Internet Mathematics Vol. 4, No. 1: 17-33

Using the Power of Two Choices to Improve Bloom Filters

Steve Lumetta and Michael Mitzenmacher

Abstract. We consider the combination of two ideas from the hashing literature: the power of two choices and Bloom filters. Specifically, we show via simulations that, in comparison with a standard Bloom filter, using the power of two choices can yield modest reductions in the false positive probability using the same amount of space and more hashing. While the improvements are sufficiently small that they may not be useful in most practical situations, the combination of ideas is instructive; in particular, it suggests that there may be ways to obtain improved results for Bloom filters while using the same basic approach they employ, as opposed to designing new, more complex data structures for the problem.

1. Introduction

A Bloom filter [Bloom 70] is a simple, space-efficient randomized data structure for representing a set in order to support membership queries that sometimes gives false positives. Bloom filters have proven useful, for example, in the context of routers, where off-chip memory accesses are dramatically slower than on-chip memory accesses and sets (e.g., lists of source-destination pairs, attack signatures, etc.) must be represented space-efficiently. Further applications and related variations are discussed extensively in the survey paper [Broder and Mitzenmacher 04]. While alternative data structures have recently appeared [Pagh et al. 05], the Bloom filter's simplicity, straightforward mapping to hardware, and excellent performance ensure that it will continue to be of great use in many applications. We briefly review Bloom filters in Section 2. For now, it

1542-7951/07 $0.50 per page

suffices to know that Bloom filters hash each element of the set to a vector of indices using a group of hash functions.

In this paper, we combine a Bloom filter with a seemingly unrelated but powerful technique often applied to improve load balancing, the power of two choices [Azar et al. 99, Mitzenmacher 96, Mitzenmacher et al. 01, Vöcking 99]. With this technique, an item is stored at the least loaded of two (or more) random alternatives. We briefly review the power of two choices in Section 2.

The standard analysis for deriving the optimal configuration of a Bloom filter assumes the use of a single group of hash functions to check for set membership. The power of two choices allows us to reduce the false positive probability while maintaining the form and the character of the original Bloom filter. The basic idea for combining the two techniques is to use two independent groups of hash functions for inserting elements into the Bloom filter. When checking for the membership of an element, both groups of hash functions are used, and a positive response is returned if either group indicates that the element is present. By allowing a choice of which hash function group is used to record the presence of an element in the set, we can reduce the false positive probability. Specifically, we show via simulations and numerical analysis that, in comparison with a standard Bloom filter, using the power of two choices can yield modest reductions in the false positive probability using the same amount of space and more hashing. As Bloom filters are often used in situations where a fixed amount of space is the primary constraint, if these gains were sufficiently large, they could prove useful in practice. While the gains we have found thus far do not yet suggest that this is the case, we find the fact that improvement occurs an interesting combinatorial curiosity that may suggest further ideas for improvement.

2. Background

2.1. Bloom Filter Review

We first review the concepts behind Bloom filters; for mathematical details, see the survey [Broder and Mitzenmacher 04]. A Bloom filter represents a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements from a universe U using an array of m bits, which we denote by $B[0], \ldots, B[m-1]$, initially all set to zero. The filter uses a group H of k independent hash functions, $h_1, \ldots, h_k$, all with the same range $\{0, \ldots, m-1\}$, where it is assumed that these hash functions independently map each element in the universe to a random number uniformly over the range. (While the randomness of the hash functions is clearly an optimistic assumption, it is standard and convenient for Bloom filter analyses.) For each element $x \in S$,

the bits $B[h_i(x)]$ are set to one for $1 \leq i \leq k$. (A bit can be set to one multiple times.) To answer a query of the form "Is $y \in S$?", we check whether all $h_i(y)$ are set to one. If not, y is not a member of S, by the construction. If all $h_i(y)$ are set to one, then the data structure answers that y is in S, and hence a Bloom filter may yield a *false positive*.

The probability of a false positive for an element not in the set, or the *false positive probability*, can be easily derived. If p is the fraction of ones in the filter, it is simply p^k. A standard combinatorial argument gives that p is concentrated around $1 - \mathrm{e}^{-kn/m}$, and the expression $\left(1 - \mathrm{e}^{-kn/m}\right)^k$ is minimized when $k = \ln 2 \cdot (m/n)$, giving a false positive probability f of

$$f = \left(1 - \mathrm{e}^{-kn/m}\right)^k = \left(\frac{1}{2}\right)^k \approx (0.6185)^{m/n}. \tag{2.1}$$

In practice, of course, k must be an integer. Also, in practical situations where Bloom filters are used, m/n (the number of bits per set element) and k should be thought of as constants. For example, when $m/n = 8$ and $k = 5$, the false positive probability is just over 0.02.

