Fast Bloom Filters and Their Generalization

Yan Qiao, *Student Member*, *IEEE*, Tao Li, and Shigang Chen, *Senior Member*, *IEEE*

Abstract—Bloom filters have been extensively applied in many network functions. Their *performance* is judged by three criteria: query overhead, space requirement, and false positive ratio. Due to wide applicability, any improvement to the performance of Bloom filters can potentially have a broad impact in many areas of networking research. In this paper, we study Bloom-1, a data structure that performs membership check in one memory access, which compares favorably with the k memory accesses of a standard Bloom filter. We also generalize Bloom-1 to Bloom-g and Bloom-α, allowing performance tradeoff between membership query overhead and false positive ratio. We thoroughly examine the variants in this family of filters, and show that they can be configured to outperform the Bloom filters with a smaller number of memory accesses, a smaller or equal number of hash bits, and a smaller or comparable false positive ratio in practical scenarios. We also perform experiments based on a real traffic trace to support our filter design.

Index Terms—Bloom filter, memory access, false positive, hash requirement

1 Introduction

Bloom filters are compact data structures for high-speed online membership check against large data sets [2], [3]. They have wide applications [4] in routing-table lookup [5], [6], [7], online traffic measurement [8], [9], peer-to-peer systems [10], [11], cooperative caching [12], firewall design [13], intrusion detection [14], bioinformatics [15], database query processing [16], [17], stream computing [18], and distributed storage systems [19]. Many network functions require membership check. A firewall may be configured with a large watch list of addresses that are collected by an intrusion detection system. If the requirement is to log all packets from those addresses, the firewall must check each arrival packet to see if the source address is a member of the list. Another example is routing-table lookup. The lengths of the prefixes in a routing table range from 8 to 32. A router can extract 25 prefixes of different lengths from the destination address of an incoming packet, and it needs to determine which prefixes are in the routing tables [5]. Some traffic measurement functions require the router to collect the flow labels [9], [20], such as source/destination address pairs or address/port tuples that identify TCP flows. Each flow label should be collected only once. When a new packet arrives, the router must check whether the flow label extracted from the packet belongs to the set that has already been collected before. As a last example for the membership check problem, we consider the context-based access control (CBAC) function in Cisco routers [21]. When a router receives a packet, it may want to first determine whether the addresses/ports in the packet have a matching entry in the CBAC table before performing the CBAC lookup.

In all of the previous examples, we face the same fundamental problem: For a large data set, which may be an address list, an address prefix table, a flow label set, a CBAC table, or other types of data, we want to check whether a given element belongs to this set or not. If there is no performance requirement, this problem can be easily solved using textbook data structures such as binary search [22] (which stores the set in a sorted array and uses binary search for membership check), or a traditional hash table [23] (which uses linked lists to resolve hash collision). However, these approaches are inadequate if there are stringent speed and memory requirements.

Modern high-end routers and firewalls implement their per-packet operations mostly in hardware. They are able to forward each packet in a couple of clock cycles. To keep up with such high throughput, many network functions that involve per-packet processing also have to be implemented in hardware. However, they cannot store the data structures for membership check in DRAM because the bandwidth and delay of DRAM access cannot match the packet throughput at the line speed. Consequently, the recent research trend is to implement membership check in the high-speed on-die cache memory, which is typically SRAM. The SRAM is, however, small and must be shared among many online functions. This prevents us from storing a large data set directly in the form of a sorted array or a hash table. A Bloom filter [2] is a bit array that encodes the membership of data elements in a set. Each member in the set is hashed to k bits in the array at random locations, and these bits are set to ones. To query for the membership of a given element, we also hash it to k bits in the array and see if these bits are all ones.

The *performance* of the Bloom filter and its many variants is judged based on three criteria: The first one is the *query overhead*, including the number of memory accesses and the number of hash operations for each membership query. The overhead limits the highest throughput that the filter can support. Because both SRAM and the hash function circuit may be shared among different network functions, it is important for them to minimize their query overhead to achieve good system performance. In the rest of the paper, when we refer to *overhead*, we always mean the query overhead. The second performance criterion is the *space requirement*. Minimizing the space requirement to encode

• *The authors are with the Department of Computer & Information Science & Engineering, University of Florida, Gainesville, FL 32611.*

Manuscript received 23 June 2012; revised 29 Nov. 2012; accepted 4 Jan. 2013; published online 15 Feb. 2013.
Recommended for acceptance by M. Guo.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2012-06-0591.
Digital Object Identifier no. 10.1109/TPDS.2013.46.

each member allows a network function to fit a large set in the limited SRAM space for membership check. The third criterion is the *false positive ratio*. A Bloom filter may mistakenly claim a nonmember to be a member due to its lossy encoding method. There is a tradeoff between the space requirement and the false positive ratio. We can reduce the latter by allocating more memory.

Given the fact that Bloom filters have been applied so extensively in the network research, any improvement to their performance can potentially have a broad impact. In this paper, we study a data structure, called *Bloom-1*, which makes just one memory access to perform membership check, comparing with $k(>1)$ memory accesses of the standard Bloom filter. We point out that, due to its high overhead, the traditional Bloom filter is not practical when the optimal value of k is used to achieve a low false positive ratio. We generalize Bloom-1 to Bloom-g, which allows g memory accesses. We show that they can achieve the low false positive ratio of the Bloom filter with optimal k, without incurring the same kind of high overhead. We further generalize Bloom-1 to Bloom-α, which achieves better false positive ratio with small increase in overhead. We perform a thorough analysis to reveal the properties of this family of filters. We discuss how they can be applied for static or dynamic data sets. We also conduct experiments based on a real traffic trace to study the performance of the new filters.

The rest of the paper is organized as follows: Section 2 presents the Bloom-1 filter that makes one memory access per membership query. Section 3 generalizes Bloom-1 to Bloom-g, which allows more than one memory access. Section 4 presents another generalization of Bloom-1, reducing false positive ratio considerably with a slight increase in memory access overhead. Section 5 draws the conclusion. The related work [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44] and the experimental results can be found in the supplemental file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.46.

2 Bloom-1: One Memory Access Bloom Filter

2.1 Bloom Filter

A Bloom filter is a space-efficient data structure for membership check. It includes an array B of m bits, which are initialized to zeros. The array stores the membership information of a set as follows: Each member e of the set is mapped to k bits that are randomly selected from B through k different hash functions, $H_i(e)$, $1 \le i \le k$, whose range is $[0, m-1)$. Some lightweight hash functions used in Bloom filters can be found in [24], [25]. To encode the membership information of e, the bits, $B[H_1(e)], \ldots, B[H_k(e)]$, are set to ones. These are called the *membership bits* in B for the element e. Some frequently used notations in this paper can be found in Table 1.

To check the membership of an arbitrary element e', if the k bits, $B[H_i(e')]$, $1 \le i \le k$, are all ones, e' is considered to be a member of the set. Otherwise, it is not a member.

