Theory and Practice of Bloom Filters for Distributed Systems

Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz

***Abstract*— Many network solutions and overlay networks utilize probabilistic techniques to reduce information processing and networking costs. This survey article presents a number of frequently used and useful probabilistic techniques. Bloom filters and their variants are of prime importance, and they are heavily used in various distributed systems. This has been reflected in recent research and many new algorithms have been proposed for distributed systems that are either directly or indirectly based on Bloom filters. In this survey, we give an overview of the basic and advanced techniques, reviewing over 20 variants and discussing their application in distributed systems, in particular for caching, peer-to-peer systems, routing and forwarding, and measurement data summarization.**

***Index Terms*— Bloom filters, probabilistic structures, distributed systems**

I. Introduction

Many network solutions and overlay networks utilize probabilistic techniques to reduce information processing and networking costs. This survey presents a number of frequently used and useful probabilistic techniques. Bloom filters (BF) and their variants are of prime importance, and they are heavily used in various distributed systems. This has been reflected in recent research and many new algorithms have been proposed for distributed systems that are either directly or indirectly based on Bloom filters.

Fast matching of arbitrary identifiers to values is a basic requirement for a large number of applications. Data objects are typically referenced using locally or globally unique identifiers. Recently, many distributed systems have been developed using probabilistic globally unique random bit strings as node identifiers. For example, a node tracks a large number of peers that advertise files or parts of files. Fast mapping from host identifiers to object identifiers and vice versa are needed. The number of these identifiers in memory may be great, which motivates the development of fast and compact matching algorithms.

Given that there are millions or even billions of data elements, developing efficient solutions for storing, updating, and querying them becomes increasingly important. The key idea behind the data structures discussed in this survey is that by allowing the representation of the set of elements to lose some information, in other words to become lossy, the storage requirements can be significantly reduced.

The data structures presented in this survey for probabilistic representation of sets are based on the seminal work by Burton Bloom in 1970. Bloom first described a compact probabilistic data structure that was used to represent words in a dictionary. There was little interest in using Bloom filters for networking until 1995, after which this area has gained widespread interest both in academia and in the industry. This survey provides an up-to-date view to this emerging area of research and development that was first surveyed in the work of Broder and Mitzenmacher [1].

Section II introduces the functionality and parameters of the Bloom filter as a hash-based, probabilistic data structure. The theoretical analysis is complemented with practical examples and common practices in the underpinning hashing techniques. Section III surveys as many as twenty-three Bloom filter variants discussing their key features and their differential behaviour. Section IV covers a number of recent applications in distributed systems, such as caches, database servers, routers, security, and packet forwarding relying on packet header size Bloom filters. Finally, Section V concludes the survey with a brief summary on the rationale behind the widespread use of the polymorphic Bloom filter data structure.

S. Tarkoma and E. Lagerspetz are with University of Helsinki, Department of Computer Science

C. E. Rothenberg is with the University of Campinas (Unicamp), Department of Computer Engineering and Industrial Automation

II. Bloom Filters

The Bloom filter is a space-efficient probabilistic data structure that supports set membership queries. The data structure was conceived by Burton H. Bloom in 1970 [2]. The structure offers a compact probabilistic way to represent a set that can result in false positives (claiming an element to be part of the set when it was not inserted), but never in false negatives (reporting an inserted element to be absent from the set). This makes Bloom filters useful for many different kinds of tasks that involve lists and sets. The basic operations involve adding elements to the set and querying for element membership in the probabilistic set representation.

The basic Bloom filter does not support the removal of elements; however, a number of extensions have been developed that also support removals. The accuracy of a Bloom filter depends on the size of the filter, the number of hash functions used in the filter, and the number of elements added to the set. The more elements are added to a Bloom filter, the higher the probability that the query operation reports false positives.

Broder and Mitzenmacher have coined the *Bloom filter principle* [1]:

> Whenever a list or set is used, and space is at a premium, consider using a Bloom filter if the effect of false positives can be mitigated.

A Bloom filter is an array of m bits for representing a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements. Initially all the bits in the

filter are set to zero. The key idea is to use k hash functions, $h_i(x), 1 \leq i \leq k$ to map items $x \in S$ to random numbers uniform in the range $1, \ldots m$. The hash functions are assumed to be uniform. The MD5 hash algorithm is a popular choice for the hash functions.

An element $x \in S$ is inserted into the filter by setting the bits $h_i(x)$ to one for $1 \leq i \leq k$. Conversely, y is assumed a member of S if the bits $h_i(y)$ are set, and guaranteed not to be a member if any bit $h_i(y)$ is not set. Algorithm 1 presents the pseudocode for the insertion operation. Algorithm 2 gives the pseudocode for the membership test of a given element x in the filter. The weak point of Bloom filters is the possibility for a false positive. False positives are elements that are not part of S but are reported being in the set by the filter.

Data: x is the object key to insert into the Bloom filter.
Function: *insert(x)*
for $j : 1 \ldots k$ **do**
    /* Loop all hash functions k */
    $i \leftarrow h_j(x)$;
    if B_i == *0* **then**
        /* Bloom filter had zero bit at position i */
        $B_i \leftarrow 1$;
    end
end

Algorithm 1: Pseudocode for Bloom filter insertion

Data: x is the object key for which membership is tested.
Function: *ismember(x)* returns true or false to the membership test
$m \leftarrow 1$;
$j \leftarrow 1$;
while m == 1 *and* $j \leq k$ **do**
    $i \leftarrow h_j(x)$;
    if B_i == *0* **then**
        $m \leftarrow 0$;
    end
    $j \leftarrow j + 1$;
end
return m;

Algorithm 2: Pseudocode for Bloom member test

Figure 1 presents an overview of a Bloom filter. The Bloom filter consists of a bitstring of length 32. Three elements have been inserted, namely x, y, and z. Each of the elements have been hashed using $k = 3$ hash functions to bit positions in the bitstring. The corresponding bits have been set to 1. Now, when an element not in the set, w, is looked up, it will be hashed using the same three hash functions into bit positions. In this case, one of the positions is zero and hence the Bloom filter reports correctly that the element is not in the set. It may happen that all the bit positions of an element report that the corresponding bits have been set. When this occurs, the Bloom filter will erroneously report that the element is a member of the set. These erroneous reports are called false positives. We observe that for the inserted elements, the hashed positions correctly report that the bit is set in the bitstring.

