Bloom Filter Introduction

CS110 - Professor Stern, P.

Ash Nguyen

2018/12/06

1 Introduction

1.1 What is a Bloom Filter?

Bloom Filter is a data structure that checks the existence of an element in a group of element in a probabilistic manner. Essentially, a Bloom Filter stores elements that is added to it, but consolidates the information content of the added elements into something much more compact (usually in terms of bits) that the original content of the element. However, this consolidation loses some part of the original content, therefore risking false positive results. Bloom Filter, structurally, is a bit array of m bits, where m is a positive number. The bits of a Bloom Filter can be set to either 0 or 1, depends on the elements being added to the Filter.

Primitively, Bloom Filter supports two operations:

- Adding element: Element can be added to the Bloom Filter using the same principle as hash table. Suppose for an m bit array Filter we have k hash functions, each of which will map an element in the universe to one of the bit among the m array. To add an element to the Filter, simply feed the content of the element to k hash functions, each of which will return a index between 0 and m (by the definition of each hash function). Set the indices of the Filter bit arrays to 1 if the hash function returns those indices.

- Querying element: Element can be check to see whether it has been added to the Filter using the k hash functions. Feed the content of the element we wish to find in the Filter to the k hash functions, each of which will returns a index between 0 and m for a properly implemented set of hash functions. To check if the element was added to the Filter or not, check those indices and if all of the bits are $1s$ then the element was probably added to the set at some point, and if one or more of the bits are 0 then the element was definitely not added to the Filter.

While there is a chance of returning a false positive (the element was not added to the list, but querying the element says otherwise), Bloom Filter does not allow false negative

(the element was added to the list, but querying the element says otherwise). False positives happens because the adding operation does not check whether the current bits is "occupied" (being 1 due to the previous adds) or not before setting it to 1, therefore the Filter cannot differentiate if a bit is occupied is due to one specific element or another element. Imagine a normal hash table with one hash function: if two elements are hashed onto the same slot, we have a collision and need to resolve it (either by chaining or direct-addressing with probing), but Bloom Filter in its primitive form does not resolve "collisions" and therefore risking false positive. In this regard, a hash table with no collision resolution is similar to a Bloom Filter with $k = 1$.

1.2 Advantages and disadvantages of Bloom Filter

There are three main advantages of Bloom Filters:

- Bloom Filter has a good space complexity: For every Filter we can specify the m bits array that constitute the Filter, and with only m bits we can store unlimited elements without the need to resize, albeit with more elements added to a Filter the false positive rate increase, as intuition suggests. This is a huge advantage for a storage system compared to other data structures like linked list or hash table.

- Bloom Filter querying takes constant time: The querying of element in the Filter takes constant time, as we only need to hash the element k times for the k functions, then compare the results with k indices in the bits array. This process takes constant time regardless of how many elements had been added to the Filter.

- Bloom Filter can be parallelized: Since the the add/query operation of the Filter is performed under k different hash function, they can be processed independently. Changing/comparing bits also can be performed independently on each bit of the bits array. Therefore the workload of performing operations by a Filter can be parallelized onto different processing units, speeding up the operations if hashing k functions takes too much time on a single processing unit.

There are also some disadvantages associated with Bloom Filters that is worth considering when using them:

- Bloom Filter querying can return false positive: As mentioned it is possible for a Bloom Filter to return yes even though the element was not previously added to the list.

- Bloom Filter cannot store the element information in its entirety: Because the Filter compressed the information in an element to bits, we cannot store all the information an element contains in a Bloom Filter. For example, with a hash table that has collision-resolution we can store the word "Professor Stern" in a slot (or in a chained list associated with a slot on the table), end we can later retrieve it completely by checking that slot. For a Bloom Filter, the word "Professor Stern" will be turned into bits on an m bits

array, and we cannot retrieve it later because the information is lost.

- Bloom Filter does not support deleting: We cannot delete elements from a Filter because changing the value of a bit in the bits array that associated with an added element will not guarantee that we won't accidentally delete another element that was hashed onto the same bit. However, this problem can be addressed using a modified version of the Filter.

1.3 False positive and false negative

By initializing all initial values of the bits array to 0 and properly hash elements to the Filter, it is impossible to get a false negative. With a set of k hash functions that uniformly hash any element to m bits array and n element already added, we have the probability of false positive being:

$$Pr = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx (1 - e^{-kn/m})^k \tag{1}$$

while the above approximation can be proved by expanding a Taylor expansion for the left hand side.