We can treat the k hash functions logically as a single one that produces $k \log_2 m$ hash bits. For example, suppose

TABLE 1
Notations

n	number of members in a set
B or $B1$	bit array
m	number of bits in the bit array B or $B1$
k	number of membership bits for each element
h	number of hash bits for locating all membership bits
k^*	the optimal value of k that minimizes the false positive ratio of a Bloom filter
$k1^*$	the optimal value of k that minimizes the false positive ratio of a Bloom-1 filter
f_B, f_{B1}, f_{Bg}	false positive ratios of a Bloom filter, a Bloom-1 filter, and a Bloom-g filter, respectively
l	number of words in a bit array
w	number of bits in a word, $m = l \times w$
$B(k=3)$	Bloom filter that uses three bits to encode each member
$B1(k=3)$	Bloom-1 filter that uses three bits to encode each member
$B1(h = 3\log_2 m)$	Bloom-1 filter that uses up to $3\log_2 m$ hash bits
$B(\text{optimal } k)$	Bloom filter that uses the optimal number k^* of bits to encode each member to minimize its false positive ratio
$B1(\text{optimal } k)$	Bloom-1 filter that uses the optimal number $k1^*$ of bits to encode each member to minimize its false positive ratio

m is 2^{20}, $k=3$, and a hash routine outputs 64 bits. We can extract three 20-bit segments from the first 60 bits of a single hash output and use them to locate three bits in B. Hence, from now on, instead of specifying the number of hash functions required by a filter, we will state *the number of hash bits* that are needed, which is denoted as h.

A Bloom filter does not have *false negatives*, meaning that if it answers that an element is not in the set, it is truly not in the set. The filter, however, has *false positives*, meaning that if it answers that an element is in the set, it may not be really in the set. According to [2], [3], the false positive ratio f_B, which is the probability of mistakenly treating a nonmember as a member, is

$$f_B = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k \approx \left(1 - e^{-\frac{nk}{m}}\right)^k, \quad (1)$$

where n is the number of members in the set. Obviously, the false positive ratio decreases as m increases, and increases as n increases. The optimal value of k (denoted as k^*) that minimizes the false positive ratio can be derived by taking the first-order derivative on (1) with respect to k, then letting the right side be zero, and solving the equation. The result is

$$k^* = \ln 2 \times m/n \approx 0.7m/n. \quad (2)$$

The optimal k sometimes can be very large. To avoid too many memory accesses, we may also set k as a small constant in practice.

2.2 Bloom-1 Filter

To check the membership of an element, a Bloom filter requires k memory accesses. We introduce the *Bloom-1 filter*, which requires one memory access for membership check. The basic idea is that instead of mapping an element to k bits randomly selected from the entire bit array, we map it to

k bits in a word that is randomly selected from the bit array. A word is defined as a continuous block of bits that can be fetched from the memory to the processor in one memory access. In today's computer architectures, most general-purpose processors fetch words of 32 or 64 bits. Specifically, designed hardware may access words of 72 bits or longer.

A Bloom-1 filter is an array $B1$ of l words, each of which is w bits long. The total number m of bits is $l \times w$. To encode a member e during the filter setup, we first obtain a number of hash bits from e, and use $\log_2 l$ hash bits to map e to a word in $B1$. It is called the *membership word* of e in the Bloom-1 filter. We then use $k \log_2 w$ hash bits to further map e to k membership bits in the word and set them to ones. The total number of hash bits that are needed is $\log_2 l + k \log_2 w$. Suppose $m = 2^{20}$, $k = 3$, $w = 2^6$, and $l = 2^{14}$. Only 32 hash bits are needed, smaller than the 60 hash bits required in the previous Bloom filter example under similar parameters.

To check if an element e' is a member in the set that is encoded in a Bloom-1 filter, we first perform hash operations on e' to obtain $\log_2 l + k \log_2 w$ hash bits. We use $\log_2 l$ bits to locate its membership word in $B1$, and then use $k \log_2 w$ bits to identify the membership bits in the word. If all membership bits are ones, it is considered to be a member. Otherwise, it is not.

The change from using k random bits in the array to using k random bits in a word may appear simple, but it is also fundamental. An important question is how it will affect the false positive ratio and the query overhead. A more interesting question is how it will open up new design space to configure various new filters with different performance properties. This is what we will investigate in depth.

The false negative ratio of a Bloom-1 filter is also zero. The false positive ratio f_{B1} of Bloom-1, which is the probability of mistakenly treating a nonmember as a member, is derived as follows: Let F be the false positive event that a nonmember e' is mistaken for a member. The element e' is hashed to a membership word. Let X be the random variable for the number of members that are mapped to the same membership word. Let x be a constant in the range of $[0, n]$, where n is the number of members in the set. Assume we use fully random hash functions. When $X = x$, the conditional probability for F to occur is

$$Prob\{F \mid X = x\} = \left(1 - \left(1 - \frac{1}{w}\right)^{xk}\right)^k. \quad (3)$$

Obviously, X follows the binomial distribution, $Bino(n, \frac{1}{l})$, because each of the n elements may be mapped to any of the l words with equal probabilities. Hence,

$$Prob\{X = x\} = \binom{n}{x}\left(\frac{1}{l}\right)^x\left(1 - \frac{1}{l}\right)^{n-x}, \ \forall\ 0 \le x \le n. \quad (4)$$

Therefore, the false positive ratio can be written as

$$f_{B1} = Prob\{F\} = \sum_{x=0}^{n}(Prob\{X = x\} \cdot Prob\{F \mid X = x\})$$
$$= \sum_{x=0}^{n}\left(\binom{n}{x}\left(\frac{1}{l}\right)^x\left(1 - \frac{1}{l}\right)^{n-x}\left(1 - \left(1 - \frac{1}{w}\right)^{xk}\right)^k\right). \quad (5)$$

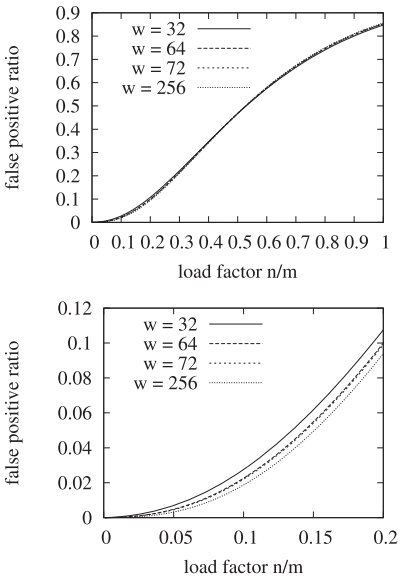


Fig. 1. *Upper Plot:* False positive ratios for Bloom-1 under different word sizes. *Lower Plot:* Magnified false positive ratios for Bloom-1 under different word sizes.

2.3 Impact of Word Size

We first investigate the impact of word size w on the false positive ratio of a Bloom-1 filter. If n, l, and k are known, we can obtain the optimal word size that minimizes (5). However, in reality, we can only decide the amount of memory (i.e., m) to be used for a filter, but cannot choose the word size once the hardware is installed. In the upper plot of Fig. 1, we compute the false positive ratios of Bloom-1 under four word sizes: 32, 64, 72, and 256 bits, when the total amount of memory is fixed at $m = 2^{20}$ and k is set to 3. Note that the number of words, $l = \frac{m}{w}$, is inversely proportional to the word size.

The horizontal axis in the figure is the *load factor*, $\frac{n}{m}$, which is the number of members stored by the filter divided by the number of bits in the filter. Since most applications require relatively small false positive ratios, we zoom in at the load-factor range of [0, 0.2] for a detailed look in the lower plot of Fig. 1. The computation results based on (5) show that a larger word size helps to reduce the false positive ratio. In that case, we should simply set $w = m$ for the lowest false positive ratio. However, in practice, w is given by the hardware, not a configurable parameter. Without losing generality, we choose $w = 64$ in our computations and simulations for the rest of the paper.

2.4 Bloom with $k = 3$ versus Bloom-1

Although the optimal k always yields the best false positive ratio of the Bloom filter, a small value of k is sometimes preferred to bound the query overhead in terms of memory accesses and hash operations. We compare the performance of the Bloom-1 filter and the Bloom filter in both scenarios. In this section, we use the Bloom filter with $k = 3$ as the benchmark.

