Chart below shows number of bits in bloom filter per element as a function of probability of getting false positive.



As you can see, with probability around 0.01%, you will need 19 bits per element and with probability of 5% you can settle with just 6 bits per element. And with 9 bits per element, you will get 1% chance of getting false positive.

Now lets see how many bits in the set will represent single entry in the filter. Here comes another useful approximation.

$$k = \frac{m}{n} \ln 2 = \frac{m}{n} \cdot 0.693$$

So, with $\frac{m}{n} = 10$ you will set 7 bits per element in the bloom. This will give us 1% chance of false positive. From here, the math is fairly simple, so you can handle it yourself.

Deleting entries from bloom filter

As long as you set single bit per every element that you insert into bloom filter, you can always reset that bit to delete the entry. However, once you start using multiple bits per element, you cannot delete entries anymore. This is because when you reset a bit, same bit can represent other entry in the bloom filter. Bloom filters have this property of returning 100% correct answer when entry is in the bloom filter. When you zero a bit, you break this property.

In fact, you cannot delete entries from bloom filter. You can only insert new entries. This means that when implementing bloom filter and its use, it might be wise to add mechanisms that will expire it. I.e. delete entire bloom filter once in a while.

There is a way to implement bloom filter so that you'd be able to delete its entries. However, this comes with a price.

Counting bloom filter

The idea is this. Instead of having array of bits in the bloom filter, lets have an array of small counters. For instance, you may want to use 4 bit counter instead of just one bit. This obviously increases bloom filter size 4 times, but it brings a useful property.

With counter instead of plain bit, you can actually count how many times you've set this "bit". I.e. instead of setting the bit, you'll increment the counter. This allows you to delete entries. To delete an entry, decrement all counters that represent that entry.

There is one caveat however. You have to be careful with not overflowing the counter. If it overflows and becomes 0 again, this will break "true positive" property of the bloom filter. I.e. checking if entry that is in the filter is indeed in the filter will return negative result.

So, to keep the counter from overflowing, when incrementing, you should check if you've reached maximum value. Once bloom filter reaches it maximum value, you cannot change it anymore. If you increase it, it will overflow and you will break the filter. If you decrease it, well, its value will no longer represent correct number of times you've incremented it and you won't know that. So the only way around this problem is to keep the counter at its maximum value and never change it again.

To conclude, obviously this solution has two major drawbacks:

1. Bloom filter size increases significantly.
2. This approach does not solve the problem completely because if you reach maximum value of your counter, you cannot change it anymore, thus you revert to original bloom filter design (one with one bit), at least for one entry in the filter.

However, with careful design, you can use this approach to solve the problem of deletes from bloom filter using this approach, while keeping memory consumption under control.

Hash function

One last topic that I would like to touch is hash function. As we saw, we need a hash function that will receive an object as its input and produce number of buckets as its output. Obviously we can use one of the well known cryptographically secure hash functions such as SHA1 or MD5.

One problem with these hash functions is that they require more processing and may be an overkill. Another problem is that it is not obvious how to get two, three or even may-be four different hash values for the same input. One way might be running SHA1 on input, then running it on output of the first run, then running on output of the second run, etc.

While searching for various hash functions for bloom filter, I ran into one particular implementation that I liked. Its called murmur and it is available here.

Leave a Reply

Name (required)

Mail (will not be published) (required)

Website

Submit Comment

Prove you are not a computer or die *
3 + = nine

« printf() vs stream IO in C++

Spam »

- Search

- **Subscribe Now**
  email or RSS

more info

Categories

- Articles
- Blog
- Code library
- Howto
- News
- Opinion
- Other Stuff
- Programming Articles
- Resources
- Reviews
- Short articles
- System Administrator Articles

Interesting blogs

- CodeBrainz.ca
- dmiessler.com
- Evan Jones
- I geek
- Insanely Low-Level
- Ivan Novick's Code Snippets
- Nick Black
- Sparc86´s Blog
- tenshu.net