Figure 2 illustrates a practical example of a Bloom filter through adding and querying elements. In this example, the

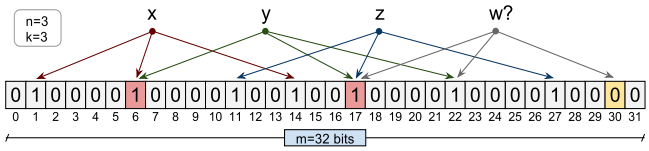

Fig. 1. Overview of a Bloom filter

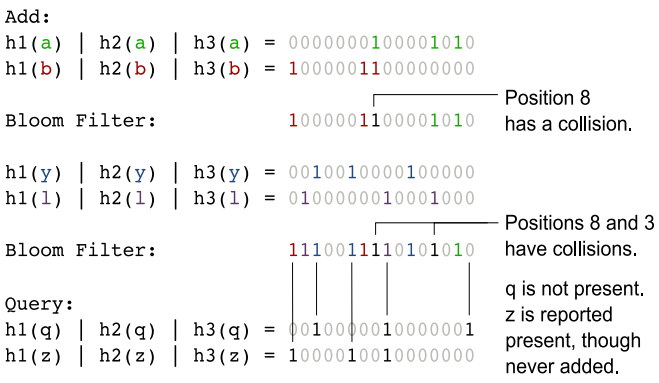

Fig. 2. Addition and query example using a Bloom filter

Bloom filter is a bitstring of length 16. The bit positions are numbered 0 to 15, from right to left. Three hash functions are used: h_1, h_2, and h_3, being MD5, SHA1 and CRC32, respectively. The elements added are text strings containing only a single letter. The Bloom filter starts out empty, with all bits unset, or zero. When adding an element, the values of h_1 through h_3 (modulo 16) are calculated for the element, and corresponding bit positions are set to one. After adding a and b, the Bloom filter has positions 15, 9, 8, 3 and 1 set. In this case, a and b have one common bit position (8). We further add elements y and l. After this, positions 15, 14, 13, 10, 9, 8, 7, 5, 3 and 1 are set. When we query for q and z, the same hash functions are used. Bit positions that correspond to q and z are examined. If the three bits for an element are set, that element is assumed to be present. In the case of q, position 0 is not set, and therefore q is guaranteed not to be present in the Bloom filter. However, z is assumed to be present, since the corresponding bits have been set. We know that z is a *false positive*: it is reported present though it is not actually contained in the set of added elements. The bits that correspond to z (positions 15, 10 and 7) were set through the addition of elements b, y and l.

For optimal performance, each of the k hash functions should be a member of the class of universal hash functions, which means that the hash functions map each item in the universe to a random number uniform over the range. The development of uniform hashing techniques has been an active area of research. An almost ideal solution for uniform hashing is presented in [3]. In practice, hash functions yielding sufficiently uniformly distributed outputs, such as MD5 or CRC32, are useful for most probabilistic filter purposes. For candidate implementations, see the empirical evaluation of 25 hash functions by Henke et al. [4]. Later in Section II-C we discuss relevant hashing techniques further.

A Bloom filter constructed based on S requires space $O(n)$ and can answer membership queries in $O(1)$ time. Given $x \in$

TABLE I
KEY BLOOM FILTER PARAMETERS

Parameters	Increase
Number of hash functions (k)	More computation, lower false positive rate as $k \to k_{opt}$
Size of filter (m)	More space is needed, lower false positive rate
Number of elements in the set (n)	Higher false positive rate

S, the Bloom filter will always report that x belongs to S, but given $y \notin S$ the Bloom filter may report that $y \in S$.

Table I examines the behaviour of three key parameters when their value is either decreased or increased. Increasing or decreasing the number of hash functions towards k_{opt} can lower false positive ratio while increasing computation in insertions and lookups. The cost is directly proportional to the number of hash functions. The size of the filter can be used to tune the space requirements and the false positive rate (fpr). A larger filter will result in fewer false positives. Finally, the size of the set that is inserted into the filter determines the false positive rate. We note that although no false negatives (fn) occur with regular BFs, some variants will be presented later in the article that may result in false negatives.

A. False Positive Probability

We now derive the false positive probability rate of a Bloom filter and the optimal number of hash functions for a given false positive probability rate. We start with the assumption that a hash function selects each array position with equal probability. Let m denote the number of bits in the Bloom filter. When inserting an element into the filter, the probability that a certain bit is not set to one by a hash function is

$$1 - \frac{1}{m}. \tag{1}$$

Now, there are k hash functions, and the probability of any of them not having set a specific bit to one is given by

$$\left(1 - \frac{1}{m}\right)^k. \tag{2}$$

After inserting n elements to the filter, the probability that a given bit is still zero is

$$\left(1 - \frac{1}{m}\right)^{kn}. \tag{3}$$

And consequently the probability that the bit is one is

$$1 - \left(1 - \frac{1}{m}\right)^{kn}. \tag{4}$$

For an element membership test, if all of the k array positions in the filter computed by the hash functions are set to one, the Bloom filter claims that the element belongs to the set. The probability of this happening when the element is not part of the set is given by

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k. \tag{5}$$

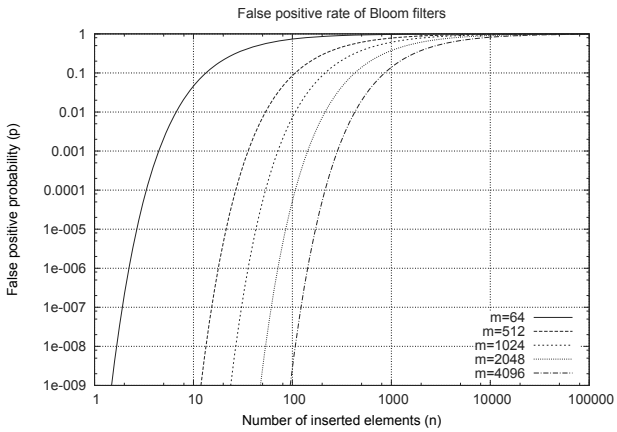

Fig. 3. False positive probability rate for Bloom filters.

We note that $e^{-kn/m}$ is a very close approximation of $(1 - \frac{1}{m})^{kn}$ [1]. The false positive probability decreases as the size of the Bloom filter, m, increases. The probability increases with n as more elements are added. Now, we want to minimize the probability of false positives, by minimizing $(1-e^{-kn/m})^k$ with respect to k. This is accomplished by taking the derivative and equaling to zero, which gives the optimal value of k

$$k_{opt} = \frac{m}{n} \ln 2 \approx \frac{9m}{13n}. \tag{6}$$

This results in the false positive probability of

$$\left(\frac{1}{2}\right)^k \approx 0.6185^{m/n}. \tag{7}$$

Using the optimal number of hashes k_{opt}, the false positive probability can be rewritten and bounded

$$\frac{m}{n} \geq \frac{1}{\ln 2}. \tag{8}$$

This means that in order to maintain a fixed false positive probability, the length of a Bloom filter must grow linearly with the number of elements inserted in the filter. The number of bits m for the desired number of elements n and false positive rate p, is given by

$$m = -\frac{n \ln p}{(\ln 2)^2}. \tag{9}$$

Figure 3 presents the false positive probability rate p as a function of the number of elements n in the filter and the filter size m. An optimal number of hash functions $k = (m/n) \ln 2$ has been assumed.