We compare the performance of three types of filters: 1) $B(k = 3)$, which represents a Bloom filter that uses 3 bits to encode each member; 2) $B1(k = 3)$, which represents a Bloom-1 filter that uses 3 bits to encode each member;

TABLE 2
Query Overhead Comparison of Bloom-1 Filters and Bloom Filter with $k = 3$ and $w = 64$

Data Structure	# Memory Access	# Hash Bits		
		$m = 2^{16}$	$m = 2^{20}$	$m = 2^{24}$
$B(k = 3)$	3	48	60	72
$B1(k = 3)$	1	28	32	36
$B1(h = 3\log_2 m)$	1	32-44	40-58	48-72

Note that $B(k = 3)$ uses $3\log_2 m$ hash bits and $B1(h = 3\log_2 m)$ may use fewer than that number.

3) $B1(h = 3\log_2 m)$, which represents a Bloom-1 filter that uses the same number of hash bits as $B(k = 3)$ does.

For $B1(h = 3\log_2 m)$, we are allowed to use $3\log_2 m$ hash bits, which can encode up to $\frac{3\log_2 m - \log_2 l}{\log_2 w}$ membership bits in the filter, where $\log_2 l$ hash bits are used to locate the membership word and $\log_2 w$ hash bits are used to locate each membership bit in the word. However, it is not necessary to use more than the optimal number k of membership bits that minimizes the false positive ratio of the Bloom-1 filter in (5). Let $k1^*$ be the optimal value number k. Hence, $B1(h = 3\log_2 m)$ actually uses $k1' = \min\{k1^*, \frac{3\log_2 m - \log_2 l}{\log_2 w}\}$ membership bits to encode each member. The number of hash bits to locate them is, therefore, $\log_2 l + k1' \log_2 w$, which may be smaller than $3\log_2 m$.

Table 2 presents numerical results of *the number of memory accesses* and *the number of hash bits* needed by the three filters for each membership query. First, we compare $B(k = 3)$ and $B1(k = 3)$. The Bloom-1 filter saves not only memory accesses but also hash bits. When the hash routine is implemented in hardware (such as CRC [24]), the memory access may become the performance bottleneck, particularly when the filter's bit array is located off-chip. In this case, the query throughput of $B1(k = 3)$ can be up to three times of the throughput of $B(k = 3)$.

Next, we consider $B1(h = 3\log_2 m)$. Even though it still makes one memory access to fetch a word, the processor may check more than 3 bits in the word for a membership query. If the operations of hashing, accessing memory, and checking membership bits are pipelined and the memory access is the performance bottleneck, the throughput of $B1(h = 3\log_2 m)$ will also be three times of the throughput of $B(k = 3)$.

Finally, we compare the false positive ratios of the three filters in Fig. 2. The figure shows that the false positive ratio of $B1(k = 3)$ is slightly worse than that of $B(k = 3)$. The reason is that concentrating the membership bits in one word reduces the randomness. $B1(h = 3\log_2 m)$ is better than $B(k = 3)$ when the load factor is smaller than 0.1. This is because the Bloom-1 filter requires a fewer number of hash bits on average to locate each membership bit than the Bloom filter does. Therefore, when available hash bits are the same, $B1(h = 3\log_2 m)$ is able to use a larger k than $B(k = 3)$, as shown in Fig. 3.

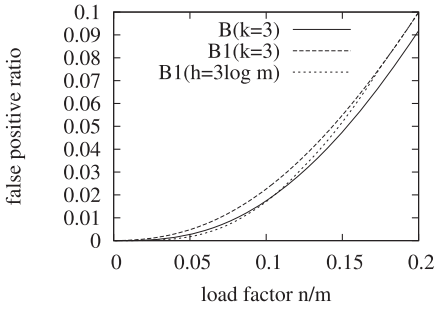


Fig. 2. Performance comparison in terms of false positive ratio. Parameters: $w = 64$ and $m = 2^{20}$.

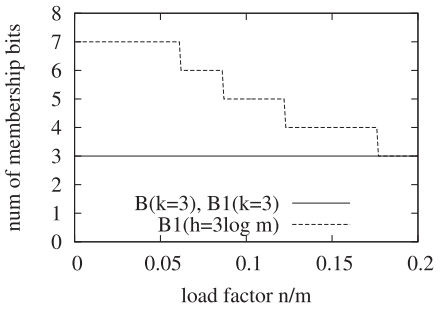


Fig. 3. Number of membership bits used by the filters.

2.5 Bloom with Optimal k versus Bloom-1 with Optimal k

We can reduce the false positive ratio of a Bloom filter or a Bloom-1 filter by choosing the optimal number of membership bits. From (2), we find the optimal value k^* that minimizes the false positive ratio of a Bloom filter. From (5), we can find the optimal value $k1^*$ that minimizes the false positive ratio of a Bloom-1 filter. The values of k^* and $k1^*$ with respect to the load factor are shown in Fig. 4. When the load factor is less than 0.1, $k1^*$ is significantly smaller than k^*.

We use $B(\text{optimal } k)$ to denote a Bloom filter that uses the optimal number k^* of membership bits, and $B1(\text{optimal } k)$ to denote a Bloom-1 filter that uses the optimal number $k1^*$ of membership bits.

To make the comparison more concrete, we present the numerical results of memory access overhead and hashing overhead with respect to the load factor in Table 3. For example, when the load factor is 0.04, the Bloom filter requires 17 memory accesses and 340 hash bits to minimize its false positive ratio, whereas the Bloom-1 filter requires only one memory access and 62 hash bits. In practice, the load factor is determined by the application requirement on the false positive ratio. If an application requires a very small false positive ratio, it has to choose a small load factor.

Next, we compare the false positive ratios of $B(\text{optimal } k)$ and $B1(\text{optimal } k)$ with respect to the load factor in Fig. 5.

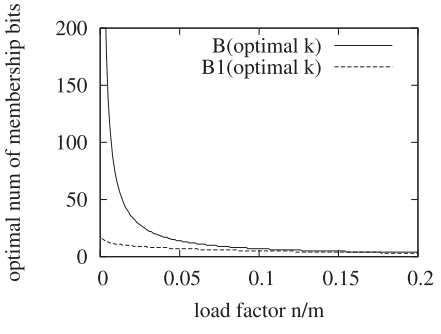


Fig. 4. Optimal number of membership bits with respect to the load factor. Parameters: $m = 2^{20}$ and $w = 64$.

TABLE 3
Query Overhead Comparison of Bloom-1 Filter and Bloom Filter with Optimal Number of Membership Bits

a. Number of memory accesses per query

	Load Factor n/m				
	0.01	0.02	0.04	0.08	0.16
B(optimal k)	69	35	17	9	4
$B1$(optimal k)	1	1	1	1	1

b. Number of hash bits per query

	Load Factor n/m				
	0.01	0.02	0.04	0.08	0.16
B(optimal k)	1380	700	340	180	80
$B1$(optimal k)	80	74	62	50	38

Parameters: $m = 2^{20}$ *and* $w = 64$.

The Bloom filter has a much lower false positive ratio than the Bloom-1 filter. On one hand, we must recognize the fact that, as shown in Table 3, the overhead for the Bloom filter to achieve its low false positive ratio is simply too high to be practical. On the other hand, it raises a challenge for us to improve the design of the Bloom-1 filter so that it can match the false positive ratio of the Bloom filter at much lower overhead. In the next section, we generalize the Bloom-1 filter to allow performance-overhead tradeoff, which provides flexibility for practitioners to achieve a lower false positive ratio at the expense of modestly higher query overhead.