We emphasize an important intuition that guides some of our results. Note that as the number of hash functions k increases, there are more chances to find a zero when examining the Bloom filter, decreasing the chance of a false positive. On the other hand, larger values of k also produce more ones in the Bloom filter, increasing the chance of a false positive. The end result of this tradeoff yields the above optimization.

While the above analysis gives the optimal configuration for a standard Bloom filter, it is known that Bloom filters are not information-theoretically optimal [Broder and Mitzenmacher 04, Pagh et al. 05]. That is, it is possible to devise a data structure using less space that obtains the same false positive probability. It is therefore not surprising that we can improve on the Bloom filter construction; what is surprising is that we can improve on it while keeping essentially the same fundamental approach and the essential simplicity of a standard Bloom filter. While the improvement we have found is small, if more substantial improvements from similar methods could be found, it could be of significant practical importance. Bloom filters are used in many applications, and particularly in hardware applications, the simplicity of the Bloom filter approach is key for implementation.

An important variation of a Bloom filter is a *counting Bloom filter*, where each entry in the Bloom filter is not a single bit but a small counter that tracks the number of elements that have hashed to that location [Fan et al. 00]. A standard Bloom filter can be derived from a counting Bloom filter by setting all nonzero counts to one. Counting Bloom filters allow deletions; when an

element is deleted, the corresponding counters are decremented. Counters must be chosen large enough to avoid overflow; for most applications, four bits suffice [Fan et al. 00].

2.2. Power of Two Choices Review

The seminal example of the power of two choices stems from the results of Azar, Broder, Karlin and Upfal [Azar et al. 99]. It is well known that when n balls are sequentially placed uniformly at random into n bins, the fullest bin has with high probability $(1 + o(1)) \ln n / \ln \ln n$ balls in it. The result of [Azar et al. 99] is that if for each ball we instead choose two bins at random and sequentially place each ball into the one which is less full at the time of placement, then with high probability the fullest bin contains only $\ln \ln n / \ln 2 + O(1)$ balls. This idea of allowing choice to improve load balance has spawned a large literature; see, for example, [Mitzenmacher et al. 01].

A natural extension of this idea in the setting of Bloom filters is to allow each set element a choice of how to be recorded in the filter, by giving two (or more) groups of hash functions that can record an element. Here, the goal is not to balance the load but rather to minimize the number of bits set to one in the filter. The idea, however, follows the theme of providing better performance by introducing choices. We note that the conjunction of the power of choice and Bloom filters has appeared previously [Kumar and Crowley 05]. There the authors, in the context of a hash table construction, have choices in how to partition items into subtables of the hash table. A Bloom filter is used to summarize which subtable contains which item. Each subtable has an associated Bloom filter; the authors use the effect of the item placement on the false positive probabilities of the Bloom filters to guide their choice of subtable. They found experimentally that this approach greatly improved lookup performance, in that they could generally easily detect in which subtable the item appeared, with few errors caused by false positives on set items in other Bloom filters. Our work can be seen as an extrapolation of these ideas to the different problem of creating a single Bloom filter for a set.

3. Using Two Choices with Bloom Filters

We consider the following variation on a Bloom filter. We again have a set S of n elements and m bits available for a Bloom filter. Instead of one group of hash functions, we now have two groups: group H consists of $h_1, h_2, \ldots, h_k$ and group G consists of $g_1, g_2, \ldots, g_k$. Again, we treat all hash functions as being

perfectly independent random hash functions. Note that this requires $2k$ total hash functions, instead of k as for a standard Bloom filter.

Each element in the set S should have all bits from the hashes from either group G or group H set to one. That is, if we let $B[h_i(x)]$ be the Bloom filter bit given by hashing x with h_i, we have the following constraint: we require that for every element $x \in S$, either $B[h_i(x)] = 1$ for all $i \in [1, k]$ or $B[g_i(x)] = 1$ for all $i \in [1, k]$. (The *or* is not exclusive; both events may occur.) We say that for $x \in S$ either G must cover x or H must cover x, where cover has the natural meaning. A false positive occurs for an element $y \notin S$ if either $B[h_i(y)] = 1$ for all $i \in [1, k]$ or $B[g_i(y)] = 1$ for all $i \in [1, k]$ (or both); that is, either G or H covers y even though $y \notin S$. We address the question of how to choose which bits are actually set to one given the set S shortly. Although this idea extends naturally to more than two groups of hash functions, we focus on the case of two groups for ease of exposition.

