Scalable Bloom Filters ☆

Paulo Sérgio Almeida [a,*], Carlos Baquero [a,1], Nuno Preguiça [b], David Hutchison [c]

[a] *CCTC/DI, Universidade do Minho, Portugal*
[b] *CITI/DI, FCT, Universidade Nova de Lisboa, Portugal*
[c] *Computing Department, Lancaster University, UK*

Abstract

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. This paper proposes Scalable Bloom Filters, a variant of Bloom filters that can adapt dynamically to the number of elements stored, while assuring a maximum false positive probability.

© 2006 Elsevier B.V. All rights reserved.

Keywords: Data structures; Bloom filters; Distributed systems; Randomized algorithms

1. Introduction

Bloom filters [1] provide space-efficient storage of sets at the cost of a probability of false positive on membership queries. Insertion and membership testing in Bloom filters implies an amount of randomization, since elements are transformed using one-way hash functions. Testing for the presence of elements that have actually been inserted in the filter will always give a positive result; there are no false negatives. On the contrary, there is always some probability of false positives: elements that have not been inserted into the filter can erroneously pass the membership test.

An important property of Bloom filters is the linear relation between the filter size and the number of elements that can be stored. For any given maximum false positive probability, it is possible to determine how much filter state is needed per element [1,2]. As expected, lower false positive rates require more state per element.

If the maximum allowable error probability and the number of elements to store are both known, it is straightforward to dimension an appropriate filter. However, it is not always possible to know in advance how many elements will need to be stored; this leads to over-dimensioning the filters or relinquishing the maximum error probability.

In this paper we provide a solution for the case in which not only is the number of elements not known in advance but also we need to strictly enforce some maximum error probability. We prove that this is possible, by means of a novel construction: Scalable Bloom Filters (SBF).

After a brief review of related work, this paper is organized as follows. Section 3 reviews the basic mathematical properties of Bloom filters. Section 4 introduces Scalable Bloom Filters and gives an evaluation of their properties. Section 5 ends the paper with our conclusions.

2. Related work

In recent years, Bloom filters have received increased attention, and they are now being used in a large number of systems, including peer-to-peer systems [3,4], web caches [5], database systems [6] and others [7,2]. Several variants of the basic Bloom filter technique have been proposed in the literature.

In [5] the authors introduce the idea of a *counting Bloom filter*, allowing elements to be removed from the set represented by the Bloom filter; Spectral Bloom Filters [8] use a similar approach to store multi-sets; [9] proposes a multi-segment Bloom filter that allows efficient access when this data structure is stored on disk; a similar approach [10] is used in a network routing algorithm; Compressed Bloom Filters [11] improve performance when the Bloom filter is passed as a message, by using larger but sparser filters that lead to smaller compressed sizes; Bloomier filters [12] allow to efficiently associate values with a subset of the domain elements, by using sequences of pairs of Bloom filters.

All these variants suffer from the same limitation of the original Bloom filters: it is necessary to dimension, a priori, the size of the filters. We believe that it would be possible to drop this limitation for most (or even all) of these proposals by creating scalable variants along the lines of SBF.

3. Bloom filters

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 zeroes. 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 should output one index in M. When inserting an element e on the filter, the bits in the k indexes $h_1(e), h_2(e), \ldots, h_k(e)$ are set.

In particular, a filter with $M = 15$ bits and $k = 3$ hash functions could become as follows, after the insertion of one element:

0	0	1	0	0	0	0	0	0	0	1	0	1	0	0

The same procedure is used to insert other elements, each time setting the bits given by the corresponding k indexes.

In order to query a Bloom filter, say for element x, it suffices to verify if all bits in indexes $h_1(x), h_2(x), \ldots, h_k(x)$ are set. If one or more of these bits is not set, then the queried element is definitely not present on the filter. Otherwise, if all these bits are set, then the element is considered to be on the filter. Given this procedure, an error probability exists for positive matches, since the tested indexes might have been set by the insertion of other elements.

