# Bloom Filters

## Overview

Bloom filters are probabilistic data structures that support membership queries. That is, they are designed to support two operations:

1. **contains?** Is item $x$ in the set?

2. **add** Add a new item $y$ to the set

The first operation isn't a deterministic operation; that is, it's probabilistic. There's a chance that $contains?(x)$ will return True even if $x$ is not in the list (false positive). But if $contains?(x)$ returns False, then $x$ is *definitely* not in the list; that is, bloom filters never return false negatives.

"Abstract" from *Scalable Bloom Filters* by Almeida et al:

> "Bloom Filters provide space-efficient storage of sets at the cost of a probability of false positives on membership queries. The size of the filter must be defined a priori based on the number of elements to store and the desired false positive probability, being impossible to store extra elements without increasing the false positive probability. This leads typically to a conservative assumption regarding maximum set size, possibly by orders of magnitude, and a consequent space waste."

An important property of bloom filters is the linear relation between the filter size and the number of elements that can be stored. As detailed above, traditional bloom filters are less flexible to space requirements. So in this paper/summary, I'll compare and constrast the use of traditional bloom filters vs. scalable bloom filters proposed by Almeida et al.

## Bloom Filters Internals

A Bloom filter is traditionally implemented by a single array of $M$ bits, where $M$ is the filter size. On filter creation all bits are reset to 0's. A filter is also parameterized by a constant $k$ that defines the number of hash functions used to activate and test bits on the filter. Each hash function outputs an index in the range $\{1, \ldots, |M|\}$.

Let $h_1, \ldots, h_k$ be the hash functions. In order to query element $x$ in a bloom filter, it suffices to verify that all bits in indexes $h_1(x), \ldots, h_k(x)$ are set. If one or more of these bits is not set, then $x$ is definitely not present in the filter. Otherwise, if all these bits are set, then the element is *probably* in the filter. It's clear that this approach is probabilistic. As a result, an error probability exists for positive matches, since the tested indexes might have been set by the insertion of other elements.

We must ensure, just as in the case of Count-Min sketches studied last, that the hash functions $h_1, \ldots, h_k$ are pairwise independent to reduce hash collisions.

### False Positives

As stated above, false positives can occur when testing for the presence of some element that wasn't added to the bloom filter. This is caused by the insertion of other elements whose hash values collide with one or more hashes of this element. The probability of a given bit being set in a filter is the fill ratio $p$ between the number of set bits in the slice and the slice size $m$ where $0 \leq p \leq 1$. Let $P$ be the false positive probability. If the filter consits of only one slice (dimension 1-array), then $P = p$. On the other hand, if the filter is adapted to consist of several slices (dimension $k$-array), then $P = p^k$ if $m$ is reasonably large.

In each slice, the probability that a 0 bit is set after adding a new element is $1/m$; then, it will remain unset with probability $1 - 1/m$. If $n$ elements have been inserted, the probability that the given bit is still 0 is $(1 - 1/m)^n$. As a result, the probability that a specific bit in a slice is set after $n$ insertions, which is also the expected fill ratio $p$, is

$$p = 1 - \left(1 - \frac{1}{m}\right)^n \tag{1}$$

### Bounding the Error

From the previous paragraph, it's clear that the error probability $P$ increases with $n$ and decreases with $m$ and $k$. We can choose $k$ (and thus $m$) such that for a given filter size $M = m \times k$, the number of stored elements $n$ is maximized while keeping the error probability below a certain value $P$.

Through a series of reductions, Almeida et al, proposed finding $k$ using the formulae $k = \log_2 \frac{1}{P}$ where $P$ is the error probability set by the end user.

## Scalable Bloom Filters

A scalable bloom filter (SBF) addresses the problem of having to choose on a priori the maximum size for the set, and allows for an arbitrary growth of the set being represented. Key ideas proposed by Almeida et al:

- SBF is made up of a series of one or more traditional bloom filters; when these filters are full due to the limit on the fill ration, a new one is added. Querying is done across all filters in use.
- Each successive bloom filter is created with a smaller maximum error probability on a geometric progression, so that the compounded probability over the whole series converges to some wanted value.

The SBF starts with one filter with $k_0$ slices and error probability $P_0$. When this filter gets full, a new one is added with $k_1$ slices and $P_1 = P_0 r$ error probability, where $r$ is the tightening ratio with $0 < r < 1$. As such $P_i = P_0 r^i$. The number of slices for slice 0 is $k_0 = \log_2 P_0^{-1}$ and for slices $i > 1$ is

$$k_i = \log_2 P_i^{-1} = \log_2 (P_0 r^i)^{-1} = \log_2 P_0^{-1} r^{-i} = \log_2 P_0^{-1} + \log_2 r^{-i} = k_0 + i \log_2 r^{-1}.$$

To have each $k_i$ be an integer, the natural choice will be $r = 1/2$, so that $k_i = k_0 + i$.

## References

- *Network Applications of Bloom Filters: A Survey* by Broder and Mitzenmacher
- *Less Hashing, Same Performance* by Kirsch and Mitzenmacher
- *Scalable Bloom Filters* by Almeida et al
- *Compressed Bloom Filters* by Michael Mitzenmacher