3 Bloom-g: A Generalization of Bloom-1

3.1 Bloom-g Filter

As a generalization of Bloom-1 filter, a *Bloom-g filter* maps each member e to g words instead of one, and spreads its k membership bits evenly in the g words. More specifically, we use $g \log_2 l$ hash bits derived from e to locate g membership words, and then use $k \log_2 w$ hash bits to locate k membership bits. The first one or multiple words are each assigned $\lceil \frac{k}{g} \rceil$ membership bits, and the remaining words are each assigned $\lfloor \frac{k}{g} \rfloor$ bits, so that the total number of membership bits is k.

To check the membership of an element e', we have to access g words. Hence, the query overhead includes g memory accesses and $g \log_2 l + k \log_2 w$ hash bits.

The false negative ratio of a Bloom-g filter is zero and the false positive ratio f_{Bg} of the Bloom-g filter is derived as follows: Each member encoded in the filter randomly selects g membership words. There are n members. Together they select gn membership words (with replacement). These words are called the *encoded words*. In each encoded word, $\frac{k}{g}$ bits are randomly selected to be set as ones during the filter setup. To simplify the analysis, we use $\frac{k}{g}$ instead of taking the ceiling or floor.

Now consider an arbitrary word D in the array. Let X be the number of times this word is selected as an encoded word during the filter setup. Assume we use fully random hash functions. When any member randomly selects a word to encode its membership, the word D has a probability of $\frac{1}{l}$ to be selected. Hence, X is a random number that follows the binomial distribution $Bino(gn, \frac{1}{l})$. Let x be a constant in the range $[0, gn]$:

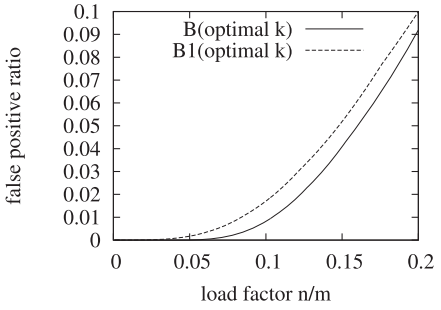


Fig. 5. False positive ratios of the Bloom filter and the Bloom-1 filter with optimal k. $m = 2^{20}$ and $w = 64$.

$$Prob\{X = x\} = \binom{gn}{x} \left(\frac{1}{l}\right)^x \left(1 - \frac{1}{l}\right)^{gn-x}. \tag{6}$$

Consider an arbitrary nonmember e'. It is hashed to g membership words. A false positive happens when its membership bits in each of the g words are ones. Consider an arbitrary membership word of e'. Let F be the event that the $\frac{k}{g}$ membership bits of e' in this word are all ones. Suppose this word is selected for x times as an encoded word during the filter setup. We have the following conditional probability:

$$Prob\{F \mid X = x\} = \left(1 - \left(1 - \frac{1}{w}\right)^{x\frac{k}{g}}\right)^{\frac{k}{g}}. \tag{7}$$

The probability for F to happen is

$$\begin{aligned} Prob\{F\} &= \sum_{x=0}^{gn} (Prob\{X = x\} \cdot Prob\{F \mid X = x\}) \\ &= \sum_{x=0}^{gn} \left(\binom{gn}{x} \cdot \left(\frac{1}{l}\right)^x \cdot \left(1 - \frac{1}{l}\right)^{gn-x} \cdot \left(1 - \left(1 - \frac{1}{w}\right)^{x\frac{k}{g}}\right)^{\frac{k}{g}} \right). \end{aligned} \tag{8}$$

Element e' has g membership words. Hence, the false positive ratio is

$$\begin{aligned} f_{Bg} &= (Prob\{F\})^g \\ &= \left[\sum_{x=0}^{gn} \left(\binom{gn}{x} \left(\frac{1}{l}\right)^x \left(1 - \frac{1}{l}\right)^{gn-x} \left(1 - \left(1 - \frac{1}{w}\right)^{x\frac{k}{g}}\right)^{\frac{k}{g}} \right) \right]^g. \end{aligned} \tag{9}$$

When $g = k$, exactly one bit is set in each membership word. This special Bloom-k is identical to a Bloom filter with k membership bits. To prove this, we first let $g = k$, and (9) becomes

$$\begin{aligned} f_{Bk} &= \left[\sum_{x=0}^{kn} \left(\binom{kn}{x} \cdot \left(\frac{1}{l}\right)^x \cdot \left(1 - \frac{1}{l}\right)^{kn-x} \cdot \left(1 - \left(1 - \frac{1}{w}\right)^x\right) \right) \right]^k \\ &= \left[\sum_{x=0}^{kn} \left(\binom{kn}{x} \cdot \left(\frac{1}{l}\right)^x \cdot \left(1 - \frac{1}{l}\right)^{kn-x} \right) \right. \\ &\quad \left. - \sum_{x=0}^{kn} \left(\binom{kn}{x} \cdot \left(\frac{1}{l}\right)^x \cdot \left(1 - \frac{1}{l}\right)^{kn-x} \cdot \left(1 - \frac{1}{w}\right)^x \right) \right]^k \end{aligned}$$

TABLE 4
Query Overhead Comparison of Bloom-2 Filter and Bloom Filter with $k = 3$

Data Structure	# Memory Access	# Hash Bits		
		$m = 2^{16}$	$m = 2^{20}$	$m = 2^{24}$
$B(k = 3)$	3	48	60	72
$B2(k = 3)$	2	38	46	54
$B2(h = 3\log_2 m)$	2	32–44	40–58	48–72

$$= \left[1 - \sum_{x=0}^{kn} \left(\binom{kn}{x} \cdot \left(\frac{1}{l} \cdot \left(1 - \frac{1}{w}\right)\right)^x \cdot \left(1 - \frac{1}{l}\right)^{kn-x} \right)\right]^k$$
$$= \left(1 - \left(1 - \frac{1}{lw}\right)^{kn}\right)^k. \quad (10)$$

As $m = lw$, we have $f_{Bk} = f_B$. In other words, a Bloom-k filter is identical to a Bloom filter.

3.2 Bloom with $k = 3$ versus Bloom-g

We compare the Bloom-g filters and the Bloom filter with $k = 3$ in terms of their overhead and false positive ratios. Because the overhead of Bloom-g increases with g, it is highly desirable to use a small value for g. Hence, we focus on Bloom-2 and Bloom-3 filters for evaluation.

We compare the following filters: 1) $B(k = 3)$, the Bloom filter with $k = 3$; 2) $B2(k = 3)$, the Bloom-2 filter with $k = 3$; 3) $B2(h = 3\log_2 m)$, the Bloom-2 filter that is allowed to use the same number of hash bits as $B(k = 3)$ does. In this section, we do not consider Bloom-3 because it is equivalent to $B(k = 3)$, as we have discussed in Section 3.1.

From (9), when $g = 2$, we can find the optimal value of k, denoted as $k2^*$, that minimizes the false positive ratio. Similar to the discussion for $B1(h = 3\log_2 m)$ in Section 2.4, we set the number of membership bits for each element in $B2(h = 3\log_2 m)$ to be $k2' = \min\{k2^*, \frac{3\log_2 m - 2\log_2 l}{\log_2 w}\}$. The number of hash bits to locate them is, therefore, $\log_2 l + k2' \log_2 w$, which may be smaller than $3\log_2 m$.