With the above setup, all hash functions are used to generate indexes over M. Since these hash functions are independent, nothing prevents collisions in the outputs. In the most extreme case we could have $h_1(x) = h_2(x) = \cdots = h_k(x)$. This means that in the general case each element will be described by 1 to k distinct indexes. Although for large values of M a collision seldom occurs, this aspect makes some elements more prone to false positives (and also complicates the analytical derivation of probabilities) [13].

A variant of Bloom filters [2], which we adopt in this paper, consists of partitioning the M bits among the k hash functions, thus creating k slices of $m = M/k$ bits. In this variant, each hash function $h_i()$, with $1 \leqslant i \leqslant k$, produces an index over m for its respective slice. Therefore, each element is always described by exactly k bits, which results in a more robust filter, with no element specially sensitive to false positives.

For $M = 15$ and $k = 3$ a filter would have 3 slices with 5 bits in each. After insertion of one element, the resulting configuration would have exactly one bit set in each slice. Each slice is depicted here in a column:

$$
\begin{array}{|c|c|c|}
\hline
0 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 0 \\
1 & 0 & 1 \\
0 & 0 & 0 \\
\hline
\end{array}
$$
$$\underbrace{}_{k}$$

3.1. False positives

False positives can occur when testing for the presence of a given element x, not present in the filter, and all k bits given by $h_i(x)$, $1 \leqslant i \leqslant k$, happen to be set

due to the insertion of other elements. Intuitively, if the number of slices k or the slice size m are increased the error probability will decrease.

The probability of a given bit being set in a slice is the fill ratio p between the number of set bits in the slice and the slice size m. For a large value m, this ratio will be approximately the same across all slices, and the false positive probability P for the filter will be

$$P = p^k.$$

In the example above, with one element inserted, p is $1/5$ and the overall error probability P is $(1/5)^3$, thus 0.8%.

In each slice, the probability that a given 0 bit becomes set after introducing one element is $1/m$; 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 $(1 - 1/m)^n$. Therefore, the probability that a specific bit in a slice is set after n insertions, which is also the expected fill ratio p, is

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

3.2. Bounding the error

From the analysis in the previous section, it is evident that the error probability P increases with n and decreases with m and k. We now determine how to choose k (and thus m) such that, for a given filter size M, we can maximize the number of stored elements n, while keeping the error probability below a certain value P.

For usable values of m, $1 - 1/m$ is almost the same as $e^{-1/m}$ (from the Taylor series expansion); we can use this approximation to obtain:

$$p \approx 1 - e^{-n/m},$$

from which we obtain

$$n \approx -m \ln(1 - p).$$

From $M = km$ and $P = p^k$ we obtain $m = M \ln p / \ln P$; therefore:

$$n \approx M \frac{\ln p \ln(1 - p)}{-\ln P}.$$

For any given error probability P and filter size M, n is maximized by making $p = 1/2$, regardless of P or M. As p corresponds to the fill ratio of a slice, a filter depicts an optimal use when slices are half full. With $p = 1/2$ we obtain

$$n \approx M \frac{(\ln 2)^2}{|\ln P|}.$$

Table 1
Several capacities for a bloom filter with 32 KB

P	0.1%	0.01%	0.001%	0.0001%
k	10	14	17	20
m	26 214	18 724	15 420	13 107
n	18 232	13 674	10 939	9116

In this expression it is clear that the number of elements n that can be stored, for a given error P, is linear on the filter size M. Finally, from $P = p^k$ and with $p = \frac{1}{2}$ we obtain

$$k = \log_2 \frac{1}{P}.$$

With these formulae it is now possible to determine the optimal filter parameters in order to respect a maximum error probability. For example, to have a maximum error of 0.1% we should have at least 10 slices, since $\log_2 \frac{1}{0.001} \approx 9.96$ ($2^{10} = 1024$). If this filter is allocated 32 kilobytes, each slice will have 26 214 bits and the filter is predicted to hold up to 18 232 elements. See Table 1.