Using two groups of hash functions would seem to increase the chances of a false positive, in that there are now two ways for a false positive to occur. Specifically, if p is the fraction of ones in the filter, the probability of a false positive is now $1-(1-p^k)^2 \approx 2p^k$. On the other hand, we now have an additional choice of which bits in the filter are set to one; if this choice sufficiently reduces the number of bits set to one in the filter, overall there may be a reduction in the false positive probability. This is exactly the tradeoff that we explore. The cost of this scheme is that additional hashing and lookups into the Bloom filter are now required; for any specific application, this cost must be considered against the improvement in the false positive probability. Whether this approach takes more time depends upon the setting; for example, hashes are often parallelized in hardware, in which case more hardware resources may be required instead of more time to compute more hashes and perform more lookups in parallel. Also, as previously stated, in many cases fixed space is the primary constraint when Bloom filters are used, as one is trying to fit the Bloom filter into a cache or fast memory. Finally, when we discuss our experiments, we will suggest a technique that greatly reduces the amount of hashing (but not the number of lookups) required. We assume henceforth that additional hashing and lookups are not a concern.

We consider two main settings. In the *online* setting, elements of S are presented one at a time, and after each element x_i appears, the Bloom filter must be updated so that either G or H covers x in the filter; moreover, a bit once set to the value one keeps the value one subsequently. This latter restriction is meant to encode the idea that we are not storing the elements of S, so we cannot naturally change which group is being used to fulfill the constraint for a previously seen element.

In the *offline* setting, all the elements of S are initially given, and we seek to minimize the number of ones in the filter while covering all elements of S. Obviously, this provides us with significantly more power in setting the bits of the Bloom filter. Ideally, we might find the optimal offline solution, which would minimize the number of ones in the filter, but this appears to be a hard problem, even in this setting, as we discuss in Section 3.2.

In both the online and offline settings, we find that reductions in the false positive probability are possible, using simple and efficient greedy algorithms to set up the filter. We emphasize that the additional work for our modified Bloom filter is essentially all in the construction phase; lookups are done in the same fashion, except that one must do more hashing and examine more bits. While the gains are small, the fact that gains are possible suggests further exploration of these techniques.

3.1. Online Solutions

In this section, we analyze the optimal online solution, using both simulations and numerical analysis.

In the online setting, since the false positive probability depends only on the number of ones in the filter, the natural greedy approach is to determine, when considering an element x, how many additional bits would have to be set to one to cover it by G and by H and then to modify the smallest number of bits from these two choices. We prove that this greedy process is stochastically optimal, in that it stochastically dominates any other strategy. That is, if X_t is the number of bits set to one by the greedy strategy after t elements are placed in the filter, and Y_t is the number of bits set to one by some other strategy (that is independent of future choices), then

$$Pr(X_t \geq w) \leq Pr(Y_t \geq w)$$

for any number w and any $1 \leq t \leq n$.

Theorem 3.1. *The greedy strategy stochastically dominates any other strategy.*

Proof. We need to show that

$$Pr(X_t \geq w) \leq Pr(Y_t \geq w),$$

where X_t is the number of bits set to one by the greedy strategy and Y_t is the number of bits set to one by an alternative strategy. We induct on t, with the statement being vacuously true at $t = 0$. Suppose that at time $t-1$, for all w, we have $Pr(X_{t-1} \geq w) \leq Pr(Y_{t-1} \geq w)$. We can then couple the configurations of

the bit array given by the greedy strategy with the configurations of the bit array given by the alternative strategy so that, in the coupling, the greedy configuration always has fewer bits set to one than the corresponding alternative configuration. Consider any such pair of configurations $C_{X,t-1}$ and $C_{Y,t-1}$ for the greedy and alternative strategies, respectively. We can then couple the random locations given by the hash functions at time t so that, as much as possible, whenever a bit from $C_{Y,t-1}$ is chosen that is already covered, a corresponding bit in $C_{X,t-1}$ is chosen that is already covered. With such a coupling, we are guaranteed that the resulting configuration $C_{X,t}$ from the greedy algorithm will have fewer bits set to one than the resulting configuration $C_{Y,t}$ for the alternative; it follows that

$$Pr(X_t \geq w) \leq Pr(Y_t \geq w)$$

as desired. □

Let X denote the final number of ones in the filter when the greedy strategy is used. Although we have not found a simple closed form for the expectation $E[X]$, a standard martingale argument shows that X is concentrated around $E[X]$ with high probability.