There is a factor of $\log_2 e \approx 1.44$ between the amount of

space used by a Bloom filter and the optimal amount of space that can be used. There are other data structures that use space closer to the lower bound, but they are more complicated (cf. [5], [6], [7]).

Recently, Bose et al. [8] have shown that the false positive analysis originally given by Bloom and repeated in many subsequent articles is optimistic and only a good approximation for large Bloom filters. The revisited analysis proves that the commonly used estimate (Eq. 5) is actually a lower bound and the real false positive rate is larger than expected by theory, especially for small values of m.

B. Operations

Standard Bloom filters do not support the removal of elements. Removal of an element can be implemented by using a second Bloom filter that contains elements that have been removed. The problem of this approach is that the false positives of the second filter result in false negatives in the composite filter, which is undesirable. Therefore a number of dedicated structures have been proposed that support deletions. These are examined later in this survey.

A number of operations involving Bloom filters can be implemented easily, for example the *union* and *halving* of a Bloom filter. The bit-vector nature of the Bloom filter allows the union of two or more Bloom filters simply by performing bitwise OR on the bit-vectors. Given two sets S_1 and S_2, a Bloom filter B that represents the union $S = S_1 \cup S_2$ can be created by taking the OR of the original Bloom filters $B = B_1 \vee B_2$ assuming that m and the hash functions are the same. The merged filter B will report any element belonging to S_1 or S_2 as belonging to set S. The following theorem gives a lower bound for the false positive rate of the union of Bloom filters [9]:

Theorem 1: The false positive probability of $BF(A \cup B)$ is not less than that of $BF(A)$ and $BF(B)$. At the same time, the false positive probability of $BF(A) \cup BF(B)$ is also not less than that of $BF(A)$ and $BF(B)$.

If the BF size m is divisible by 2, *halving* can be easily done by bitwise ORing the first and second halves together. Now, the range of the hash functions needs to be accordingly constrained, for instance, by applying the $mod(m/2)$ to the hash outputs.

Bloom filters can be used to approximate set *intersection*; however, this is more complicated than the union operation. One straightforward approach is to assume the same m and hash functions and to take the logical AND operation between the two bit-vectors. The following theorem gives the probability for this to hold [9]:

Theorem 2: If $BF(A \cap B)$, $BF(A)$, and $BF(B)$ use the same m and hash functions, then $BF(A \cap B) = BF(A) \cap BF(B)$ with probability $(1 - 1/m)^{k^2|A-A\cap B||B-A\cap B|}$.

The inner product of the bit-vectors is an indicator of the size of the intersection [1]. The idea of a *bloomjoin* was presented by Mackert and Lohman in 1986 [10]. In a bloomjoin, two hosts, A and B, compute the intersection of two sets S_1 and S_2, when A has the first set and B the second. It is not feasible to send all the elements from A to B, and vice versa. In a bloomjoin, S_1 is represented using a Bloom filter and sent from A to B. B can then compute the intersection and send back this set. Host A can then check false positives with B in a final round.

C. Hashing techniques

Hash functions are the key building block of probabilistic filters. There is a large literature on hash functions spanning from randomness analysis to security evaluation over many networking and computing applications. We focus on the best practices and recent developments in hashing techniques which are relevant to the performance and practicality of Bloom filter constructs. For further details, deeper theoretical foundations and system-specific applications we refer to related work, such as [4], [11], [12], [13].

One noteworthy property of Bloom filters is that the false positive performance depends only on the bit-per-element ratio (m/n) and not on the form or size of the hashed elements. As long as the size of the elements can be bounded, hashing time can be assumed to be a constant factor. Considering the trend in computational power versus memory access time, the practical bottleneck is the amount of (slow) memory accesses rather than the hash computation time. Nevertheless, whenever a filter application needs to run at line speed, hardware-amenable per-packet operations are critical [13].

In the following subsections, we briefly present hashing techniques that are the basis for good Bloom filter implementations. We start with perfect hashing, which is an alternative to Bloom filters when the set is known beforehand and it is static. Double hashing allows reducing the number of true hash computations. Partitioned hashing and multiple hashing deal with how bits are allocated in a Bloom filter. Finally, the use of simple hash functions is considered.

1) Perfect Hashing Scheme: A simple technique called *perfect hashing* (or explicit hashing) can be used to store a static set S of values in an optimal manner using a perfect hash function. A perfect hash function is a computable bijection from S to an array of $|S| = n$ hash buckets. The n-size array can be used to store the information associated with each element $x \in S$ [5].

Bloom filter like functionality can be obtained by, given a set of elements S, first finding a perfect hash function P and then storing at each location an $f = 1/\epsilon$ bit fingerprint, computed using some (pseudo-)random hash function H. Figure 4 illustrates this perfect hashing scheme.

Lookup of x simply consists of computing $P(x)$ and checking whether the stored hash function value matches $H(x)$. When $x \in S$, the correct value is always returned, and when $x \notin S$ a false positive (claiming the element being in S) occurs with probability at most ϵ. This follows from the definition of 2-universal hashing by Carter and Wengman [14], that any element y not in S has probability at most ϵ of having the same hash function value $h(y)$ as the element in S that maps to the same entry of the array.

While space efficient, this approach is disconsidered for dynamic environments, because the perfect hash function needs to be recomputed when the set S changes.

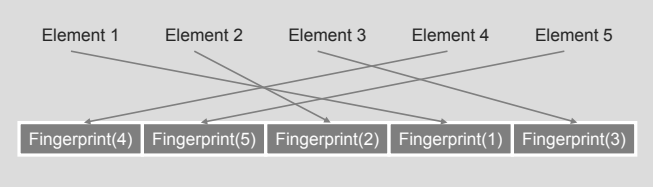

Fig. 4. Example of explicit hashing

Another technique for minimal perfect hashing was introduced by Antichi et al. [15]. It relies on Bloom filters and Blooming Trees to turn the imperfect hashing of a Bloom filter into a perfect hashing. The technique gives space and time savings. This technique also requires a static set S, but can handle a huge number of elements.