4. Scalable Bloom Filters

A Scalable Bloom Filter addresses the problem of having to choose an a priori maximum size for the set, and allows an arbitrary growth of the set being represented. The two key ideas are:

– A SBF is made up of a series of one or more (plain) Bloom filters; when filters get full due to the limit on the fill ratio, a new one is added; querying is made by testing for the presence in each filter.
– Each successive bloom filter is created with a tighter maximum error probability on a geometric progression, so that the compounded probability over the whole series converges to some wanted value, even accounting for an infinite series.

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 $0 < r < 1$. At a given moment we will have l filters with error probabilities $P_0, P_0 r, P_0 r^2, \ldots, P_0 r^{l-1}$. The compounded error probability for the SBF will be:

$$P = 1 - \prod_{i=0}^{l-1} \left(1 - P_0 r^i\right).$$

We can use the known approximation

$$1 - \prod_i (1 - P_i) \leqslant \sum_i P_i,$$

to obtain an upper bound (which will be tight for small P_i):

$$P \leqslant \sum_{i=0}^{l-1} P_0 r^i \leqslant \lim_{l \to \infty} \sum_{i=0}^{l-1} P_0 r^i$$

and therefore

$$P \leqslant P_0 \frac{1}{1-r}.$$

The number of slices for each filter will be:

$$k_0 = \log_2 P_0^{-1}$$

and

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

To have each k_i as an integer, a natural choice will be $r = 1/2$, resulting in:

$$k_i = k_0 + i,$$

which means an extra slice per new filter. The compounded error probability for the SBF will be bounded by:

$$P \leqslant 2P_0 = 2^{1-k_0}.$$

Another possibility is to use an r other than $1/2$ and round up the resulting k_i's to obtain the number of slices. We will see below that choosing r around 0.8–0.9 will result in better average space usage for wide ranges of growth.

4.1. Scalable growth

The estimation of the set size that is to be stored in a filter may be wrong, possibly by several orders of magnitude. We may also want to use not much more memory than needed at a given time, and start a filter with a small size. Therefore, a SBF should be able to adapt to variations in size of several orders of magnitude in an efficient way.

When a new filter is added to a SBF, its size can be chosen orthogonally to the required false positive probability. A flexible growth can be obtained by making the filter sizes grow exponentially. We can have a SBF made up of a series of filters with slices having sizes $m_0, m_0 s, m_0 s^2, \ldots, m_0 s^{l-1}$.

Given that filters stop being used when the fill ratio reaches $1/2$, filter i will hold approximately:
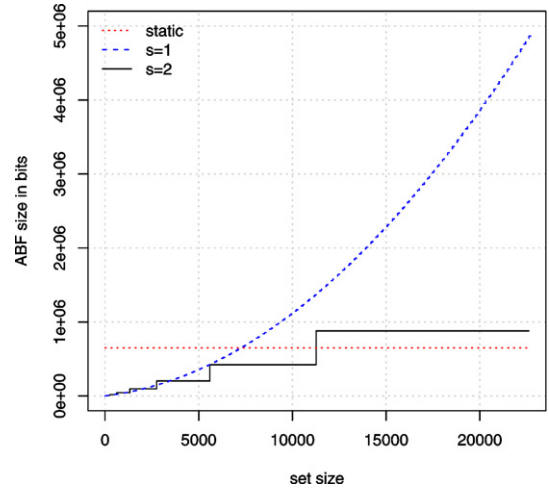
$$n_i \approx m_0 s^i \ln 2$$

Fig. 1. Space usage as a function of set size. Two SBFs, with slice growth factors $s = 1$ and $s = 2$, are compared with a static bloom. Both with $r = 0.5$, $m_0 = 128$ and $P = 10^{-6}$.

elements. The SBF with l stages will hold about

$$(\ln 2) m_0 \sum_{i=0}^{l-1} s^i$$

elements. This geometric progression allows a fast adaptation to set sizes of different orders of magnitude. A practical choice will be $s = 2$, which preserves m_i as a power of 2, if m_0 already starts as such; this is useful, as the range of a hash function is typically a power of 2.