Table 4 compares the query overhead of three filters. $B2(k = 3)$ incurs fewer memory accesses and fewer hash bits than $B(k = 3)$. For $B2(h = 3\log_2 m)$, its number of hash bits is no more than that used by $B(k = 3)$, and it makes fewer memory accesses.

Fig. 6 presents the false positive ratios of $B(k = 3)$, $B2(k = 3)$, and $B2(h = 3\log_2 m)$; Fig. 7 shows the number of membership bits used by the filters. The figures show that $B(k = 3)$ and $B2(k = 3)$ have comparable false positive ratios in load factor range of [0.1, 1], whereas $B2(h = 3\log_2 m)$ performs better in load factor range of [0.00005, 1]. For example, when the load factor is 0.04, the false positive ratio of $B(k = 3)$ is 1.5×10^{-3} and that of $B2(k = 3)$ is 1.6×10^{-3}, while the false positive ratio of $B2(h = 3\log_2 m)$ is 3.1×10^{-4}, about one-fifth of the other two. Considering that $B2(h = 3\log_2 m)$ uses the same number of hash bits as $B(k = 3)$ but only two memory accesses per query, it is a very useful substitute of the Bloom filter to build fast and accurate data structures for membership check.

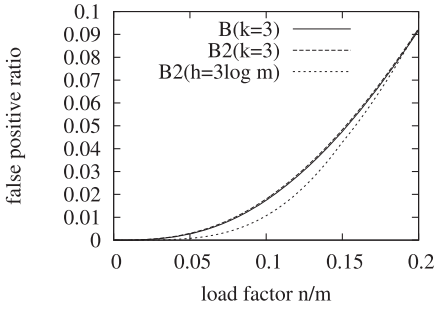


Fig. 6. False positive ratios of Bloom filter and Bloom-2 filter. Parameters: $m = 2^{20}$ and $w = 64$.

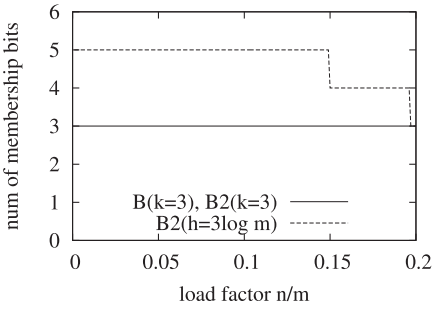


Fig. 7. Number of membership bits used by the filters.

3.3 Bloom with Optimal k versus Bloom-g with Optimal k

We now compare the Bloom-g filters and the Bloom filter when they use the optimal numbers of membership bits determined from (1) and (9), respectively. We use $B(\text{optimal } k)$ to denote a Bloom filter that uses the optimal number k^* of membership bits to minimize the false positive ratio. We use $Bg(\text{optimal } k)$ to denote a Bloom-g filter that uses the optimal number kg^* of membership bits, where $g =$ 1, 2 or 3. Fig. 8 compares their numbers of membership bits (i.e., k^*, $k1^*$, $k2^*$, and $k3^*$). It shows that the Bloom filter uses many more membership bits when the load factor is small.

Next, we compare the filters in terms of query overhead. For $1 \le g \le 3$, $Bg(\text{optimal } k)$ makes g memory accesses and uses $g\log_2 l + kg^* \log_2 w$ hash bits per membership query. Numerical comparison is provided in Table 5. To achieve a small false positive ratio, one has to keep the load factor small, which means that $B(\text{optimal } k)$ will have to make a large number of memory accesses and use a large number

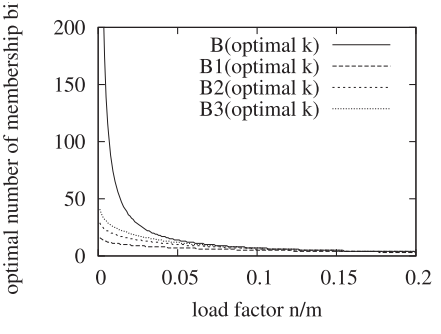


Fig. 8. Optimal number of membership bits with respect to the load factor. Parameters: $m = 2^{20}$ and $w = 64$.

TABLE 5
Query Overhead Comparison of Bloom Filter and Bloom-g Filter with Optimal k

a. Number of memory accesses per query

	Load Factor n/m				
	0.01	0.02	0.04	0.08	0.16
B(optimal k)	69	35	17	9	4
$B1$(optimal k)	1	1	1	1	1
$B2$(optimal k)	2	2	2	2	2
$B3$(optimal k)	3	3	3	3	3

b. Number of hash bits per query

	Load Factor n/m				
	0.01	0.02	0.04	0.08	0.16
B(optimal k)	1380	700	340	180	80
$B1$(optimal k)	80	74	62	50	38
$B2$(optimal k)	142	118	94	70	52
$B3$(optimal k)	198	162	126	90	66

Parameters: $m = 2^{20}$ *and* $w = 64$.

of hash bits. For example, when the load factor is 0.08, it makes nine memory accesses with 180 hash bits per query, whereas the Bloom-1, Bloom-2 and Bloom-3 filters make one, two, and three memory accesses with 50, 70, and 90 hash bits, respectively. When the load factor is 0.02, it makes 35 memory accesses with 700 hash bits, whereas the Bloom-1, Bloom-2, and Bloom-3 filters make just one, two, and three memory accesses with 74, 118, and 162 hash bits, respectively.

Fig. 9 presents the false positive ratios of the Bloom and Bloom-g filters. As we already showed in Section 2.5, $B1(\text{optimal } k)$ performs worse than $B(\text{optimal } k)$. However, the false positive ratio of $B2(\text{optimal } k)$ is very close to that of $B(\text{optimal } k)$. Furthermore, the curve of $B3(\text{optimal } k)$ is almost entirely overlapped with that of $B(\text{optimal } k)$ for the whole load-factor range. The results indicate that with just two memory accesses per query, $B2(\text{optimal } k)$ works almost as good as $B(\text{optimal } k)$, even though the latter makes many more memory accesses.

3.4 Discussion

The mathematical and numerical results demonstrate that Bloom-2 and Bloom-3 have smaller false positive ratios than Bloom-1 at the expense of larger query overhead. Below we give an intuitive explanation: Bloom-1 uses a single hash to map each member to a word before encoding. It is well known that a single hash cannot achieve an evenly distributed load; some words will have to encode much more members than others, and some words may be empty as no members are mapped to them. This uneven distribution of members to the words is the reason for larger false positives. Bloom-2 maps each member to two words and splits the membership bits among the words. Bloom-3 maps each member to three words. They achieve better load balance such that most words will each encode about the same number of membership bits. This helps them improve their false positive ratios.

3.5 Using Bloom-g in a Dynamic Environment

To compute the optimal number of membership bits, we must know the values of n, m, w, and l. The values of m, w, and l are known once the amount of memory for the filter is allocated. The value of n is known only when the filter is used to encode a static set of members. In practice, however, the filter may be used for a dynamic set of members. For example, a router may use a Bloom filter to store a watch list of IP addresses, which are identified by the intrusion detection system as potential attackers. The router inspects the arrival packets and logs those packets whose source addresses belong to the list. If the watch list is updated once a week or at the midnight of each day, we can consider it as a static set of addresses during most of the time. However, if the system is allowed to add new addresses to the list continuously during the day, the watch list becomes a dynamic set. In this case, we do not have a fixed optimal value of k^* for the Bloom filter. One approach is to set the number of membership bits to a small constant, such as three, which limits the query overhead. In addition, we should also set the maximum load factor to bound the false positive ratio. If the actual load factor exceeds the maximum value, we allocate more memory and set up the filter again in a larger bit array.