2) Double Hashing: The improvement of the *double hashing* technique over basic hashing is being able to generate k hash values based on only two universal hash functions as base generators (or "seed" hashes). As a practical consequence, Bloom filters can be built with less hashing operations without sacrificing performance. Kirsch and Mitzenmacher have shown [16] that it requires only two independent hash functions, $h_1(x)$ and $h_2(x)$, to generate additional "pseudo" hashes defined as:

$$h_i(x) = h_1(x) + f(i) * h_2(x) \tag{10}$$

where i is the hash value index, $f(i)$ can be any arbitrary function of i (e.g., i^2), and x is the element being hashed. For Bloom filter operations, the double hashing scheme reduces the number of true hash computations from k down to two without any increase in the asymptotic false positive probability [16].

3) Partitioned Hashing: In this hashing technique, the k hash functions are allocated disjoint ranges of m/k consecutive bits instead of the full m-bit array space. Following the same false positive probability analysis of Sec. II-A, the probability of a specific bit being 0 in a partitioned Bloom filter can be approximated to:

$$(1 - k/m)^n \approx e^{-kn/m} \tag{11}$$

While the asymptotic performance remains the same, in practice, partitioned Bloom filters exhibit a poorer false positive performance as they tend to have larger fill factors (more 1s) due to the m/k bit range restriction. This can be explained by the observation that:

$$(1 - 1/m)^{k*n} > (1 - k/m)^n \tag{12}$$

4) Multiple Hashing: Multiple hashing is a popular technique that exploits the notion of having multiple hash choices and having the power to choose the most convenient candidate. When applied for hash table constructions, multiple hashing provides a probabilistic method to limit the effects of collisions by allocating elements more-or-less evenly distributed. The original idea was proposed by Azar et al. in his seminal work on balanced allocations [17]. Formulating hashing as a balls into bins problem, the authors show that if n balls are placed sequentially into m for $m = O(n)$ with each ball being placed in one of a constant $d = 2$ randomly chosen bins, then, after all balls are inserted, the maximal load in a bin is, with high probability, $(ln\ ln\ n)/ln\ d + O(1)$. Vöcking et al. [18] elaborate on this observation and propose the always-go-left algorithm (or d-left hashing scheme) to break ties when inserting (chained) elements to the least loaded one among the d partitioned candidates.

As a result this hashing technique provides an almost optimal (up to an additive constant) load-balancing scheme. In addition to the balancing improvement, partitioning the hash buckets (i.e., bins) into groups makes d-left hashing more hardware friendly as it allows the parallelized look-up of the d hash locations. Thus, hash partitioning and tie-breaking have elevated d-left hashing as an optimal technique for building high performance (negligible overflow probabilities) data structures such as the multiple level hash tables (MHT) [19] or counting Bloom filters [20]. A breakthrough Bloom filter design was recently proposed using an open-addressed multiple choice hash table based on d-left hashing, element fingerprints (a smaller representation like the last f bits of the element hash) and dynamic bit reassignment [21]. After all optimizations, the authors show that the performance is comparable to plain Bloom filter constructs, outperforms traditional counting Bloom filter constructs (see d-left CBF in Sec. III-B), and easily extensible to support practical networking applications (e.g., flow tracking in Sec. IV-D).

The power of (two) choices has been exploited by Lumetta and Mitzenmacher to improve the false positive performance of Bloom filters [22]. The key idea consists of considering not one but two groups of k hash functions. On element insertion, the selection criteria is based on the group of k hash functions that sets fewer bits to 1. The caveat is that when checking for elements, both groups of k hash functions need to be checked since there is no information on which group was initially used and false positives can potentially be claimed for either group. Although it may appear counter-intuitive, under some settings (high m/n ratios), setting fewer ones in the filter actually pays off the double checking operations.

Fundamentally similar in exploiting the power of choices in producing less dense (improved) Bloom filters, the method proposed by Hao et al. [23] is based on a partitioned hashing technique which results in a choice of hash functions that set fewer bits. Experimental results show that this improvement can be as much as a ten-fold increase in performance over standard constructs. However, the choice of hash functions cannot be done on an element basis as in [22], and its applicability is constrained to non-dynamic environments.

5) Simple hash functions: A common assumption is to consider output hash values as truly random, that is, each hashed element is independently mapped to a uniform location. While this is a great aid to theoretical analyses, hash function implementations are known to behave far worse than truly random ones. On the other hand, empirical works using standard universal hashing have been reporting negligible differences in practical performance compared to predictions assuming ideal hashing (see [24] for the case of Bloom filters).

Mitzenmacher and Vadhany [25] provide the seeds to formally explaining this gap between the theory and practice

of hashing. In a nutshell, the foundation of why simple hash functions work can be explained naturally from the combination of the randomness of choosing the hash function and the randomness in the data. Hence, only a small amount of randomness in the data is enough to mimic truly random hash function in practice. These results apply for any hash-based technique, and as a practical consequence, they suggest that simple (non-cryptographic) "commodity" hash functions (e.g., CRC32) are well suited for high performance Bloom filter applications.

III. Bloom filter variants

A number of Bloom filter variants have been proposed that address some of the limitations of the original structure, including counting, deletion, multisets, and space-efficiency. We start our examination with the basic counting Bloom filter construction, and then proceed to more elaborate structures including Bloomier and Spectral filters.

A. Counting Bloom Filters

As mentioned with the treatment on standard Bloom filters, they do not support element deletions. A Bloom filter can easily be extended to support deletions by adding a counter for each element of the data structure. Probabilistic counting structures have been investigated in the context of database systems [26]. A counting Bloom filter has m counters along with the m bits. Fan et al. [27] first introduced the idea of a counting Bloom filter in conjunction with Web caches.

The structure works in a similar manner as a regular Bloom filter; however, it is able to keep track of insertions and deletions. In a counting Bloom filter, each entry in the Bloom filter is a small counter associated with a basic Bloom filter bit. When an item is inserted, the corresponding counters are incremented; when an item is deleted, the corresponding counters are decremented. To avoid counter overflow, we need choose sufficiently large counters.

The analysis from [27] reveals that 4 bits per counter should suffice for most applications [1], [28]. To determine a good counter size, we can consider a counting Bloom filter for a set with n elements, k hash functions, and m counters. Let $\mathrm{c}(i)$ be the count associated with the ith counter. The probability that the ith counter is incremented j times is a binomial random variable:

$$P(c(i) = j) = \binom{nk}{j}(\frac{1}{m})^j(1 - \frac{1}{m})^{nk-j} \qquad (13)$$

The probability that any counter is at least j is bounded above by $mP(c(i) = j)$, which can be calculated using the above formula.