Differentiate (1) with respect to k to find the k corresponds to the optimal value of Pr, we have:

$$k_{optimal} = ln2 \times \frac{m}{n} \tag{2}$$

Assuming we choose the optimal k, plugging the results of (2) to (1) and solve for $\frac{m}{n}$ we have:

$$\frac{m}{n} = -\left(\frac{lnPr}{ln2}\right)\frac{1}{ln2} = -\frac{log_2 Pr}{ln2} \tag{3}$$

So the optimal k is:

$$k_{optimal} = -log_2 Pr \tag{4}$$

1.4 Practical usage of Bloom Filter

Bloom Filter is generally preferred in situations where querying data using normal data structures might take a lot of time or situations where we only care about determining whether something has not been added before (and do not care much if that thing has been added). Some applications of Bloom Filter include:

- Cache system: Some cache system might want to employ Bloom Filter to prevent caching things that is only requested once and then forgotten. For example, a web

caching service like Archive.org or Google Web Cache can implement a primitive Bloom Filter to check for requests of cached websites, and if the Filter response "No, this has not been requested before and I'm 100% sure of it." then the website might not be worth caching since nobody seems to be interested in viewing it anyway (except maybe someone using http://www.theuselessweb.com, a service that takes you to random, useless website). If the Filter says "This probably has been requested at least one before." then the caching services might want to cache that website for more than one user seem to request it.

- Detecting malicious urls: The Internet is a big and wild place where every link we click can be from someone who wants to steal our information by phishing. Therefore, some web browsers do use Bloom Filter to check whether a website is among the recorded list of malicious urls, and if the Filter say No then it's definitely not malicious (or at least not has been recorded to be so). If the Filter says Yes however, we might want to still re-check the website against a database of malicious website. By this mechanism the Filter can help us to not waste the time of people who only visit the Internet for cute cat videos or useless, harmless websites (and let's be optimistic and believe that the good part of the Internet is bigger than the bad part, therefore it's more likely for the Filter to says "No, this is not in the harmful list, please go on with your cat videos.".

- Querying large database: There are some database that contains petabytes of information, storing on many computers at the same time. By using the Bloom Filter, we can pre-check if some elements is definitely not in the stored database with minimum time to make sure we don't waste time looking through the huge database for a non-existent element.

2 Python implementation

The following implementation of Bloom Filter assumes that the elements added to the Filter are strings and uses a group of hash function that is defined as following:
1. Calculating a unique value given an input string by incrementing the ord() of each character in the string.
2. Using a pseudo-random number generator in Python, setting the seed of the unique number in step 1 for querying. Calculate the product of the given number (ranging from 1 to k) with a random number from 1 to the unique indicator from step 1.
3. Taking the modulus of the result of step 2 with m to return the index.

The above hash function is not really an ideal one, since the distribution of hashed values are not uniform, as indicated in Figure 1. [1]

[1] Pardon the code box that does not really resembles real Python code. I tried to make the keywords the green color I see in Python (I think it's Forest Green or something like that) but I can't replicate the color style in RGB. Guess I'm bad with colors.

```
#importing relevant packages
import bitarray
import random
import math

#a hash function that return the index from 0 to m, with the
#input of the m, the element to hash and the order of the
#hash function(number)
def hashit(length,element,number):
    #calculating a unique number for each string element
    val=0
    for chr in element:
        val += ord(chr)
    #setting seed for this unique value so querying can be performed
    random.seed(val)
    #calculating a number unique to the element and the order of the hash
    function
    ans=random.randint(1,val)*number
    return(ans % length)

#defining the Bloom Filter class
class Bloom(object):
    def __init__(self,m,k):
        self.b_array=bitarray.bitarray(m)   #creating the bits array
        self.len=m    #initialize the length of the bits array
        self.hashfun=k   #initialize the number of hash function to use
        self.b_array.setall(0)   #setting all the bits to 0s

    def add(self,element):
        #setting all indices the k hash functions return to 1s
        for _ in range(self.hashfun):
            index=hashit(self.len,element,_)
            self.b_array[index]=1

    def query(self,element):
        #checking for at least 1 zero in the indices the k
        #hash function hashed to. If there's 1 zero, the element is not in
    the
        #Filter. If there's no zero, the element might be in the Filter
        for _ in range(self.hashfun):
            index=hashit(self.len,element,_)
            if self.b_array[index]==0:
                return("Element was not added in the Filter")
        return("Element was probably added to the Filter")

#a function to calculate the optimal k for a given
#false positive rate Pr
def op_k(Pr):
    k=-math.log(Pr,2)
    return(int(k))

#a function to calculate the optimal m for a given
```

```
#false positive rate Pr and number of expected element n
def op_m(Pr,n):
    m=-n*math.log(Pr,2)/math.log(2)
    return(int(m))

#testing the Bloom Filter
bloom=Bloom(op_m(0.01,1000),op_k(0.01))
bloom.add("Ash")
bloom.query("Stern")
bloom.query("Ash")
```