Theorem 3.2.

$$Pr(|X - E[X]| \geq \lambda) \leq 2\mathrm{e}^{-\lambda^2/2nk^2}.$$

Proof. Let Z_i represent the list of $2k$ locations given by the hashes for the ith element (ordered in any appropriate fashion, so that the locations for G and H are determined) to be greedily placed into the filter. Consider the Doob martingale given by

$$W_i = E[X \mid Z_1, Z_2, \ldots, Z_i].$$

Here, $W_0 = E[X]$, and $W_n = X$. It is clear that $|W_i - W_{i-1}| \leq k$. This can be formalized, for example, by a simple coupling argument. Consider the state of the filter after the ith element, which introduces at most k more ones into the array. Let us consider any two possible filters that may arise after the first i elements have entered. Without loss of generality, suppose that the first has a ones and the second has $b > a$ ones. We can always couple the choices of the remaining elements so that the first filter obtains at least as many ones at each step as the second filter under the online greedy algorithm (until some possible point where the number of ones becomes equal). It follows that the difference in expectations $|W_i - W_{i-1}|$ is at most k. Given this, the result follows by the standard form of the Azuma-Hoeffding inequality. □

A more refined analysis of $W_i - W_{i-1}$ along the lines of [Kamath et al. 95] could give even tighter bounds, but its point here is simply to show that concentration occurs.

Recall that X_t is the number of ones in the filter after t elements have been placed. For reasonable values, the distribution of X_t can be calculated iteratively, allowing a numerical computation of $E[X]$. Specifically, given a value for X_t, one can compute the distribution of the number of ones to be added; this allows one to compute the distribution of X_{t+1} from X_t.

A related approach that allows excellent approximations of the expectations with significantly less computation uses an approximate version of this iterative calculation. In what follows we for convenience assume that k is a constant independent of n, which is the standard case for a Bloom filter. Let $B_1(k,p)$ and $B_2(k,p)$ represent independent binomial random variables corresponding to k trials each with success probability p (and hence mean kp). A good approximation for the number of ones in the filter can be given by the equation

$$X_{t+1} = X_t + \min(B_1(k, 1 - X_t/m), B_2(k, 1 - X_t/m)). \tag{3.1}$$

The intuition for this equation is that the number of new one bits needed to cover an element for each family is given by k hashes, each of which gives a location that must be set to one with probability $1 - X_t/m$. This expression is only an approximation, however, since it is possible that two hash values $h_i(x)$ and $h_j(x)$ will be the same; hence, the number of ones needed to cover an element is not exactly a binomial, although this is correct up to $O(1/m)$ terms. Such smaller order terms have negligible effect, so this equation could also be used to approximately iteratively calculate the distribution of the number of ones after t elements appear.

This approximation is more useful, however, if we focus on the expected performance by letting $t' = t/n$, thereby rescaling time to run from 0 to 1 instead of from 0 to n, and letting $z(t') = X_t/m$. This yields a differential equation approximating the difference equation associated with the distribution X_t:

$$\frac{dz}{dt'} = \frac{E[\min(B_1(k, 1 - z), B_2(k, 1 - z)]}{m/n}. \tag{3.2}$$

We note that, using the theory of martingales, one can show that the evolution of X_t closely follows that of the differential equation, and Chernoff-like bounds can be given; see [Kurtz 70, Shwartz and Weiss 95, Wormald 95] for details. Using Theorem 1 of [Wormald 95], for example, gives the following result.

Theorem 3.3. *With probability* $1 - o(1)$, $X = X_n = mz(1) + o(m)$.

Proof. Note that the process X_t has bounded jumps as long as k is a constant, as $|X_{t+1} - X_t| \leq k$. Further, Equation (3.1) shows that

$$E[X_{t+1} - X_t \mid X_0, X_1, \ldots, X_t] = E[\min(B_1(k, 1 - X_t/m), B_2(k, 1 - X_t/m))].$$

The right-hand side is a continuous function of X_t/n and satisfies a natural Lipschitz condition as given in [Wormald 95]. The conditions of Theorem 1 of [Wormald 95] are therefore satisfied, and the correspondence to the differential equation and the theorem follow. □

Equation (3.2) does not appear to have a simple closed-form solution, although it can easily be numerically evaluated and used in place of or in conjunction with simulations. Notice that Equation (3.2) highlights the role played by the fraction m/n, the number of bits per set element, since the result of this differential equation depends only on m/n and not on m and n individually. Hence, up to asymptotically vanishing terms, the false positive probability is a function of m/n and k, as is Equation (2.1) for the standard Bloom filter analysis. Equation (3.2) generalizes in the natural way when the number of choices is greater than two. We compare the results from this equation to our simulations below. Also, note that the form of Equation (3.2) for one group of hash functions, namely

$$\frac{dz}{dt'} = \frac{E[(B_1(k, 1 - z)]}{m/n} = \frac{k(1 - z)}{m/n},$$

can be solved exactly to yield Equation (2.1), giving another derivation of this equation.

