Bloom Filters in Distributed Systems

Matthew Flammia and Gildian Gonzales

Introduction

We will discuss what a Bloom Filter is, and one application of it in a distributed system.

Bloom Filters have many applications, so we will discuss Burton Howard Bloom’s original concept first, then cover how it can be used in a Set Reconciliation in a distributed system.

What is a Bloom Filter?

A Bloom Filter is a probabilistic data structure where *n* elements are run through *k* hashing algorithms, and indexed into an array of length *m*, when an index is hit, we change its value from 0 to 1. After doing this for all *n* elements, we get our populated Bloom Filter.

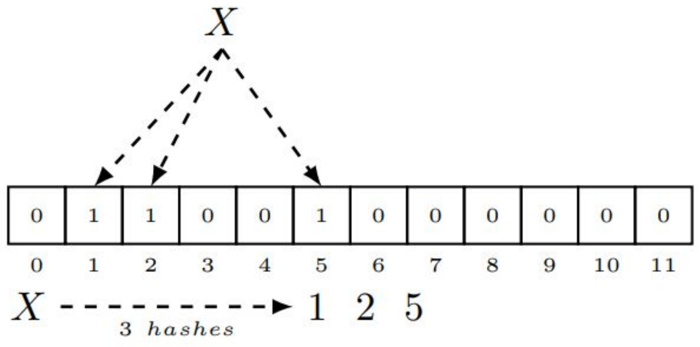

Figure 1: Inserting an element into a bloom filter.

In this example, *m* is 12 and *k* is 3

How can we use a Bloom Filter for Membership?

After populating our Bloom Filter, we can check the membership of an element by running it through our hashing algorithms, if all indexes hit are 1's, then we know our element is a part of our set. Any 0 would indicate that the element does not exist in the set. A problem we run into though is there is a false positive rate, but the benefit is that we will never have a false negative. The query time is $\log_2(1/\varepsilon)$

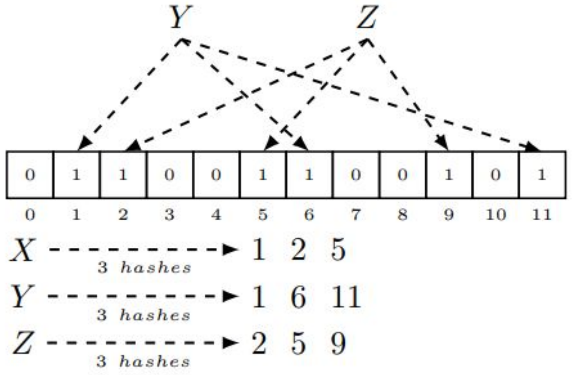

Figure 2: Inserting multiple elements into a bloom filter. "X" was never inserted, yet it appears as if it has, i.e. a false positive has occurred.

The Disadvantage of a Bloom Filter

As stated before, we encounter the issue of a false positive while running the Bloom Filter. This can cause us to accidentally assume membership when there isn’t, causing any subsequent search for an element to fail. This is unavoidable, but can be calculated with this equation to our preference.

m is our array size, *k* is our amount of hash functions, *n* is the number of elements

Let’s call this rate ε

$$\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k}$$

A Bloom Filter of size 10 with 3 hash functions and 4 elements has a false positive rate of approx. 0.37 or 37%

Bloom Filter Space Usage

The space usage of Bloom Filters can be written as $1.44n*\log_2(1/\varepsilon)$ bits. If we wanted to compute the space complexity of a false positive rate of 1/32

$1.44*n*\log_2(1/(1/32)) \rightarrow 1.44*n*\log_2(32) \rightarrow 1.44*n*5 \rightarrow 7.2n$ bits

While we would want to minimize our space, we would have to allow for a higher false positive rate. This tradeoff has to be balanced correctly to make a Bloom Filter worth using.

Calculating A Bloom Filter Given A F.P. Rate

We can calculate all the parameters of our Bloom Filter given a desired false positive rate, ε. Lets use 1/32 again.

m, the length of our array would be $1.44n*\log_2(1/\varepsilon)$

k, our number of hash functions needed is $\log_2(1/\varepsilon)$

We find that $m = 7.2n$ and $k = 5$

The Advantage of Bloom Filters

Bloom filters are faster than comparing every element, allowing for us to quickly determine if an element is a member of a set. It is small enough that it doesn’t consume excessive space, and it doesn’t store a copy of the elements.

By cutting back on size and being easy to implement, Bloom Filters have become an important data structure.

Overall Bloom Filters with the right parameters can approximate membership with high probability, but let’s now show its application in a distributed system.

Bloom Filters in Distributed Systems

Say we have two nodes, *A* and *B* who have sets of elements that are the same. Assume that *B* wants to check if its missing any elements that *A* has. We call this the Set Reconciliation Problem.

The naive approach would be for *A* to send every element it has to *B* so that *B* can compare the elements to what it has. Obviously this is a bad idea, and consumes tons of bandwidth. We can solve this problem using Bloom Filters.

$$A = [1\ 0\ 1\ 1] \rightleftharpoons B = [1\ 0\ 1\ 0]$$

Just by looking at *A* and *B*'s Bloom Filters,
we can tell something is different

Set Reconciliation with Bloom Filters

Assume our previous problem where *B* wants to check if its missing any elements from *A*. We can have *A* send its Bloom Filter to *B* where *B* can do a reduction to see if its missing any elements. If any index is negative, we can say that *B* is missing that element, while if any index is positive we can say *A* is missing that element.

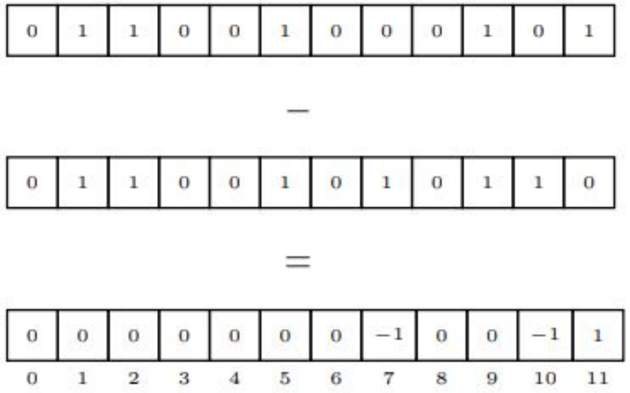

Figure 3: Reducing a bloom filter from another bloom filter

Problems with this Set Reconciliation

While we showed that a Bloom Filter cannot have a false negative, it still can have a false positive. The only way to avoid a false positive would be to create a larger Bloom Filter, which defeats the purpose of saving bandwidth. There are many solutions to this problem, such as Invertible Bloom Lookup Tables (IBLT), Counting Bloom Filters (CBF), etc.

The main benefit is that we save a tremendous amount of bandwidth with only minor sacrifices. This tradeoff is acceptable, and with the other implementations of Bloom Filters, this problem can be made more efficient.

Conclusion

Bloom Filters are a powerful data structure, which can reduce the bandwidth usage between two nodes drastically. Along with solving this problem, Bloom Filters can be used in a number of applications. The Wikipedia article for Bloom Filters has 15 different applications and extensions, showing how versatile this idea is.

Sources

Data Structures and Algorithms (cs.DS); Distributed, Parallel, and Cluster Computing (cs.DC), arXiv:1910.07782 [cs.DS], https://arxiv.org/abs/1910.07782