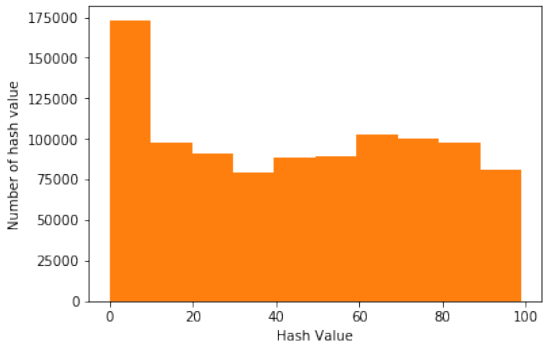

Figure 1: Histogram of hashed value for hashit() for 100000 random words

Next, we will use a non-cryptographic (for faster calculation) hash function named MurmurHash invented by Austin Appleby, which uses multiplication and rotation optimizing for computer configuration (32 or 64) [2]. The hash function seems to provide a pretty uniform hashed values, as shown in Figure 2.

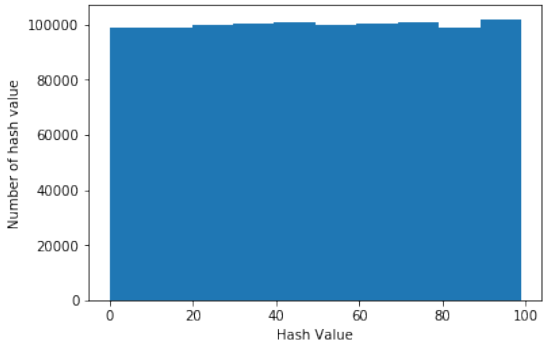

Figure 2: Histogram of hashed value for mmh3() for 100000 random words

[2] cs110- hashing: Here I roughly analyzed the performances of two hash functions family: one I wrote and one non-cryptographic hash function I found. I compared them using two simple histograms to test whether they will hash elements into uniform positions or not

The following is Python implementation of Bloom Filter using MurmurHash:

```
!pip install mmh3
import mmh3

class Bloom(object):
    def __init__(self,m,k):
        self.b_array=bitarray.bitarray(m)
        self.len=m
        self.hashfun=k
        self.b_array.setall(0)

    def add(self,element):
        for _ in range(self.hashfun):
            index=mmh3.hash(element,_) % self.len
            self.b_array[index]=1

    def query(self,element):
        for _ in range(self.hashfun):
            index=mmh3.hash(element,_) % self.len
            if self.b_array[index]==0:
                return(False)
        return(True)
```

3 Scaling behavior

3.1 Scaling of memory requirement and access time

Assuming we have implemented an optimal Bloom Filter. As equation (3) suggests, if we want to keep the expected number of element to add a constant we will see that the required memory (m) scales with $O(log_2 Pr)$, with Pr being the false positive rate. Similarly, if we keep the false positive rate constant (or as (4) suggests, equals to keep k constant), the required memory will scale with $O(n)$, n being the number of added elements.