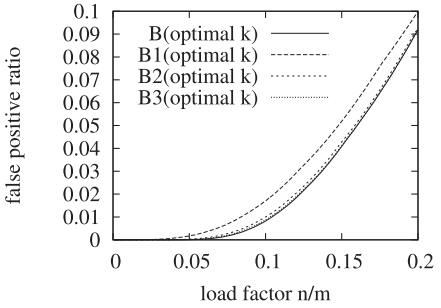


Fig. 9. False positive ratios of Bloom and Bloom-g with optimal k. Parameters: $m = 2^{20}$ and $w = 64$.

The same thing is true for the Bloom-g filter. For a dynamic set of members, we do not have a fixed optimal number of membership bits, and the Bloom-g filter will also have to choose a fixed number of membership bits. The good news for the Bloom-g filter is that its number of membership bits is unrelated to its number of memory accesses. The flexible design allows it to use more membership bits while keeping the number of memory accesses small or even a constant one.

Comparing with the Bloom filter, we may configure a Bloom-g filter with more membership bits for a smaller false positive ratio, while in the mean time keeping both the number of memory accesses and the number of hash bits smaller. Imagine a filter of 2^{20} bits is used for a dynamic set of members. Suppose the maximum load factor is set to be 0.2 to ensure a small false positive ratio. Fig. 10 compares the Bloom filter with $k = 3$, the Bloom-1 filter with $k = 6$, and the Bloom-2 filter with $k = 5$. As new members are added over time, the load factor increases from zero to 0.2. Before the load factor reaches 0.17, the Bloom-2 filter has significantly smaller false positive ratios than the Bloom filter. When the load factor is 0.04, the false positive ratio of Bloom-2 is just one-fourth of the false positive ratio of Bloom. Moreover, it makes fewer memory accesses per membership query. The Bloom-2 filter uses 58 hash bits per query, and the Bloom filter uses

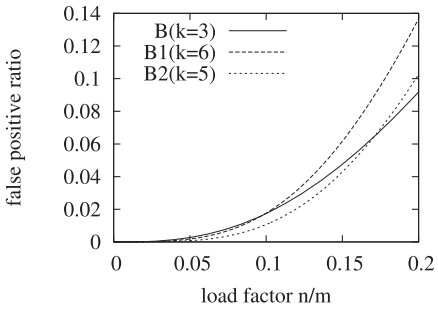


Fig. 10. False positive ratios of the Bloom filter with $k = 3$, the Bloom-1 filter with $k = 6$, and the Bloom-2 filter with $k = 5$. Parameters: $m = 2^{20}$ and $w = 64$.

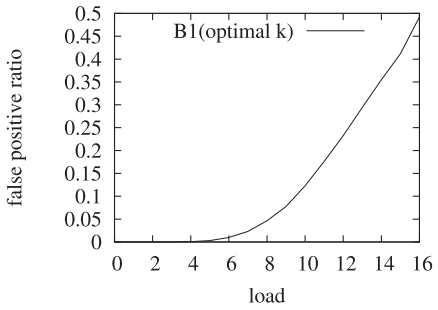


Fig. 11. False positive ratio of Bloom-1 with respect to load of words. Parameters: $m = 2^{20}$, $w = 64$, load factor $n/m = 0.1$.

60 bits. The false positive ratios of the Bloom-1 filter are close to or slightly better than those of the Bloom filter in load factor range of [0, 0.1]. It achieves such performance by making just one memory access per query and uses 50 hash bits.

4 Bloom-α: Another Generalization of Bloom-1

4.1 Motivation

From Fig. 6, we see that Bloom-2 is comparable to the Bloom filter in terms of false positive ratio when $k = 3$. However, it performs much better if it uses the same number of hash bits as Bloom does. From Fig. 9, when the optimal number of membership bits is used, Bloom-2 is almost as good as Bloom and better than Bloom-1. The downside is that Bloom-2 needs two memory accesses per query, whereas Bloom-1 needs just one. Our goal is to design a Bloom filter variant that makes a tradeoff between Bloom-1 and Bloom-2 such that false positive ratio is close to what Bloom-2 has, whereas overhead is close to what Bloom-1 has.

We have briefly touched upon the reason why Bloom-2 outperforms Bloom-1 in terms of false positive ratio (Section 3.4). Now, we give a closer look. Recall that a Bloom-1 filter uses an array of words. Let us define the *load* of a word to be the number of members that are encoded in the word. False positive ratios incurred in different words may vary. Naturally, the false positive ratio of a heavily loaded word will be higher than that of a lightly loaded word. This is confirmed by our simulation that keeps track of which word each false positive occurs in. We classify the words into groups based on their load values, and find the average false positive ratio for each group by simulation. The results are shown in Fig. 11. For words whose loads are 6 or smaller, false positive ratios are very small. However, false positive ratio grows superlinearly. When the loads are 10 or greater, false positive ratios are very high. Clearly, reducing the number of heavily loaded words helps reduce the overall false positive ratios. The approach adopted by Bloom-2 maps each member to two membership words. Each word only carries half of the membership bits. In other words, each word only encodes half of the member. Consider a heavily loaded word that serves as the membership word for 14 members. In Bloom-2, since the word encodes half of each member, its load is reduced from 14 to 7. Fig. 12 shows the frequency distribution of words with respect to load values. Fewer words have heavy loads of 10 or higher in Bloom-2 than Bloom-1. That is why Bloom-2 performs better.

However, two membership words require two memory accesses for each query. What about only some members have two membership words while others have one? In this case, some queries need two memory accesses, but others need just one. The average number of memory access per query will be between 1 and 2. Intuitively, for a lightly loaded word, we want its encoded members to have just one membership word. However, for a heavily loaded word, we want all or some of its encoded members to have additional membership words so that some of their membership bits will be moved elsewhere.

4.2 Bloom-α Filter

A Bloom-α filter (denoted as $B\alpha$) is also an array of l words, each of which is w bits long. The first $w - 1$ bits in each word are used to encode members of a set, and the last bit is a flag to indicate whether members mapped to this word have second membership words. Initially, all bits (including flag bits) are zeros. The setup procedure of a Bloom-α filter consists of two phases. In the first phase, we map elements in the set to the words in the same way as we do for Bloom-1. Each element is mapped to and encoded in exactly one word. We keep track of the elements that are encoded in each word. In the second phase, we pick a word W_1 that has the largest number of encoded members. We then "split" these members: for each member, we use $\log_2 l$ additional hash bits to locate a second word W_2 and move its last $\lfloor k/2 \rfloor$ membership bits from W_1 to W_2. Finally, the flag bit in W_1 is set to one; we call W_1 a *split word*. We keep splitting the heaviest loaded

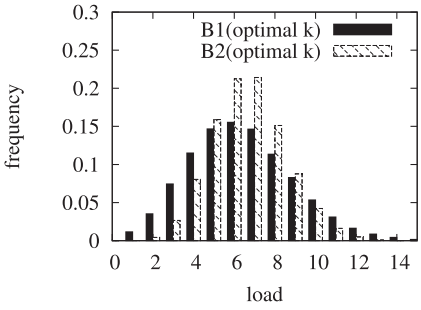


Fig. 12. Frequency distribution of words with respect to load values. Parameters: $m = 2^{20}$, $w = 64$, load factor $n/m = 0.1$ for both filters.