In general, other values of s may be used. Fig. 1 shows the required size for the SBF as a function of set size, n, for $s = 1$ and $s = 2$. The case $s = 1$ gives a constant m in all stages; this case is not feasible as it would lead to much inefficiency, as the number of stages required grows linearly with set size, and in each stage an extra slice would be required (for $r = 1/2$); this would result in rapidly increasing space per element and computational cost for the hash functions. For $s = 2$ we can see that not only the number of stages remains low, as it increases logarithmically with the set size, but also the space required for the 22 624 element set is only slightly more than for a static filter dimensioned for that size.

To better understand adaptation to growth, we should not plot space usage against an absolute set size, but against the relative growth over the initial size. We should have a scale-free graph telling us how much space will be used according to the orders of magnitude in size the filter has to adapt to. Fig. 2 plots the space usage relative to a static filter dimensioned for the required size. Here we can see that if the set had to grow by 6 orders of magnitude, for $s = 2$ the SBF would use about twice the space of a static filter exactly dimensioned for
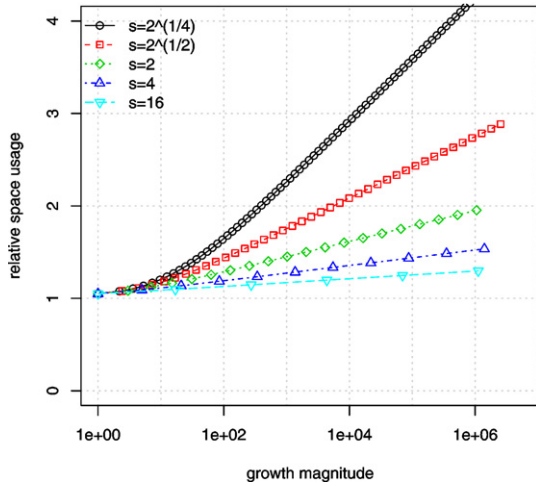
Fig. 2. Relative space usage with respect to a static filter as a function of set growth. With $r = 0.5$ and $P = 10^{-6}$.
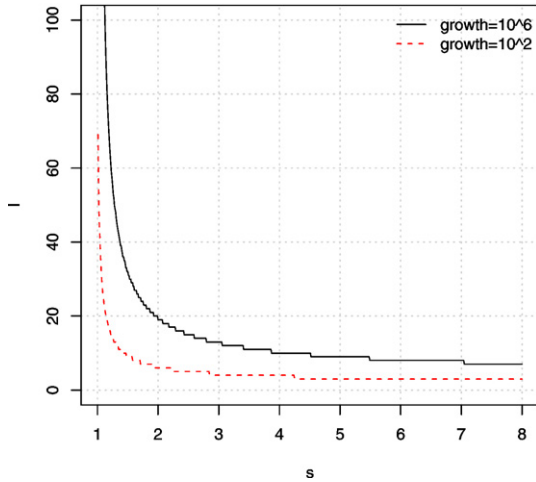

Fig. 3. Number of stages as a function of s.

the final size, and for $s = 4$ about 50% more space. In terms of space usage we can see that practical values of s like 2, 4 or above can be chosen, and values below 2 and approaching 1 will give progressively worse results.

Another aspect to consider in the choice of s is the number of stages required for the SBF. Fig. 3 plots the number of stages as a function of s, for two cases of set growth: 10^2 and 10^6. This figure confirms that s should not be chosen near 1 and that the practical choice of s as a power of two is a sensible one with this respect.