The counter counts the number of times that the bit is set to one. All the counts are initially zero. The probability that any count is greater or equal to j:

$$\Pr(\max(c) \geq j) \leq m\binom{nk}{j}\frac{1}{m^j} \leq m\left(\frac{enk}{jm}\right)^j. \qquad (14)$$

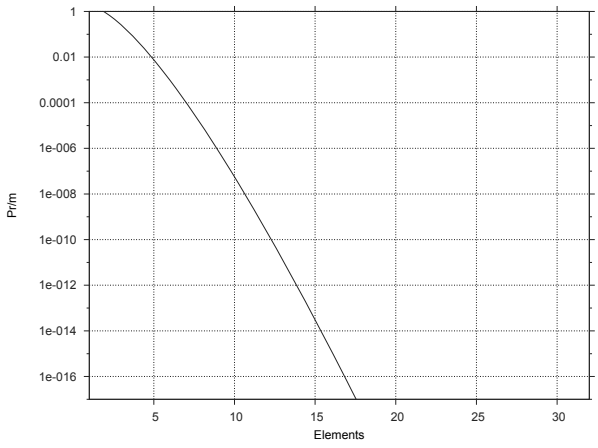

Fig. 5. Upper bound probability that any counter has at least j elements

Data: x is the item to be inserted.
Function: *insert(x)*
for $j : 1 \ldots k$ **do**
    /* Loop all hash functions k */
    $i \leftarrow h_j(x)$;
    /* Increment counter C_i */
    $C_i \leftarrow C_i + 1$;
    if B_i == *0* **then**
        /* Bit is zero at position i */
        $B_i \leftarrow 1$;
    end
end

Algorithm 3: Pseudocode for counting Bloom filter insertion

As already mentioned the optimum value for k (over reals) is $\ln 2m/n$ so assuming that the number of hash functions is less than $\ln 2m/n$ we can further bound

$$\Pr(\max(c) \geq j) \leq m\left(\frac{e \ln 2}{j}\right)^j. \qquad (15)$$

Hence taking $j = 16$ we obtain that

$$\Pr(\max(c) \geq 16) \leq 1.37 \times 10^{-15} \times m. \qquad (16)$$

In other words if we allow 4 bits per count, the probability of overflow for practical values of m during the initial insertion in the filter is extremely small. Figure 5 illustrates overflow probability as a function of counter size.

Algorithm 3 presents the pseudocode for the insert operation for element x with counting. The operation increments the counter of each bit to which x is hashed. The counting structure supports the removal of elements using the delete operation presented in Algorithm 4. The delete decrements the counter of each bit to which x is hashed. The corresponding bit is reset to zero when the counter becomes zero.

A counting Bloom filter also has the ability to keep approximate counts of items. For example, inserting element x three times results in the k bit positions being set, and the associated counters incremented by one for each insert. Therefore, the k counters associated with element x are incremented at least three times, some of them more if there are overlaps with other

```
Data: x is the item to be removed.
Function: delete(x)
for j : 1...k do
    /* Loop all hash functions k          */
    i ← h_j(x);
    /* Decrement counter C_i              */
    C_i ← C_i − 1;
    if C_i ≤ 0 then
        /* Reset bit at position i        */
        B_i ← 0;
    end
end
```

Algorithm 4: Pseudocode for counting Bloom filter deletion

inserted elements. The count estimate can be determined by finding the minimum of the counts in all locations where an item is hashed to.

In [29], Ficara et al. refine the upper bound presented above. They obtain an order of magnitude lower upper bound, producing $\Pr(\max(c) > 15) < 1.51 \times 10^{-16}$. The upper bound is given by the formula below.

$$\Pr(\max(c) > j) < Pr(\max(c) = j - 1) \tag{17}$$

Ficara et al. also propose a data structure called MultiLayer Compressed Counting Bloom Filter (ML-CCBF). The structure expands upon the idea of the CBF by adding a hierarchy of hash-based filters on top of the CBF. These are used to add space to counters that would otherwise overflow. The authors also employ Huffman coding to compress counter values, obtaining space savings. The ML-CCBF eliminates possibility of counter overflow, and retains the quick lookups of the standard BF. The cost of insert and delete operations is increased, however. For a detailed performance comparison, see [29].

B. d-left Counting Bloom Filter

Bonomi et al. [20] presented a data structure based on *d-left hashing* and fingerprints that is functionally equivalent to a counting Bloom filter, but saves approximately a factor of two or more space.

The d-left hashing scheme divides a hash table into d subtables that are of equal size. Each subtable has n/d buckets, where n is the total number of buckets. Each bucket has capacity for c cells, each cell being of some fixed bit size to store a fingerprint of the element along with a counter. When an element is placed into the table, following the d-left hashing technique, d candidate buckets are obtained by computing d independent hash values of the element. A hash-based fingerprint $f_x = H(x)$ is stored in the bucket that contains more empty cells (i.e., least inserted elements per bucket). In case of a tie, the element is placed in the bucket of the leftmost subtable with the smallest number of elements examined.

Element lookups use parallel search of the d subtables to find the fingerprint and obtain the value of the counter. In case of a deletion the counter is decremented by one. It is noteworthy that these counters can be much smaller than counters in the standard CBF due to the fewer collisions resulting from the fingerprint-based d-left construction.

The problem of knowing which candidate element fingerprint to delete – in case of fingerprint collisions – can be neatly solved by breaking the problem into two parts, namely the creation of the fingerprint, and finding the d locations by making additional (pseudo)-random permutations.

C. Compressed Bloom Filter

Compressing a Bloom filter improves performance when a Bloom filter is passed in a message between distributed nodes. This structure is particularly useful when information must be transmitted repeatedly, and the bandwidth is a limiting factor [7].

Compressed Bloom filters are used only for optimizing the transmission (over the network) size of the filters. This is motivated by applications such as Web caches and P2P information sharing, which frequently use Bloom filters to distribute routing tables. If the optimal value of the number of hash functions k in order to minimize the false positive probability is used then the probability that a bit is set in the bitstring representing the filter is $1/2$. Given the assumption of independent random hash functions, this means that the bitstring is random, and thus it does not compress well.

The key idea in compressed Bloom filters is that by changing the way bits are distributed in the filter, it can be compressed for transmission purposes. This is achieved by choosing the number of hash functions k in such a way that the entries in the m vector have a smaller probability than $1/2$ of being set. After transmission, the filter is decompressed for use. The size of k selected for compression is not optimal for the uncompressed Bloom filter, but may result in a smaller compressed filter. Compression can result in a smaller false positive rate as a function of the compressed size compared to a Bloom filter that does not use compression. The compressed Bloom filter requires that some additional compression algorithm is used for the data that is transmitted over the network, for example, Arithmetic Coding [7].