TABLE 6
Query Overhead Comparison of Bloom-1, Bloom-2, and Bloom-α Filter

Data Structure	# memory access	# hash bits
$B1(k=3)$, $B1$(optimal k)	1	$\log_2 l + k \log_2 w$
$B2(k=3)$, $B2$(optimal k)	2	$2\log_2 l + k \log_2 w$
$B\alpha(k=3)$, $B\alpha$(optimal k)	$\leq 1+\alpha$	$2\log_2 l + k \log_2 w$

word until a certain predefined *fraction α of members* are split, i.e., the number of members having two membership words reaches αn. As split words cannot be split again, this process will terminate as soon as enough members are split.

The construction of Bloom-α ensures that if the flag of a word is one, the members mapped to the word must have two membership words; if the flag is zero, the members have one membership word. Bloom-1 is a special case of Bloom-α with $\alpha = 0$.

To perform membership check on an element e', we first hash the element to locate its membership word W_1. If the flag of the word is zero, we check k membership bits in this word. If all these bits are ones, e' is a member of the set; Otherwise, it is not. If the flag of the word is one, we check the first $\lceil k/2 \rceil$ membership bits in W_1. If any of these bits is zero, we are sure that e' is not a member and there is no need to check the second word. However, if all these bits are ones, we find the second membership word by using $\log_2 l$ additional hash bits and check the remaining $\lfloor k/2 \rfloor$ membership bits in that word. We claim that e' is a member of the set only if all those bits are also ones.

We analyze the query overhead in Table 6. Consider an arbitrary element e'. If e' is mapped to a word whose flag is zero, it takes one memory access. If e' is mapped to a split word whose flag is one, it takes one or two memory accesses, depending on whether any of the first $\lceil k/2 \rceil$ membership bits is zero. The chance for e' to be mapped to any word is equal. Hence, the probability for e' to be mapped to a split word is equal to the fraction (or percentage) of words that are split. Considering that we split the most heavily loaded words, the average load of these words is larger than (or equal to, only if members are evenly distributed) that of the whole word array. Hence, the fraction of words being split is smaller than (or equal to) the fraction (i.e., α) of members being split. Therefore, the average number of memory accesses per query is bounded by $1 \times (1-\alpha) + 2 \times \alpha = 1 + \alpha$. Our experiment results in the next section show that, for $\alpha = 0.5$, the average number of memory accesses is actually very close to one. Because Bloom-α needs $\log_2 l$ additional hash bits to locate a second word, its total number of hash bits per query is $2\log_2 l + k \log_2 w$, comparing with $\log_2 l + k \log_2 w$ hash bits needed by Bloom-1.

Fig. 13 compares false positive ratios of Bloom-1, Bloom-2, and Bloom-α with $k = 3$ by simulations. The false positive ratio of Bloom-α is between those of Bloom-1 and Bloom-2. When the value of α is increased from 25 to 50 percent, the false positive ratio of Bloom-α is decreased, suggesting a performance-overhead tradeoff because the average number of memory accesses will increase. When $\alpha = 50\%$, the false positive ratio of Bloom-α is close to that of Bloom-2.

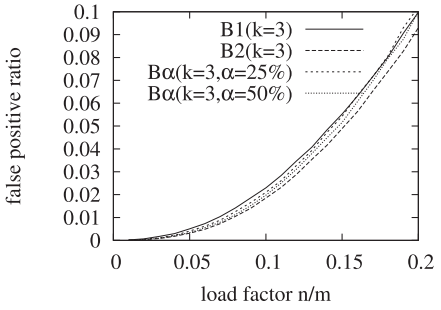


Fig. 13. False positive ratios of Bloom-1, Bloom-2 and Bloom-α Filter with $k = 3$. Parameters: $m = 2^{20}$, $w = 64$ for all filters.

5 Conclusion and Future Work

In this paper, we study one memory access Bloom filters and their generalization. This family of data structures enriches the design space of the Bloom filters and their application scope by reducing the query overhead to allow high throughput. Using a number of random bits in a word instead of from the entire bit array, we analyze the impact of this design change in terms of overhead and false positive ratio. This change also opens the door for constructing other variants for performance tradeoff. In this enlarged design space, we can configure filters that not only make fewer memory accesses but also have comparable or superior false positive ratios in scenarios where the standard Bloom filter with the optimal value of k incurs too much overhead to be practical.

In our future work, we plan to extend the techniques in this paper to counting Bloom filters. The basic idea is to divide every word into a number of small counters. Each member is first mapped to a word in the available memory, and then further mapped to k counters in the same word. To encode a member, we fetch the corresponding word in one memory access, increase k counters in the word (likely by simple hardware), and write the word back in another memory access. We will analyze this variant of counting Bloom filter theoretically and perform experiments to evaluate its performance in practical scenarios.

Another research direction is to consider multibanked on-die memory. With k banks, we can make k memory accesses in parallel. If a standard Bloom filter is implemented on multibanked memory, it may be able to perform membership query in one memory access time if its k memory accesses are made to different banks. Similarly, if we implement the Bloom-1 filter on multibanked memory, it may process up to k membership queries simultaneously in one memory access time. We will investigate the strategy of splitting the filter among the k banks to maximize the throughput of membership query.

Acknowledgments

This work was supported in part by the US National Science Foundation under grant CNS-1115548.

REFERENCES

[1] Y. Qiao, S. Chen, and T. Li, "One Memory Access Bloom Filters and Their Generalization," *Proc. IEEE INFOCOM '11,* 2011.

[2] B.H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Comm. the ACM,* vol. 13, no. 7, pp. 422-426, 1970.

[3] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Math.,* vol. 1, no. 4, pp. 485-509, 2004.

[4] S. Tarkoma, C. Rothenberg, and E. Lagerspetz, "Theory and Practice of Bloom Filters for Distributed Systems," *IEEE Comm. Surveys & Tutorials,* vol. 99, pp. 1-25, 2012.

[5] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor, "Longest Prefix Matching Using Bloom Filters," *Proc. ACM Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm. (SIGCOMM),* Aug. 2003.

[6] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing," *Proc. ACM Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm. (SIGCOMM),* Aug. 2005.

[7] H. Song, F. Hao, M. Kodialam, and T. Lakshman, "IPv6 Lookups Using Distributed and Load Balanced Bloom Filters for 100Gbps Core Router Line Cards," *Proc. IEEE INFOCOM,* Apr. 2009.

[8] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li, "Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement," *Proc. IEEE INFOCOM,* Mar. 2004.

[9] Y. Lu and B. Prabhakar, "Robust Counting via Counter Braids: An Error-Resilient Network Measurement Architecture," *Proc. IEEE INFOCOM,* Apr. 2009.

[10] P. Reynolds and A. Vahdat, "Efficient Peer-to-Peer Keyword Searching," *Proc. Int'l Middleware Conf.,* June 2003.

[11] A. Kumar, J. Xu, and E. Zegura, "Efficient and Scalable Query Routing for Unstructured Peer-to-Peer Networks," *Proc. IEEE INFOCOM,* Mar. 2005.

[12] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Trans. Networking,* vol. 8, no. 3, pp. 281-293, June 2000.

[13] L. Maccari, R. Fantacci, P. Neira, and R. Gasca, "Mesh Network Firewalling with Bloom Filters," *Proc. IEEE Int'l Conf. Comm.,* June 2007.

[14] D. Suresh, Z. Guo, B. Buyukkurt, and W. Najjar, "Automatic Compilation Framework for Bloom Filter Based Intrusion Detection," *Reconfigurable Computing: Architectures and Applications,* vol. 3985, pp. 413-418, 2006.