From these figures one could be led to think that the larger the s the better. However, as s tends to infinity, each successive stage of the SBF will take considerably more space which will remain poorly used for considerably more time until it gets full. A better criterion is to consider the average space usage over the lifetime of
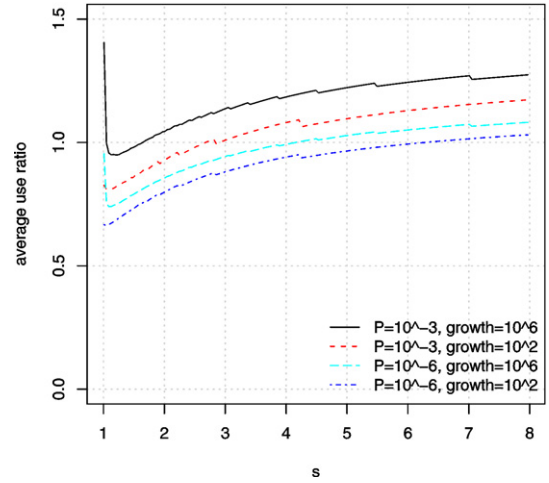

Fig. 4. Average relative space usage as a function of s, for different combinations of set growth and P, for optimal r.

the SBF from an empty set until the final set size. Fig. 4 plots this average space usage relative to a static filter (dimensioned for the final set size), as a function of s, for several combinations of error probability (10^{-3} and 10^{-6}) and set growth (10^2 and 10^6). These curves cover a wide range of scenarios; they show that, as long as s is not very close to 1, increasing s is not profitable.

Combining these two criteria, i.e., average space and number of stages, with the convenience of having a power of two, we can conclude that 2 or 4 will be a sensible choice for s. To keep the number of stages small, we can choose $s = 2$ if we expect a small set growth and $s = 4$ if we expect a larger growth.

4.2. Choosing the error probability ratio

The other parameter of a SBF that we need to choose is the error probability ratio r. We can choose values other than 0.5 and round up the resulting number of slices for stage i:

$$k_i = k_0 + i \log_2 r^{-1}.$$

Fig. 5 compares the space usage as a function of set growth for different combinations of P and r. It shows that if we use an r larger than 0.5, although we start by using more space (we need more initial slices, k_0, as P_0 needs to be smaller for the geometric series to converge to the same P), after some point we end up using less and less space as the set grows, as we add slices less frequently at each new stage. It specially pays to use a large r for a tighter error probability P, as the few extra slices needed initially will be a small overhead over the already large number of slices needed for $r = 0.5$.
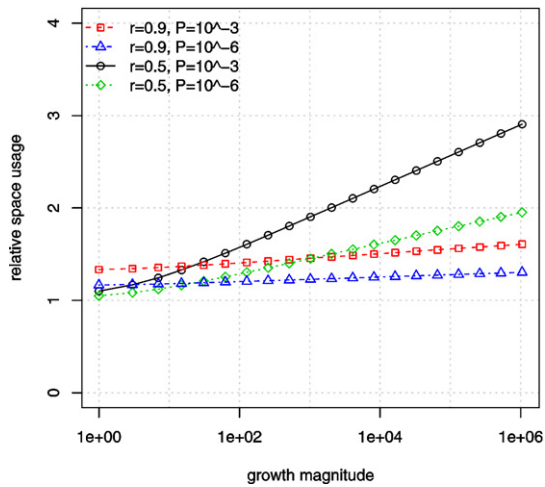
Fig. 5. Relative space usage as a function of growth, for different combinations of P and r and $s = 2$.
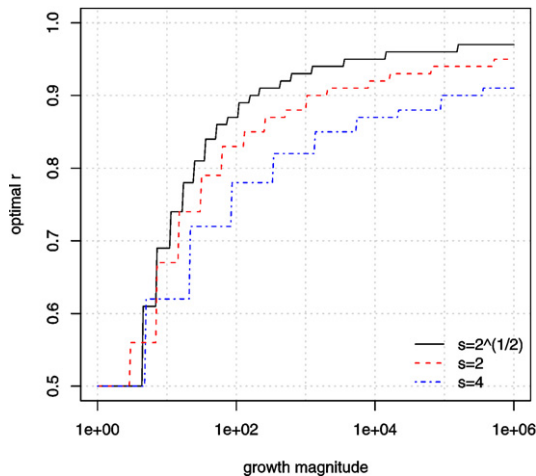

Fig. 6. Optimal r as a function of growth magnitude, for $s \in \{\sqrt{2}, 2, 4\}$ and $P = 10^{-6}$.