D. Deletable Bloom filter

The Deletable Bloom filter (DlBF) [30] addresses the issue of enabling element deletions at a minimal cost in memory — compared to previous variants like the CBFs — and without introducing false negatives. The DlBF is based on a simple yet powerful idea, namely keeping record of the bit regions where collisions happen and exploiting the notion that elements can be effectively removed if at least one of its bits is reset. The DlBF divides the bit array of size m into r regions. The compact representation of the collisions information consists of a bitmap of size r that codes with 0 a collision-free region (i.e., bit deletions are allowed) and with 1 otherwise (see Fig. 6).

Hence, element removal is only probabilistic and depends on the size r of the bitmap (see Fig. 7). Depending on how much memory space one is willing to invest, different rates on element deletability and false positives rates (before and after element deletions) can be achieved. The DlBF is a simple

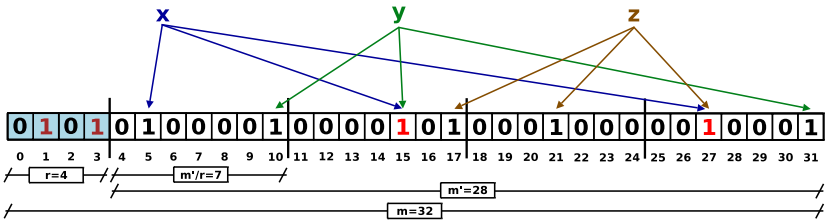

Fig. 6. Example of a DlBF with $m = 32$, $k = 3$ and $r = 4$, representing $S = \{x, y, z\}$. The 1s in the first r bits indicate collisions in the corresponding regions and bits therein cannot be deleted. All elements are deletable as each has at least one bit in a collision-free zone.

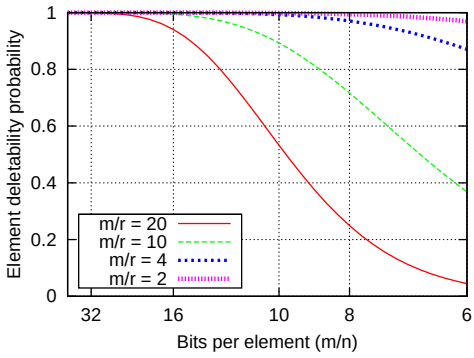

Fig. 7. Deletability estimate as function of the filter density m/n for different collision bitmap sizes r.

extension that can be easily plugged to existing BFs variants to enable probabilistic element deletions.

E. Hierarchical Bloom Filters

Shanmugasundaram et al. [31] presented a data structure called *Hierarchical Bloom Filter* to support substring matching. This structure supports the checking of a part of string for containment in the filter with low false positive rates. The filter works by splitting an input string into a number of fixed-size blocks. These blocks are then inserted into a standard Bloom filter. By using the Bloom filter, it is possible to check for substrings with a block-size granularity. This substring matching may result in combinations of strings that are incorrectly reported as being in the set (false positives). For example, a concatenation of two blocks from different strings would be incorrectly recognized as an inserted substring. Figure 8 illustrates the hierarchical nature of this construction.

The hierarchical Bloom filter construction improves matching accuracy by inserting the concatenation of blocks into the filter in addition to inserting them separately. This means that two subsequent single block matches can be verified by looking up their concatenation. This approach generalizes to a sequence of blocks; however, storage space requirements grow as more block sequences are added to the structure.

This filter was used to implement a payload attribution system that associates excerpts of packet payloads to their source and destination hosts. The filter was used to create compact digests of payloads. The system works by dividing the payload of each packet into a set of blocks of a certain fixed size. Each block is appended with its offset in the payload: *(content||offset)*. The blocks are then hashed and inserted into a Bloom filter. A hierarchical Bloom filter is a collection of the standard Bloom filters for increasing block sizes.

When a string is inserted, it is first broken into blocks which are inserted into the filter hierarchy starting from the lowest level. For the second level, two subsequent blocks are concatenated and inserted into the second level. This block-based concatenation continues for the remaining levels of the hierarchy. The resulting structure can then be used to verify whether or not a given string occurs in the payload. The search

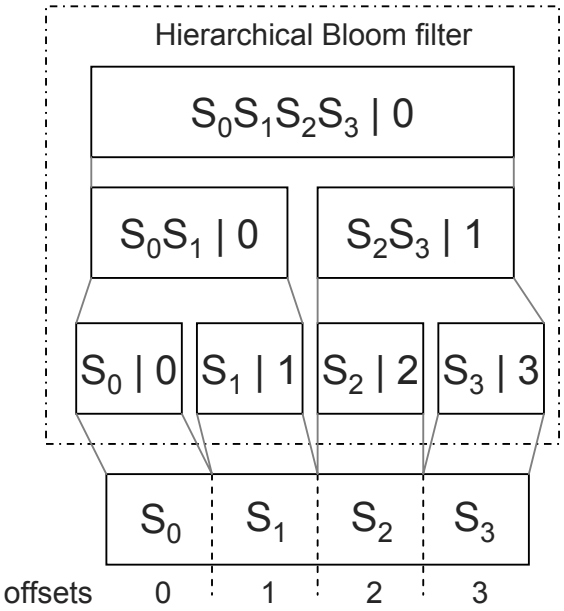

Fig. 8. Example of inserting a string into a hierarchical Bloom filter

starts at the first level and then continues upwards in the hierarchy to verify whether the substrings occurred together in the same or different packets.

F. Spectral Bloom Filters

Spectral Bloom filters generalize Bloom filters to storing an approximate multiset and support frequency queries [32]. The membership query is generalized to a query on the multiplicity of an element. The answer to any multiplicity query is never smaller than the true multiplicity, and greater only with probability ϵ. In this sense, *spectral* refers to the range within which multiplicity answers are given. The space usage is similar to that of a Bloom filter for a set of the same size (including the counters to store the frequency values). The time needed to determine a multiplicity of k is $O(\log k)$. The query time is $\Theta(\log(\frac{1}{\epsilon}))$. The answer estimate is given by returning the minimum value of the k counters determined

by the hash functions. Element additions using the minimum increase (MI) method consist of increasing only the smallest counter value(s). This helps in reducing the error rate (i.e., fraction of answer values larger than the true multiplicity) at the cost of disabling deletions. A further improvement of the error rate can be achieved using the recurring minimum (RM) method, which consists of storing elements with a single minimum (among the k counters) in a secondary Spectral Bloom filter with a smaller error probability.

G. Bloomier Filters

Bloom filters have been generalized to *Bloomier* filters [33] that compactly store function values. The Bloomier filter can encode functions instead of sets and allows the association of values with a subset of the domain elements. Bloomier filters are implemented using a cascade of Bloom filters.