Our experimental results for the online setting are given in Table 1. We have considered cases with 8, 16, and 32 bits per set element, which are standard configurations. The case of $c = 1$ choice, corresponding to a standard Bloom filter, is compared with $c = 2$ choices and $c = 3$ choices. Recall that the total number of hashes required is then ck. The results presented are for the best value of k, the number of hash functions, over 1000 trials for each setting of parameters. We model our random hash functions by assigning each hash of an element a number generated using a standard 48-bit pseudorandom number generator.

For a small number of bits per set element ($m/n = 8$), we see no improvement. When $m/n = 16$, we see small improvement, and for $m/n = 32$ the false positive rate is reduced by a factor slightly less than two for $c = 2$ choices and by a factor slightly greater than two for $c = 3$ choices. In intuitive terms, we require many bits per set element before the greedy algorithm reduces the number of ones

n	m	c	k	Observed false positive probability
10,000	80,000	1	6	2.159×10^{-2}
10,000	80,000	2	7	2.323×10^{-2}
10,000	80,000	3	7	2.389×10^{-2}
10,000	160,000	1	11	4.588×10^{-4}
10,000	160,000	2	13	3.935×10^{-4}
10,000	160,000	3	13	3.607×10^{-4}
10,000	320,000	1	22	2.106×10^{-7}
10,000	320,000	2	24	1.285×10^{-7}
10,000	320,000	3	25	9.980×10^{-8}

Table 1. Average results for the greedy online algorithm from 1000 trials per experiment. Recall that n is the number of elements, m is the number of bits, c is the number of choices, and k is the number of hash functions. The value of k with the best observed false positive probability is used for each setting of n, m, and c. Note the case $c = 1$ corresponds to a standard Bloom filter.

in the filter sufficiently to make up for the two opportunities to obtain a false positive. Also, note that because fewer bits are set to one in the Bloom filter using this approach, the optimal results are obtained when using larger values of k than in a standard Bloom filter, and more generally the best k increases with the number of choices c.

We point out that the use of 10,000 elements does not affect our results; the numbers are nearly identical for 100,000 elements, as the key parameter is the ratio m/n. There is some variance; over 1000 trials with $m/n = 32$, $c = 2$, and $k = 24$, the observed false positive probability (calculated based on the number of ones in the filter) varied between 1.206×10^{-7} and 1.378×10^{-7}; this is similar to the variance for $m/n = 32$ for a standard Bloom filter, where the false positive probability ranged from 1.955×10^{-7} to 2.270×10^{-7}. Finally, our results are quite robust to small changes in the parameters. For example, changing the number of hash functions to another number near the optimal also leads to very small changes. For example, while the best choice of 13 hash functions when $m/n = 16$ gave an average false positive probability of 3.935×10^{-4} over 1000 trials, using 12 hash functions gave an average of 3.954×10^{-4}, and using 14 gave an average of 4.035×10^{-4}.

It is also worth noting that the differential Equation (3.2) is very accurate, as we show in Table 2. For the optimal configurations presented in Table 1, the predicted fraction of ones in the Bloom filter from Equation (3.2) matches the average from our experiments to essentially four decimal places (with one case off in the last digit due to rounding). We conclude that Equation (3.2) can be used to accurately predict performance.

n	m	c	k	Observed fraction of ones	Prediction of (3.2)
10,000	80,000	2	7	0.5296	0.5296
10,000	80,000	3	7	0.5020	0.5019
10,000	160,000	2	13	0.5187	0.5187
10,000	160,000	3	13	0.4994	0.4994
10,000	320,000	2	24	0.5016	0.5016
10,000	320,000	3	25	0.5022	0.5022

Table 2. Comparing the average results for the greedy online algorithm from 1000 trials per experiment with a numerical evaluation from the differential equations (to four decimal places).