Fig. 4 shows average relative space usage, calculated for the optimal r that minimizes average space, for each combination of growth and s values (the optimal r does not depend on P).

In order to select an appropriate value for r we can observe how the optimal r behaves for different growth and s values. Fig. 6 shows the optimal r as a function of set growth, for three different values of s ($\sqrt{2}, 2, 4$). Considering the choice of $s = 2$ for small expected growth and $s = 4$ for larger growth, one can see that r around 0.8–0.9 is a sensible choice, that gives better space usage than the natural $r = 1/2$.

5. Conclusions

Bloom filters and the existing variants require a priori dimensioning of the maximum size of the set to be stored in the filter. Given that it is not always possible to know in advance how many elements will need to be stored, this leads to over-dimensioning the filters, possibly by several orders of magnitude.

In this paper we have introduced Scalable Bloom Filters (SBF), a mechanism that allows representing sets without having to know a priori the maximum set size and yet being able to choose from the start the maximum false positive probability. The mechanism adapts to set growth by using a series of classic Bloom filters of increasing sizes and tighter error probabilities, added as needed.

A SBF is parameterized not only by the initial size and error probability but also by the growth rate of the size and by the error probability tightening rate. In this paper we have studied the impact of these parameters on space usage and shown how they can be chosen for a range of scenarios.

References

[1] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Comm. ACM 13 (7) (1970) 422–426.

[2] F. Chang, W. Chang Feng, K. Li, Approximate caches for packet classification, in: Proc. of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2004), IEEE, 2004.

[3] P. Reynolds, A. Vahdat, Efficient peer-to-peer keyword searching, in: M. Endler, D.C. Schmidt (Eds.), Middleware, in: Lecture Notes in Computer Science, vol. 2672, Springer, Berlin, 2003, pp. 21–40.

[4] S.C. Rhea, J. Kubiatowicz, Probabilistic location and routing, in: Proc. of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002), 2002.

[5] L. Fan, P. Cao, J. Almeida, A.Z. Broder, Summary cache: A scalable wide-area web cache sharing protocol, IEEE/ACM Trans. Networks 8 (3) (2000) 281–293.

[6] L.F. Mackert, G.M. Lohman, R^* optimizer validation and performance evaluation for distributed queries, in: Proceedings of the Twelfth International Conference on Very Large Data Bases (VLDB'86), Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1986, pp. 149–159.

[7] A. Broder, M. Mitzenmacher, Network applications of Bloom filters: A survey, in: Proc. of Allerton Conference, 2002.

[8] S. Cohen, Y. Matias, Spectral Bloom filters, in: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03), ACM Press, New York, NY, USA, 2003, pp. 241–252.

[9] U. Manber, S. Wu, An algorithm for approximate membership checking with application to password security, Inform. Process. Lett. 50 (4) (1994) 191–197.

[10] S. Dharmapurikar, P. Krishnamurthy, D.E. Taylor, Longest prefix matching using Bloom filters, in: Proceedings of the 2003 Con-

ference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'03), ACM Press, New York, NY, USA, 2003, pp. 201–212.

[11] M. Mitzenmacher, Compressed Bloom filters, IEEE/ACM Trans. Networks 10 (5) (2002) 604–612.

[12] B. Chazelle, J. Kilian, R. Rubinfeld, A. Tal, The Bloomier filter: an efficient data structure for static support lookup tables,

in: SODA'04: Proceedings of the Fifteenth Annual ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2004, pp. 30–39.

[13] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, Y. Tang, On the false-positive rate of Bloom filters, Inform. Process. Lett. (2004). Available at http://citeseer.ist.psu.edu/649161.html.