[15] K. Malde and B. OÕSullivan, "Using Bloom Filters for Large Scale Gene Sequence Analysis in Haskell," *Proc. 11th Int'l Symp. Practical Aspects of Declarative Languages,* pp. 183-194, 2009.

[16] J. Mullin, "Optimal Semijoins for Distributed Database Systems," *IEEE Trans. Software Eng.,* vol. 16, no. 5, pp. 558-560, May 1990.

[17] W. Wang, H. Jiang, H. Lu, and J. Yu, "Bloom Histogram: Path Selectivity Estimation for XML Data with Updates," *Proc. 30th Int'l Conf. Very Large Data Bases (VLDB),* pp. 240-251, 2004.

[18] Z. Yuan, J. Miao, Y. Jia, and L. Wang, "Counting Data Stream Based on Improved Counting Bloom Filter," *Proc. Ninth Int'l Conf. Web-Age Information Management (WAIM),* pp. 512-519, 2008.

[19] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Trans. Computer Systems,* vol. 26, no. 2, article 4, 2008.

[20] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter Braids: A Novel Counter Architecture for Per-Flow Measurement," *Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computer Systems,* June 2008.

[21] "Context-Based Access Control (CBAC): Introduction and Configuration," http://www.cisco.com/en/US/products/sw/secursw/ps1018/products_tech_note09186a0080094e8b.shtml, 2008.

[22] E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer Algorithms C++ (Chapter 3.2).* WH Freeman, 1996.

[23] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Heide, H. Rohnert, and R. Tarjan, "Dynamic Perfect Hashing: Upper and Lower Bounds," *SIAM J. Computing,* vol. 23, no. 4, pp. 738-761, 1994.

[24] M.K.F. Hao and T. Lakshman, "Building High Accuracy Bloom Filters Using Partitioned Hashing," *Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computer Systems,* June 2007.

[25] F. Hao, M. Kodialam, T. Lakshman, and H. Song, "Fast Multiset Membership Testing Using Combinatorial Bloom Filters," *Proc. IEEE INFOCOM,* 2009.

[26] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The Dynamic Bloom Filters," *IEEE Trans. Knowledge & Data Eng.,* vol. 22, no. 1, pp. 120-133, Jan. 2010.

[27] O. Rottenstreich, Y. Kanizo, and I. Keslassy, "The Variable-Increment Counting Bloom Filter," *Proc. IEEE INFOCOM,* 2012.

[28] Y. Lu, B. Prabhakar, and F. Bonomi, "Bloom Filters: Design Innovations and Novel Applications," *Proc. Allerton Conf.,* 2005.

[29] M. Moreira, R. Laufer, P. Velloso, and O. Duarte, "Capacity and Robustness Tradeoffs in Bloom Filters for Distributed Applications," *IEEE Trans. Parallel & Distributed Systems,* vol. 23, no. 12, pp. 2219-2230, Dec. 2012.

[30] F. Bonomi, M. Mitzenmacher, R. Panigrah, S. Singh, and G. Varghese, "Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines," *ACM SIGCOMM Computer Comm. Rev.,* vol. 36, no. 4, pp. 315-326, 2006.

[31] A. Pagh, R. Pagh, and S. Rao, "An Optimal Bloom Filter Replacement," *Proc. ACM-SIAM Symp. Discrete Algorithms,* pp. 823-829, 2005.

[32] M. Mitzenmacher, "Compressed Bloom Filters," *IEEE/ACM Trans. Networking,* vol. 10, no. 5, pp. 604-612, Oct. 2002.

[33] Y. Zhu, H. Jiang, and J. Wang, "Hierarchical Bloom Filter Arrays (HBA): A Novel, Scalable Metadata Management System for Large Cluster-Based Storage," *Proc. IEEE Int'l Conf. Cluster Computing,* pp. 165-174, 2004.

[34] Y. Chen, A. Kumar, and J. Xu, "A New Design of Bloom Filter for Packet Inspection Speedup," *Proc. IEEE GLOBECOM,* 2007.

[35] M. Canim, G. Mihaila, B. Bhattacharhee, C. Lang, and K. Ross, "Buffered Bloom Filters on Solid State Storage," *Proc. VLDB ADMS Workshop,* 2010.

[36] B. Debnath, S. Sengupta, J. Li, D. Lilja, and D. Du, "BloomFlash: Bloom Filter on Flash-Based Storage," *Proc. 31st Int'l Conf. Distributed Computing Systems (ICDCS),* pp. 635-644, 2011.

[37] S. Lumetta and M. Mitzenmacher, "Using the Power of Two Choices to Improve Bloom Filters," *Internet Math.,* vol. 4, no. 1, pp. 17-33, 2007.

[38] A. Kirsch and M. Mitzenmacher, "Less Hashing, Same Performance: Building a Better Bloom Filter," *Proc. 14th Conf. Ann. European Symp.,* Sept. 2006.

[39] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables," *Proc. ACM-SIAM Symp. Discrete Algorithms (SODA),* 2004.

[40] S. Cohen and Y. Matias, "Spectral Bloom Filters," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 241-252, 2003.

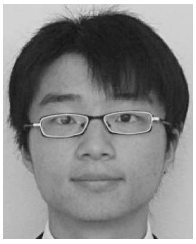
[41] Y. Hua and B. Xiao, "A Multi-Attribute Data Structure with Parallel Bloom Filters for Network Services," *Proc. 13th Int'l Conf. High Performance Computing (HiPC),* pp. 277-288, 2006.

[42] B. Xiao and Y. Hua, "Using Parallel Bloom Filters for Multi-Attribute Representation on Network Services," *IEEE Trans. Parallel & Distributed Systems,* vol. 21, no. 1, pp. 20-32, Jan. 2010.

[43] E. Porat, "An Optimal Bloom Filter Replacement Based on Matrix Solving," *Computer Science-Theory and Applications,* pp. 263-273, 2009.

[44] S. Lovett and E. Porat, "A Lower Bound for Dynamic Approximate Membership Data Structures," *Proc. Foundations of Computer Science (FOCS),* pp. 797-804, 2010.

Yan Qiao received the BS degree in computer science and Technology from Shanghai Jiao Tong University, China in 2009. She is currently working toward the PhD degree at the University of Florida (as of 2012) and her advisor is Dr. Shigang Chen. Her research interests include network measurement, algorithms, and RFID protocols. She is a student member of the IEEE.



Tao Li received the BS degree in computer science from the University of Science and Technology of China in 2007, and the PhD degree from the University of Florida in 2012. His advisor is Dr. Shigang Chen, and his research interests include network traffic measurement and RFID technologies.

Shigang Chen is a professor with the Department of Computer and Information Science and Engineering at the University of Florida. He received the BS degree in computer science from the University of Science and Technology of China in 1993. He received the MS and PhD degrees in computer science from the University of Illinois at Urbana-Champaign in 1996 and 1999, respectively. After graduation, he was with Cisco Systems for three years before joining the University of Florida in 2002. He served on the technical advisory board for Protego Networks in 2002-2003. His research interests include computer networks, Internet security, wireless communications, and distributed computing. He published more than 100 peer-reviewed journal/conference papers. He received IEEE Communications Society Best Tutorial Paper Award in 1999 and US National science Foundation (NSF) CAREER Award in 2007. He holds 11 US patents. He is an associate editor for *IEEE/ACM Transactions on Networking*, *Elsevier Journal of Computer Networks*, and *IEEE Transactions on Vehicular Technology*. He has been serving in the steering committee of IEEE IWQoS since 2010. He is a senior member of the IEEE.