A Quick Introduction to Bloom Filters

Christian Grothoff
Department of Computer Sciences
Purdue University
christian@grothoff.org

A Bloom filter [1] is a compact probabilistic data structure which is used to determine if an element might be part of a set. The test may return true for elements that are not actually in the set (false-positives), but will never return false for elements that are in the set; for each element in the set the test must return true. Bloom filters are used in systems where many set tests must be performed and where putting the entire set in a location with fast access times is not feasible. A recent example which employs the use of Bloom filters is PlanetP [2], a peer-to-peer network in which peers exchange summary information about the content available at each peer in the form of Bloom filters. If a PlanetP peer is searching for content, it determines which of the peers advertise Bloom filters which have the correct bits set for the search key. Only these peers are then contacted over the network. This avoids searches of peers that are guaranteed not to have the desired content available while keeping the amount of bandwidth required for transmitting set information small; Bloom filters can be extraordinarily compact while still resulting in small numbers of false positive tests.

Bloom filters are implemented using a large bit vector with m bits. For an empty set, all bits of the bloom filter are unset. If the set E is non-empty, k bits are set for each element e in the set. The indices of these k bits are obtained using k hash functions $H_k : E \rightarrow \{0, \ldots, m-1\}$. Bits can be set multiple times. In order to test if an element e may be a member of the set, the k bits $H_k(e)$ are checked. If all are set, e is considered a potential member of the set and the Bloom filter test returns true, otherwise false. If only p percent of the bits in the entire Bloom filter are set and the H_k return optimally balanced random values, the probability that a test willreturn a false-positive result is p^k. If the Bloom filter needs to support the removal of elements from the set, a secondary datastructure that counts the number of collisions for each bit is used. These collision counters are not required for the actual test and can therefore be stored in less costly higher-latency storage. The length m of the bit vector can be tuned, trading space for fewer false-positive results.

We define b to be the number of bits in the bit vector of the Bloom filter per element in the set ($|E| \cdot b = m$). The optimal value for k can be computed for a given value of b since it is independent from the actual size of the set and only dependent on the ratio $\frac{m}{|E|}$. For the rest of the paper we will assume that b is known, constant and cannot be freely chosen. This assumption holds in particular for fixed sets (like dictionaries or other static databases) on devices where the available storage space is tightly bounded. Without this assumption, the Bloom filter may need to be recomputed when b changes significantly in order to obtain optimal false-positive rates.

This leaves the implementor concerned with the choice of k for a given b such that the number of false-positive set tests is minimized. Typically, k is chosen as a discrete value that is as close as possible to the non-discrete optimal value that would minimize the number of false-positive tests in the non-discrete setting. In other words, k is chosen by computing the optimal value for the non-discrete case and then rounding that value up or down. The rationale behind rounding to a non-discrete value is that it is obviously impossible to set a non-discrete number of bits k in the Bloom filter for a given element e.

Computation of k

Suppose k would not have to be chosen as a discrete value. In that case, the optimal choice for k is simply the value that minimizes the number of false-positive tests, $f_k(b)$:

$$f_k(b) := \left[1 - \left(1 - \frac{1}{n \cdot b}\right)^{k \cdot n}\right]^k. \tag{1}$$

The value of k that minimizes $f_k(b)$ is:

$$k(b) = b \cdot \ln 2. \tag{2}$$

For this value of $k(b)$, the number of false positives $f_k(b)$ would be $\left(\frac{1}{2}\right)^{k(b)}$. But since the actual number of hash functions must be discrete, $k(b)$ must be rounded up or down to obtain a useable discrete value for the number of hash functions. Determining which bound (lower or upper) is optimal is accomplished by inserting the value into formula (1). Note that while making a probabilistic choice between $\lfloor k \rfloor$ and $\lceil k \rceil$ could be used to achieve the optimal information density in the Bloom filter (by probabilistically setting $\frac{1}{2}$ of the bits), this improved approximation would not result in a better (smaller) value for f_k since the resulting f is just a linear combination of the two extremes $f_{\lfloor k \rfloor}$ and $f_{\lceil k \rceil}$ and can thus only be worse than the f_k for the best discrete value of k.

Acknowledgements

The author wishs to thank Igor Wronsky for pushing for the implementation of the Bloom filter in GNUnet and Krista Bennett for editing.

References

[1] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[2] Francisco Mathias Cuenca-Acuna, Richard P. Martin, and Thu D. Nguyen. Planetp: Using gossiping and random replication to support reliable peer-to-peer content search and retrieval. Technical Report DCS-TR-494, Rutgers University, 2002.