As previously mentioned, the most glaring problem with this scheme is the number of hashes required, which grows as ck. We therefore suggest an improvement that substantially reduces the amount of hashing required. In [Kirsch and Mitzenmacher 05], it is shown that two hash functions can be used to mimic k hash functions with no change in the asymptotic false positive probability of a Bloom filter. The general approach is simple: for an element x, compute hashes $\alpha_1(x)$ and $\alpha_2(x)$ in the range $[0, m)$, and let $h_i(x) = \alpha_1(x) + (i-1)\alpha_2(x) \bmod m$ for $i \in [1, k]$. (In the case here where m is even, it makes sense to choose $\alpha_2(x)$ to be a random *odd* number, which we do in our experiments; see [Kirsch and Mitzenmacher 05] for more details on the approach.) If we use the same approach here for each group of hash functions, then while ck lookups are still required, now only $2c$ hashes are required, a significant reduction. Averaging the results of 1000 trials gives the same observed numerical results for the fraction of ones in the filter as in Table 2, to four decimal places and correspondingly roughly the same false positive probability.[1]

This is not surprising, given the results of [Kirsch and Mitzenmacher 05], which show that this choice of hash functions, asymptotically, has essentially the same behavior as random hash functions in the framework of Bloom filters. It would seem that we might hope to prove this, perhaps following the statement and proof of Theorem 3.3 regarding the limiting fraction of ones, but the dependence in the hash functions makes the approach less clear. We leave this as a problem for future work.

[1]As explained more carefully in [Kirsch and Mitzenmacher 05], the asymptotic false positive probability using just two hash functions to mimic k hash functions is the same as the asymptotic false positive probability using k hash functions. For very small false positive probabilities, such as when $m/n = 32$, the asymptotics may not be suitably accurate for small m and n, leading to a noticeable performance difference. The same occurs in this setting; for m/n up to 16, however, in our experiments the difference is generally very small once n is above 10,000, and the performance of our schemes using choice still appears better than a standard Bloom filter. For more on this phenomenon, see [Kirsch and Mitzenmacher 05].

3.2. Offline Solutions

In this section, we consider a greedy algorithm that extends our online solution to the offline case, and we examine its performance via simulation.

In the offline setting, one is given the entire set in advance, and the goal is to design a Bloom filter with the smallest possible false positive probability (equivalently, the smallest number of ones given m, n, k, and the groups of hash functions) while ensuring that all elements are covered.

In its general form, the problem of finding the smallest possible covering is at least as hard as that of finding a minimum vertex cover. To see this, consider an input graph $G = (V, E)$ for the minimum vertex cover problem. Associate the vertices of the graph with filter bits and the edges with set elements. In this correspondence, we have $k = 1$; the set elements each have two corresponding bits (the adjacent vertices), and at least one of them must be set to one (i.e., chosen to be in the vertex cover). The minimum vertex cover corresponds to a minimal number of filter bits being set to one, giving the smallest false positive probability for elements outside the set with random hashes. This reduction shows that the case where $k = 1$ is at least as hard as finding a minimum vertex cover and hence is NP-hard. In our setting, because groups correspond to *random* hash values, there might still be an efficient algorithm that works with high probability. At this point, however, we know of no efficient algorithm.

Even if we have no algorithm for the optimal solution, for comparison purposes it could also be useful to have a probabilistic argument giving insight into the distribution of the number of ones required. Denote the minimum possible number of ones in the final filter in the offline setting by the random variable X. As with the online setting, while we do not currently have an argument giving the expectation of X, we can say that X is concentrated around its expectation, using almost exactly the same martingale argument as in Theorem 3.2. (One can again show that if one sets

$$W_i = E[X \mid Z_1, Z_2, \ldots, Z_i],$$

then $|W_i - W_{i-1}| \leq k$.)

Although we do not have an optimal algorithm, there are several natural heuristic algorithms one can apply to this problem: hill-climbing, tabu search, simulated annealing, etc. For our experiments, we chose an extremely simple greedy algorithm; while more sophisticated approaches may perform better, for many applications a simple and quick approach is likely to be desirable. Our algorithm repeatedly walks over the elements of the set (in some fixed order) and re-optimizes the choice of group for each element in turn. In more detail, in the first pass, one applies the greedy online algorithm. Subsequently, each element

is associated with either G or H, which was the choice last used to cover the element. As we pass through the elements in turn, each element x is temporarily deleted from the filter, and then we greedily re-choose with which group to associate x (and set the filter bits accordingly). This choice is determined by which family requires fewer additional bits to be set to one to cover the element, given the current choices of the other elements. In this setting, we break ties randomly; we note that breaking ties randomly improves performance nontrivially over breaking ties in some fixed manner such as, for example, by the order of the groups.

The re-optimization step can easily be accomplished, for example, by using a counting Bloom filter to enable deletion and then simply deleting (after the first round of insertions) and re-inserting each element in turn. Although we present this approach as an offline algorithm, it could also be used as a background task to improve performance in the online setting if elements are stored as they arrive.