Because in a proper implementation of Bloom Filters, when we query an element we only hash the element k times then check for k indices in the m bits array. Therefore, the access time scale linearly with k. By using (4) we can deduce that the access time scales with $O(log_2 Pr)$. Similarly, if we keep the m bits array constant, (2) indicates that the access time also scales linearly with $\frac{1}{n}$: as n increase, the access time decreases. In reality where we have a fixed k and a fixed m, the access time will tend to be a constant, which is an advantage of Bloom Filter. [3]

[3] cs110-complexity: I used the theoretical predictions (with several assumptions that are roughly satisfied by the MurmurHash function like uniform hashing) to predict the scaling behavior of the time and space complexity for a primitive implementation of Bloom Filter in Python. I proceeded to demonstrate this theoretical prediction by empirical testing

If we have a general Bloom Filter implementation though, equation (1) can be transformed to:

$$m = -\frac{kn}{ln(1 - log_k Pr)} \tag{5}$$

$$k = -\frac{m}{n} ln(1 - log_k Pr) \tag{6}$$

According to (5), if we keep k and n constant, memory size will scale with $O(ln(1 - log_k Pr)^{-1}$ and will scale linearly with number of stored items if we keep Pr and k constant. Equation (6) suggests that if we keep m and n constant then access time will scale with $O(ln(1 - log_k Pr)$.

3.2 Scaling of false positive rate

The scaling behavior is presented in Figure 3 for a Bloom Filter with $m = 1000$ and $k = 25$.

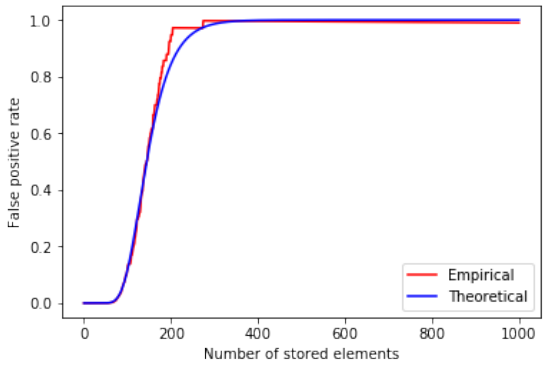

Figure 3: Theoretical versus empirical false positive rate of a Bloom Filter

References

[1] Wikipedia. (n.d.). *Bloom Filter*. Retrieved March 18th, 2018 from:
https://en.wikipedia.org/wiki/Bloom_filter

[2] Cao, P. (May 7th, 1998). *Bloom Filter - The Math*. University of Wisconsin-Madison. Retrieved March 18th, 2018 from:
http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html

[3] Ripeinu, M., Iamnitchi, A. (n.d.). *Bloom Filter - Short Tutorials*. University of Chicago. Retrieved March 18th, 2018 from:
https://www.cs.uchicago.edu/~matei/PAPERS/bf.doc

Appendix

Code to generate Figure 1 and Figure 2

```
import matplotlib.pyplot as plt
import string

def randomword(length):
    return ''.join(random.choice(string.ascii_lowercase) for i in range(
    length))

randomwords=[]
for i in range(100000):
    randomwords.append(randomword(10))

a=[]
for _ in randomwords:
    for i in range(10):
        a.append(mmh3.hash(_,10)%100)



plt.hist(a)
plt.xlabel("Hash Value")
plt.ylabel("Number of hash value")
plt.show()
```

Code to generate Figure 3

```
import mmh3
import matplotlib.pyplot as plt
import string
import math
import random
import bitarray

def randomword(length):
    return ''.join(random.choice(string.ascii_lowercase) for i in range(
    length))

randomwords=[]
for i in range(100000):
    randomwords.append(randomword(10))

class Bloom(object):
    def __init__(self,m,k):
        self.b_array=bitarray.bitarray(m)
        self.len=m
        self.hashfun=k
        self.b_array.setall(0)

    def add(self,element):
        for _ in range(self.hashfun):
            index=mmh3.hash(element,_) % self.len
```

```
            self.b_array[index]=1

    def query(self,element):
        for _ in range(self.hashfun):
            index=mmh3.hash(element,_) % self.len
            if self.b_array[index]==0:
                return(False)
        return(True)



def get_fp_rate(n):
    bloom=Bloom(1000,25)
    count=0

    for _ in randomwords[0:n]:
        bloom.add(_)
    for _ in range(len(randomwords)):
        check1=bloom.query(randomwords[_])
        if check1==True and _>=n:
            count+=1
    return(float(count)/100000)

a=[]
x=[_ for _ in range(1000)]
y=[(1-math.exp(-(_*25)/1000))**25 for _ in x]

for _ in range(1000):
    a.append(get_fp_rate(_))

ash1, =plt.plot(x,a,"red")
ash2, =plt.plot(x,y,"blue")
plt.ylabel('False positive rate')
plt.xlabel('Number of stored elements')
plt.legend([ash1, ash2], ['Empirical', 'Theoretical'])
plt.show()
```