CSC721 Bloom Filters Fall 2017

From Wikipedia:

A Bloom filter is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not in other words, a query returns either "possibly in set" or "definitely not in set". Elements can be added to the set, but not removed (though this can be addressed with a counting filter); the more elements that are added to the set, the larger the probability of false positives.

Bloom proposed the technique for applications where the amount of source data would require an impractically large amount of memory if conventional error-free hashing techniques were applied. Consider an example in which 10% of the data accesses require an expensive disk access. With sufficient core memory, an error-free hash could be used to eliminate all disk accesses; However, with limited core memory, Bloom's technique uses a smaller hash area but still eliminates most disk accesses. For example, a hash area only 15% of the size needed by an ideal error-free hash still eliminates 85% of the disk accesses–an 85/15 form of the Pareto principle.[1,2]

Algorithm:
An empty Bloom filter is a bit array B of m bits, all set to 0. There must also be k different hash functions defined, each of which maps an element to one of the m array positions, generating a uniform random distribution. Typically, k is a constant with $k << m$. The value of m is chosen proportional to the number of elements to be added. The precise choice of k and the constant of proportionality number of elements to m are determined by the intended false positive rate of the filter.

Adding a New Element to the Set
To add an element, feed it to each of the k hash functions to get k array positions. Set the bits at all these positions to 1.

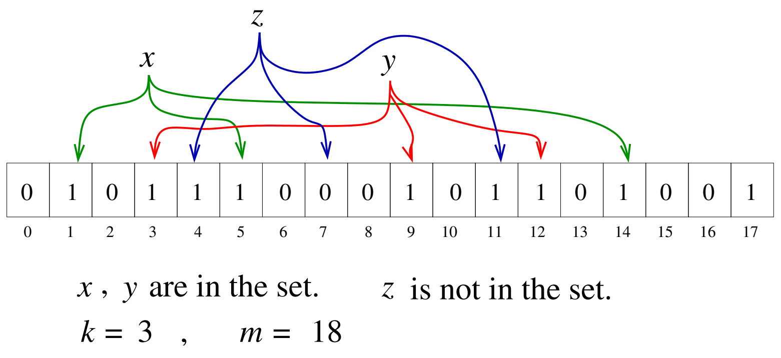
Testing for Set Membership:
To test whether an element $X$is in the set, apply each of the k hash functions to X. Obtain k array positions $[p_1, p_2, p_3, ..., p_k]$. If there exists j in the range $1 \leq j \leq k$ such that $A[p[j]] == 0$, then the element is definitely not in the set. If the element were in the set, then all the bits $A[p[j]]$ would have been set to 1 when the element was inserted. If $A[p[j]] == 1$ for all j in the range $1 \leq j \leq k$, then either:

- the element is in the set,
- or the bits have by chance been set to 1 during the insertion of other elements, resulting in a false positive.

[1]The Pareto principle (also known as the 80/20 rule, the law of the vital few, or the principle of factor sparsity) states that, for many events, roughly 80% of the effects come from 20% of the causes.

[2]One of the motivations for RISC architectures is that roughly 80% of instructions in a compiled program come from only 20% of the instruction set.

In a simple Bloom filter, there is no way to distinguish between the two cases, but more advanced techniques can address this problem. An example with $m = 18$ and $k = 3$ is illustrated below



The requirement of designing k different independent hash functions can be prohibitive for large k. For some well-chosen hash functions, there should be little if any correlation between different bit-fields of such a hash. This type of hash function can be used to generate multiple "different" hash functions by slicing its output into multiple bit fields. Alternatively, one can pass k different initial values (such as 0, 1, ..., k 1) to a hash function that takes an initial value. For larger m and k, independence among the hash functions can be relaxed with negligible increase in false positive rate. Specifically, Dillinger & Manolios [1] show the effectiveness of deriving the k indices using enhanced double hashing or triple hashing, variants of double hashing that are effectively simple random number generators seeded with the two or three hash values.

Removing an element from this simple Bloom filter is impossible because false negatives are not permitted. An element maps to k bits, and although setting any one of those k bits to zero suffices to remove the element, it also results in removing any other elements that happen to map onto that bit. Since there is no way of determining whether any other elements have been added that affect the bits for an element to be removed, clearing any of the bits would introduce the possibility for false negatives.

One-time removal of an element from a Bloom filter can be simulated by having a second Bloom filter that contains items that have been removed. However, false positives in the second filter become false negatives in the composite filter, which may be undesirable. In this approach re-adding a previously removed item is not possible, as one would have to remove it from the "removed" filter.

