Bloom filter

Discrete Stochastics - Prof. Dr. Andreas Vogt
FHNW

Kevin Buman

May 15, 2020

Contents

Summary

The purpose of this assignment was to implement a basic bloom filter operating on strings. The filter can read strings as well as check whether a given input is contained within the data structure. The implementation was done using a `BitSet` from the `java.util` package. It uses `murmur3_128` as a hash function. In the following article I will lay out the basics of bloom filters and also discuss the implementation in Java.

Concept

Bloom filters were invented in 1970 by a computer scientist by the name of Burton Howard Bloom. A bloom filter can very quickly answer the yes or no question "is this item in the set?". More importantly, a bloom filter can only ever produce false positive errors, meaning that if an element is not part of the set, the filter will never tell you otherwise. There is, however, a certain possibility for false positives. Hence, a bloom filter can tell you that a certain element is part of the set, when in actuality it is not.
Typically, an empty bloom filter is an array of m bits, all initialized to 0. In addition, k different hash functions are defined. Each hash function maps its input to one of the m array-positions within the set. Due to the nature of hash functions, this creates a uniform random distribution. The probability that the function k_i maps an element to a certain bit is therefore

$$\frac{1}{m} \tag{1}$$

When inserting a new element, it is fed to each of the k hash functions. The k resulting bits are set to 1. To query for an element, it is also fed through each hash function. If one of the resulting bits contains a 0, one can say with absolute certainty that the item is not part of the set.

This simple mechanism is also the cause of two inherent problems. Firstly, since different elements can be mapped to the same bits, it is not possible to remove an element from the set. To support this, the data structure would have to be extended to keep track of what elements have been added. Secondly, you can never be completely certain that an element is contained within the set.

False Positive Probability

Equation (1) yields the inverse probability:

$$\left(1 - \frac{1}{m}\right) \tag{2}$$

Assuming that the output of the k hash functions is independent, we can reason that

$$\left(1 - \frac{1}{m}\right)^k \tag{3}$$

and using Euler's identity we get

$$\left(1 - \frac{1}{m}\right)^k = \left(\left(1 - \frac{1}{m}\right)^m\right)^{\frac{k}{m}} \approx \left(1 - e^{-\frac{k}{m}}\right)^k \tag{4}$$

With this information we can approximate the probability of a false positive after n inserted elements to be

$$\varepsilon \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \tag{5}$$

Equation (1) shows that the probability of getting a false positive grows with the number of inserted elements. With this knowledge, it is possible to put a theoretical upper bound on the probability.

Determining the right k-value

Most bloom filters can be initialized by specifying the expected number of insertions n and the accepted False Positive Probability ε. With this information we can compute the optimal length of the data structure as well as the optimal value k for the number of hash functions:

$$m_{opt} = -\frac{n \ln(\varepsilon)}{(\ln 2)^2} \tag{6}$$

$$k_{opt} = \frac{m}{n} \ln 2 \tag{7}$$

Remark: due to various assumptions made in the calculations, the observed value for ε can exceed the initially specified probability.

Performance

To demonstrate the implemented bloom filter, a sample set of roughly 58000 strings is created. In addition, we create a list of and all elements. To verify the correctness of the code, we randomly generate a large number of candidate elements and check each one against the set. For every positive result(i.e. the filter tells us that the word is contained within the set) we double check by verifying that the list in fact does contain the element. If this is not the case we count the event as a false positive. This allows us to calculate the percentage of false positives and compare it to the initially defined $\varepsilon_{accepted}$.

Using the aforementioned optimization for k, m, n, we actually get a consistent false positive probability that lies in a narrow margin around the specified value for $\varepsilon_{accepted}$.

Benefits and drawbacks

There are good reasons why the idea of the bloom filter has stuck around with us for four decades now. It presents a very space efficient data structure to verify an elements' membership to a given set. It also can be fine-tuned to use either less space or yield lower false positive probabilities. With a search-/insert time-complexity of $\mathcal{O}(k)$ it is also very fast.

Sadly, there are also some disadvantages when using a bloom filter. In its basic form, the data structure cannot handle the removal of elements, as we are not keeping track of the elements added to it. Therefore, one would need to implement further logic to allow for such operations.
In addition, you can never be 100% confident that an element is indeed contained within the data structure.

Applications

Due to its relative simplicity and low memory footprint, bloom filters have found a wide variety of use cases. Below I list some of the more notable ones:

- Speed up synchronization of Bitcoin wallets by deciding whether to transfer a piece of data or not
- Checking of entered URL's in Google Chrome to detect whether the they pose a threat or not
- Used in in-memory key-value stores such as Redis as well as distributed databases like Apache Cassandra
- Detection of weak passwords by comparing entered passwords against a set of known weak passwords

References

[1] Wikipedia: Bloom filter,
`https://en.wikipedia.org/wiki/Bloom_filter`
(Accessed 15.05.2020)

[2] Opengenus: Applications of Bloom Filter,
`https://iq.opengenus.org/applications-of-bloom-filter/`
(Accessed 15.05.2020)

[3] Github: google/guava,
`https://github.com/google/guava/blob/master/guava/src/com/google/common/hash/BloomFilter.java/`
(Accessed 15.05.2020)