Our simple martinagle argument of Theorem 3.2 does not immediately extend to this case, as there are nontrivial interactions in the repeated re-optimization that are more difficult to deal with. Our experimental results below suggest that variance among trials is fairly small. Also, the differential equations approach does not apply for related reasons. Further analysis of the offline setting remains open.

Our experimental results for the offline setting are given in Table 3. We have again considered cases with 8, 16, and 32 bits per set element. In this table we also consider the number of rounds used in the offline greedy algorithm, where in each round each element of the set is (re-)inserted one time. The results presented are for the best value of k, the number of hash functions, over 1000 trials for each setting of parameters.

Because our greedy algorithm breaks ties randomly, it is not clear when it enters a local minimum from which no further improvement will occur. We found that with 10,000 elements and two choices, ten rounds appeared sufficient to get us very close to the minimum; five further rounds led to further decreases in the average false positive probability of around 1–2%. Similarly, about 30 rounds appear sufficient when $c = 3$. Similar results were obtained with experiments with 100,000 elements.

As one would expect, the gains are significantly greater than in the online case. Here, even when $m/n = 8$, the false positive probability is reduced from over 2% to near 1.5% using two choices, and below 1.25% using three choices. The false positive rate of 1.25% is better than what can be achieved using 9 bits per element in a standard Bloom filter, thus potentially giving about a 10% space improvement in a space-limited scenario. Similarly, two choices reduces the false positive probability by roughly a factor of two for 16 bits, and over a factor of

n	m	c	k	Rounds	Observed false positive probability
10,000	80,000	1	6	n/a	2.159×10^{-2}
10,000	80,000	2	7	10	1.505×10^{-2}
10,000	80,000	2	8	15	1.485×10^{-2}
10,000	80,000	3	8	30	1.237×10^{-2}
10,000	80,000	3	8	40	1.222×10^{-2}
10,000	160,000	1	11	n/a	4.588×10^{-4}
10,000	160,000	2	14	10	2.259×10^{-4}
10,000	160,000	2	14	15	2.223×10^{-4}
10,000	160,000	3	15	30	1.604×10^{-4}
10,000	160,000	3	15	40	1.582×10^{-4}
10,000	320,000	1	22	n/a	2.106×10^{-7}
10,000	320,000	2	26	10	6.260×10^{-8}
10,000	320,000	2	26	15	6.143×10^{-8}
10,000	320,000	3	27	30	3.579×10^{-8}
10,000	320,000	3	27	40	3.525×10^{-8}

Table 3. Average results for the greedy offline algorithm from 1000 trials per experiment. Again, the case $c = 1$ corresponds to a standard Bloom filter. Rounds refers to the number of times each element is inserted into the filter.

three for 32 bits. Going to three choices leads to further improvements; for 32 bits, the gain is roughly a factor of six.

Again, our results appear to scale: the numbers are similar for 100,000 elements, with the same number of rounds. There is some variance, but the greedy algorithm also appears to generally concentrate around its expectation over a sufficient number of rounds. For example, over 1000 trials with $m/n = 32$, $c = 2$, $k = 26$, and 15 rounds, the false positive probability varied between 5.706×10^{-8} and 6.486×10^{-8}. Also, as before, our results are quite robust to small changes in the parameters m/n and k, although here the number of rounds plays a key role in the overall performance.

Also, as with the online setting, in our experiments the fraction of ones in the filter differs little if we switch to using dependent hashes of the form $h_i(x) = \alpha_1(x) + (i-1)\alpha_2(x) \bmod m$ for $i \in [1, k]$. The false positive probability appears nearly the same. This hashing approach would likely be desirable if this variation of Bloom filters was used in practice.

4. Conclusion

We have demonstrated that, when constructing a Bloom filter, the power of two choices can yield improvements in the false positive probability using the

same amount of space, at the expense of more hashing. The advantage of this approach is that it keeps all of the appealing properties of a Bloom filter, such as its simplicity and ease of implementation, while yielding better results. It also provides an interesting and perhaps somewhat surprising example of the utility of the power of two choices in a different setting. The negatives of this approach are that it requires more hashing (although, as we have described, this can be ameliorated) and more bit lookups, and at this point the gains obtained are quite small.

This work suggests several additional open questions. First, an analysis of the optimal offline schemes, as well as an analysis of some easily implementable offline scheme, would be desirable, as they would allow design decisions without resorting to simulations. Second, a simple closed form (or very good approximation) for Equation (3.2) would be useful. Third, it would be intereseting to have a proof that using the techniques of [Kirsch and Mitzenmacher 05] to reduce hashing yields the same asymptotic performance. Fourth, it would be useful to extend these results to hold under simple implementable hash functions (as in [Dietzfelbinger and Woelfel 03]) instead of the random hash function model. Most importantly, this work opens the door to the possibility that there are further ways to improve the performance of Bloom filters through simple modifications that preserve their desirable properties. Such improvements would likely be useful in many implementations and are a worthy subject of further study.