It is often the case that all the keys are available but are expensive to enumerate (for example, requiring many disk reads). When the false positive rate gets too high, the filter can be regenerated; this should be a relatively rare event.

Probability of False Positives Assume that a hash function selects each array position with equal probability. If m is the number of bits in the array, the probability P_{one} that a certain bit is not set to 1 by a one hash function during the insertion of an element is:

$$P_{\text{one}} = 1 - \frac{1}{m}$$

If k is the number of indpendent hash functions, the probability P_{any} that the bit is not set to 1 by any of the hash functions is

$$P_{\text{any}} = \left(1 - \frac{1}{m}\right)^k$$

If we have inserted n elements, the probability $\hat{P}$ that a certain bit is still 0 is

$$\hat{P} = \left(1 - \frac{1}{m}\right)^{kn} .$$

The probability $\tilde{P}$ that it is 1 is therefore:

$$\tilde{P} = 1 - \left(1 - \frac{1}{m}\right)^{kn} .$$

Mitzenmacher and Upfal [2] derive an expression for the probability of a false positive. After all n items have been added to the Bloom filter, let q be the fraction of the m bits that are (still) set to 0. The number of bits still set to 0 is qm.

When testing membership of an element not in the set, for the array position given by any of the k hash functions, the probability that the bit is found set to 1 is $1q$. The probability that all k hash functions find their bit set to 1 is $(1 - q)^k$. Further, the expected value of q is the probability that a given array position is left untouched by each of the k hash functions for each of the n items, which is (as above)

$$E[q] = \left(1 - \frac{1}{m}\right)^{kn} .$$

It is possible to prove, that q is very strongly concentrated around its expected value. In particular, from the Azuma–Hoeffding inequality, Mitzenmacher and Upfal[2] prove that:

$$\mathrm{P}\left(|q - E[q]| \geq \frac{\lambda}{m}\right) \leq 2e^{-2\lambda^2/kn}$$

Because of this, we can say that the exact probability of false positives is

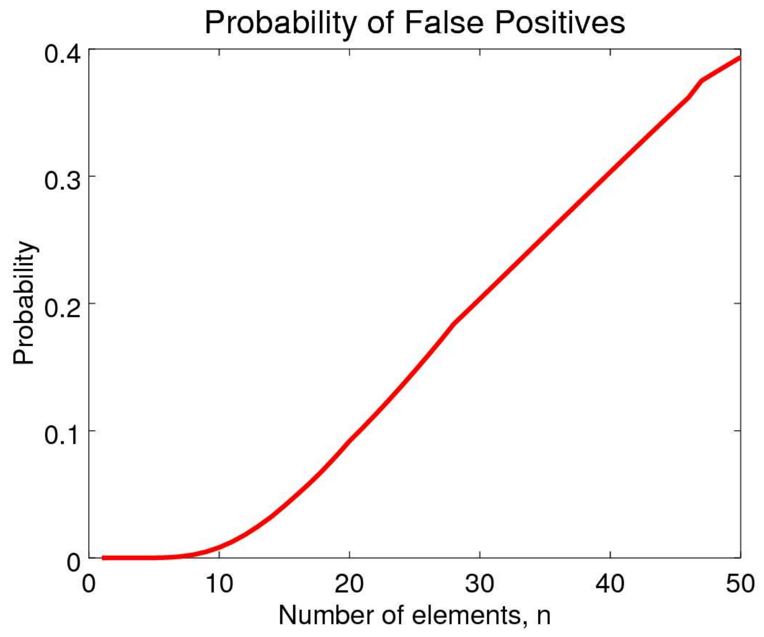
$$(1 - E[q])^k = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k \tag{1}$$

Optimal Number of Hash Functions

The number of hash functions, k, must be a positive integer. Putting this constraint aside, for a given m and n, the value of k that minimizes the false positive probability is

$$k = \frac{m}{n}\ln(2) . \tag{2}$$

To gain some intuition here, we plot the equation (1) with $m = 100$, $1 \leq n \leq 50$ and k chosen optimally according to equation (2).



There is an extensive literature on Bloom filters. We have only scratched the surface of a very well researched topic.

References

[1] Dillinger, Peter C.; Manolios, Panagiotis (2004), "Bloom Filters in Probabilistic Verification", Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design, Springer-Verlag, Lecture Notes in Computer Science 3312

[2] Mitzenmacher, Michael and Upfal, Eli (2005), Probability and computing: Randomized algorithms and probabilistic analysis, Cambridge University Press, pp. 107112, 308, ISBN 9780521835404