A Bloomier filter encodes a function $f(x)$ by associating an arbitrary value with each member $x \in S$. For each member $x \in S$, it always returns the correct value (no false negatives). For a non–member, it returns $\perp$ as a symbol for an *undefined* value not in the range of $f(x)$, with high probability $(1 - \epsilon)$. False positives occur with probability ϵ and result in a query for $x \notin S$ returning a value within the range of $f(x)$.

The query time of a Bloomier filter is constant and space requirement is linear. The basic construction of a Bloomier filter requires $O(n \log n)$ time to create; $O(n)$ space to store and $O(1)$ time to evaluate. Although a Bloomier filter can be made mutable, the set S is immutable. This means that in a mutable Bloomier filter, function values can be changed but set membership (in S) cannot change.

The Bloomier filter can be implemented as a pipeline of parallel Bloom filters. Each parallel filter is associated with one of the values of $f(x)$. The filter pipeline is checked in pairs. Each pair of filters in the sequence are programmed with the false positives of the previous stage. For example, let filters $F(A_0)$ and $F(B_0)$ represent subsets of S that map to values $true$ and $false$, respectively. To obtain the value for x, we check the value of $F(A_0)(x)$ and $F(B_0)(x)$. If x receives a non-$\perp$ value for one filter only, its value is that value. If x receives a defined value for both filters of the pair, we move on to the pair $F(A_1)(x)$ and $F(B_1)(x)$, which contain the true positives of $F(A_0)$ that are false positives in $F(B_0)$ and the true positives of $F(B_0)$ that are false positives in $F(A_0)$, respectively. For multiple values, the filters $F(A_i)$, $i \geq 1$ contain the pairwise false positives with the filters $F(J_{i-1})$ for all $J \setminus A$.

Charles and Chellapilla [34] propose alternate construction methods of Bloomier filters that yield faster alternatives, $O(n)$ vs. $O(nlogn)$, and more practical and space-efficient constructs at the cost of increased creation time. Similarly, Dietzfelbinger and Pagh [35] propose a retrieval data structure applicable to the approximate membership problem in almost optimal space and with linear construction time. Similar results are attainable with the approach by Porat [6] as an alternate method to hold a succint, one-sided error dictionary data structure in the spirit of Bloom filters.

H. Decaying Bloom Filters

Duplicate element detection is an important problem, especially pertaining to data stream processing [36]. In the general case, duplicate detection in an unbounded data stream is not practical in many cases due to memory and processing constraints. This motivates approximate detection of duplicates among newly arrived data elements of a data stream. This can be accomplished within a fixed time window. Techniques for space-efficient approximate counts over sliding windows have been proposed in [37].

The *Decaying Bloom Filter (DBF)* structure has been proposed for this application scenario. DBF is an extension of the counting Bloom filter and it supports the removal of stale elements from the structure as new elements are inserted. DBF may produce false positive errors, but not false negatives as is the case with the basic Bloom filter. For a given space G bits and sliding window size W, DBF has an amortized time complexity of $O(\sqrt{G/W})$ [38]. A variant of DBF has been applied for hint-based routing in wireless sensor networks [39]. Time Decaying Bloom filters [40] have been proposed to take time into account by decrementing counter values.

I. Stable Bloom Filter

The Stable Bloom Filter or SBF [41] is another solution to duplicate element detection. The SBF guarantees that the expected fraction of zeros in the SBF stays constant. This makes the SBF suitable for duplicate detection in a stream of data. The authors show measurements that verify the SBF performs well in the scenario and outperforms e.g. standard buffering and standard Bloom filters. The SBF introduces both false positives and false negatives, but with rates improved from standard Bloom filters or standard buffering.

Each cell in the SBF is a counter of d bits, and thus has a maximum value $Max = 2^d - 1$. The adding function for a SBF differs from the counting Bloom filter. When adding an element, P counters chosen at random are first decremented (by one). Then the k counters that correspond to the element to be added are set to Max. The parameter P can be chosen based on the other parameters for a Bloom filter, and a user-specified accepted false positive ratio f, for example $f = 0.01$. The authors suggest choosing P using the following formula:

$$P = \frac{1}{(\frac{1}{(1-f^{1/k})^{1/Max}} - 1)(1/k - 1/m)} \tag{18}$$

Please see the full paper [41] for details on setting all the parameters.

J. Space Code Bloom Filter

Per-flow traffic measurement is crucial for usage accounting, traffic engineering, and anomaly detection. Previous methodologies are either based on random sampling (e.g., Cisco's NetFlow), which is inaccurate, or only account for the "elephants". A data structure called *Space Code Bloom Filter (SCBF)* can be used to measure per-flow traffic approximately at high speeds.

A SCBF is an approximate representation of a multiset. Each element in this multiset is a traffic flow and its multiplicity is the number of packets in the flow. SCBF employs a *Maximum Likelihood Estimation (MLE)* method to measure the multiplicity of an element in the multiset. Through parameter tuning, SCBF allows for graceful tradeoff between measurement accuracy and computational and storage complexity. SCBF also contributes to the foundation of data streaming by introducing a new paradigm called blind streaming [42].

K. Adaptive Bloom filters

The Adaptive Bloom Filter (ABF) [43] is an alternative construction to counting Bloom filters especially well suited for applications where large counters are to be supported without overflows and under unpredictable collision rate dynamics (e.g., network traffic applications). The key idea of the ABF is to count the appearances of elements by an increasing set of hash functions. Instead of working with fixed c-bit counting cells like traditional CBFs, an ABF takes the same form as a plain m-bit BF.

In order to increment the count of an element, the ABF checks *sequentially* how many independent hashes (N) map to bits set to one (in addition to the k bits set on element insertion). When the $N+k+1$th hash hits an empty cell, it is set to 1 to guarantee that element frequency queries return at least $N+1$, corresponding to the 1s set so far by the sequential hashes of the element. In membership queries, the additional number of hash functions N indicates the number of appearances of each entry. False positives among the first k bits work like in plain BF constructs. The main caveat is that the estimate of the multiplicity of a each key element becomes less precise as the ABF gets filled, since bits set by other elements result in larger N values. To its benefit, the ABF requires less memory and does not require knowledge on the estimated multiplicity of individual key elements (e.g., skewed unpredictable data set in real network traffic).

L. Variable Length Signatures and Double Buffering

A Bloom filter with *Variable-length Signatures (VBF)* is similar to the BF; however, the construction differs when inserting and querying elements [44]. When inserting an element, only $t(\leq k)$ bits of $h(x)$ computed using k hash functions are set to 1. This effectively allows the setting of a partial signature. For queries, an element x is reported to be present if at least $q(\leq k)$ bits are set to 1.