We offer one final suggestion. Here, we have studied the case where the two groups of hash functions are completely independent. This may not be the right approach. Perhaps there is a natural way of making the hash functions dependent that would improve both the false positive probability and decrease the overall hashing requirements of using multiple groups. Developing and analyzing a useful dependence appears to be a challenging combinatorial problem.

Acknowledgements. The first author was supported in part by NSF grant ACI-9984492. The second author was supported in part by NSF grant CCR-0121154 and a grant from Cisco.

References

[Azar et al. 99] Y. Azar, A. Broder, A. Karlin, and E. Upfal. "Balanced Allocations." *SIAM Journal of Computing* 29:1 (1999), 180–200.

[Bloom 70] B. Bloom. "Space/Time Tradeoffs in Hash Coding with Allow-able Errors." *Communications of the ACM* 13:7 (1970), 422–426.

[Broder and Mitzenmacher 04] A. Broder and M. Mitzenmacher. "Network Applications of Bloom Filters: A Survey." *Internet Mathematics* 1:4 (2004), 485–509.

[Dietzfelbinger and Woelfel 03] M. Dietzfelbinger and P. Woelfel. "Almost Random Graphs with Simple Hash Functions." In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, pp. 629–638. New York: ACM Press, 2003.

[Fan et al. 00] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol." *IEEE/ACM Transactions on Networking* 8:3 (2000), 281–293.

[Kamath et al. 95] A. Kamath, R. Motwani, K. Palem, and P. Spirakis. "Tail Bounds for Occupancy and the Satisfiability Threshold Conjecture." *Random Structures and Algorithms* 7:1 (1995), 59–80.

[Kirsch and Mitzenmacher 05] A. Kirsch and M. Mitzenmacher. "Building a Better Bloom Filter." Harvard University Computer Science Technical Report TR-02-05, 2005. Available at ftp://ftp.deas.harvard.edu/techreports/tr-02-05.pdf.

[Kumar and Crowley 05] S. Kumar and P. Crowley. "Segmented Hash: An Efficient Hash Table Implementation for High-Performance Networking Subsystems." In *Proceedings of the 2005 Symposium on Architecture for Networking and Communications Systems*, pp. 91–103. New York: ACM Press, 2005.

[Kurtz 70] T. G. Kurtz. "Solutions of Ordinary Differential Equations as Limits of Pure Jump Markov Processes." *Journal of Applied Probability* 7 (1970), 49–58.

[Mitzenmacher 96] M. Mitzenmacher. "The Power of Two Choices in Randomized Load Balancing." PhD diss., University of California, Berkeley, 1996.

[Mitzenmacher et al. 01] M. Mitzenmacher, A. Richa, and R. Sitaraman. "The Power of Two Random Choices: A Survey of Techniques and Results." In *Handbook of Randomized Computing*, Vol. I, edited by P. Pardalos, S. Rajasekaran, J. Reif, and J. Rolim, pp. 255–312. Norwell, MA: Kluwer Academic Publishers, 2001.

[Pagh et al. 05] A. Pagh, R. Pagh, and S. Srinivas Rao. "An Optimal Bloom Filter Replacement." In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 823–829. Philadelphia: SIAM, 2005.

[Shwartz and Weiss 95] A. Shwartz and A. Weiss. *Large Deviations for Performance Analysis.* London: Chapman and Hall, 1995.

[Vöcking 99] B. Vöcking. "How Asymmetry Helps Load Balancing." In *Proceedings of the 40th Annual Sumposium on Foundations of Computer Science*, pp. 131–140. Los Alamitos, CA: IEEE Computer Society, 1999.

[Wormald 95] N. Wormald. "Differential Equations for Random Processes and Random Graphs." *Ann. Appl. Probab.* 5:4 (1995), 1217–1235.

Steve Lumetta, Department of Electrical and Computer Engineering, University of Illinois, 209 Coordinated Science Lab, 1308 West Main St., Urbana, IL 61801 (lumetta@uiuc.edu)

Michael Mitzenmacher, School of Engineering and Applied Sciences, Harvard University, 33 Oxford St., Cambridge, MA 02138 (michaelm@eecs.harvard.edu)

Received August 15, 2006; accepted January 31, 2007.