The VBF construction allows to test element membership when the set is time-varying, e.g., dynamic under insertions and deletions of elements. The VBF construction has been applied for network flow management. The key idea is to take advantage of differing flow sizes and increase or decrease the signature lengths of flows making them more easy or less easy to identify in the filter. Flow lengths can also be examined by analyzing the signature lengths. The construction can adaptively reduce the false positive rate by removing some bits of the signature, thus effectively removing the flow from the structure. The limitation of this approach is that such removal of bits may result in other valid flows being removed as well resulting in false negatives. Partial signatures can be used to alleviate this problem of false negatives. Aging of the filter can be achieved by resetting the Bloom filter bits in a round-robin fashion.

A related technique for handling time-varying sets, called *double buffering*, uses two bitmaps, active and inactive, to support time-dependent Bloom filters. When the active bitmap is half full, new signatures are stored in both bitmaps and only the active one is queried. When the inactive bitmap gets half full, it becomes active and the previously active bitmap becomes inactive and is reset. This cycle is then repeated [45].

M. Filter Banks

The standard BF only answers whether or not an element is a member of the set with some probability for misclassification. In many cases, there is a need to find which element or elements of a set are related with the input element. There is thus a requirement to support multiple binary predicates.

One straightforward technique to support multiple binary predicates is to use a set of standard BFs. For example, in a caching solution, each BF corresponds to an interface. An element originating from a certain interface is recorded in the BF corresponding to the interface. When querying for element membership, each BF is then consulted and zero or more will report containment. If multiple interfaces report containment, a number of techniques can be used to solve the issue, for example by treating the case as a cache miss and reclassifying the element in question [46].

A similar technique involving a filter bank is used to realize approximate action classification [44]. This classification answers the question, which element of S is X? This requires $\lceil \log_2 |S| \rceil$ filters. This corresponds to the selection of an action from a set of actions for a given element. This classification is important for various routing and forwarding tasks.

N. Scalable Bloom filters

One caveat with Bloom Filters is having to dimension the maximum filter size (m) a priori. This is commonly done by application designers by establishing an upper bound on the expected fpr and estimating the maximum required capacity (n). However, it is often the case that the number of elements to be stored is unknown, which leads to over-dimensioning the filters for the worse case, possibly by several orders of magnitude. Moreover, in some applications, BFs are not simply preloaded with elements and then used, but elements are added and queried independently as time passes. This may result in wasted storage space.

Scalable Bloom Filters (SBF) [47] refer to a BF variant that can adapt dynamically to the number of elements stored, while assuring a maximum false positive probability. The proposed mechanism adapts to set growth by adding "slices" of traditional Bloom Filters of increasing sizes and tighter error probabilities, added as needed. When filters get full due to the limit on the fill ratio (i.e. $\rho = 0.5$), a new one is added. Set membership queries require testing for element presence in each filter, thus the requirement on increasing sizes and tightening of error probabilities as the BF scales up. Successive

BFs are created with a tighter maximum error probability on a geometric progression, allocating $m \cdot a^{i-1}$bits for its *i*–th BF slice, where a is a given positive integer and $1 < i < s$. As a result, the compounded probability over the whole series converges to the target design value, even accounting for an infinite series.

Parameters of the SBF in addition to the initial bit size m and target fpr include the expected growth rate (s) and the error probability tightening ratio (r). Careful choosing of these extra 2 parameters ultimately determines the space usage gains of SBF compared to standard BF constructs.

O. Dynamic Bloom Filter

Standard BFs and its mainstream variations suffer from inefficiencies when the cardinality of the set under representation is unknown prior to design and deployment. In stand-alone applications with dynamic sets (i.e., with element addition and removal operations), the inefficiency arises from the impossibility of determining the optimal BF parameters (m,k) in advance. Without knowledge of the upper bound on the number of elements to be represented, a target false positive probability threshold cannot be guaranteed unless the BF is rebuilt from scratch each time the set cardinality changes. These limitations are not only a challenge for stand-alone applications. In distributed applications, BF reconstruction is cumbersome and may hinder interoperability.

Dynamic Bloom filters (DBF) address the requirement for dynamically adjusting the size of a probabilistic filter [48]. The DBF construction is based on a dynamic $s \times m$ bit matrix that consists of s standard (or counting) Bloom filters. The filter size m and the number of hash functions k are system parameters. The number of BF slices is adjusted at runtime to allow the DBF to grow dynamically.

The DBF is based on the notion of an active Bloom filter. Only one Bloom filter in DBF is active at a time and others are inactive. The number of elements inserted into each constituent Bloom filter in a DBF is tracked. During insertion, the first BF that has its element counter less than the given threshold (system parameter) is selected as the active BF. If such an active BF cannot be found, a new BF is created and designated as the active BF. The element is then inserted into the active BF. The query element membership operation iterates the set of BFs in the DBF and returns true if any of the BFs contain the element. Removing an element requires first finding the sub-BF claiming that the element is present. In case only one is found, the element is removed by decrementing the k counters by one. If multiple filters return true, the element removal may result in, at most, k potential false negatives. In this case, to conserve the false negative free properties, the element bit cells are not decremented. Such element deletion failures contribute to a gradual increase in the false positive behaviour.

The DBF has been intended for a number of distributed environments, especially those in which new data is inserted (and potentially removed) frequently. The DBF requires that the filter size and the number of hash functions are consistent among all nodes. The key applications include Bloomjoins, informed search, and index search.

P. Split Bloom Filters

A Split Bloom filter (SPBF) [49] employs a constant $s \times m$ bit matrix for set representation, where s is a pre-defined constant based on the estimation of maximum set cardinality. The SPBF aims at overcoming the limitation of standard BFs which do not take sets of variable sizes into account. The basic idea of the SPBF is to allocate more memory space to enhance the capacity of the filter before its implementation and actual deployment. The false match probability increases as the set cardinality grows. An existing SPBF must be reconstructed using a new bit matrix if the false match probability exceeds an upper bound.

Q. Retouched Bloom filters

The Retouched Bloom filter (RBF) [50] builds upon two observations. First, for many BF applications, there are some false positives, which are more troublesome than others and can be identified after BF construction but prior to deployment. Second, there are cases where a low level of false negatives is acceptable. For filter applications fulfilling these two requirements, the RBF enables trading off the most troublesome false positives for some randomly introduced false negatives.

The novel idea behind the RBF is the *bit clearing process* by which false positives are removed by resetting individual bits. Performance gains can be measured by the proportion of false positives removed compared to the proportion of false negatives introduced.

In case of a *random* bit clearing process, the gains are neutral, i.e., the fpr decrease equals the fnr increase. A better performance can be achieved using a *selective* clearing approach, which first tests for false positives for a given training set, and then resets only the bits belonging to "troublesome" elements. The authors propose four algorithms for decreasing the fpr more than the corresponding fnr increase.

R. Generalized Bloom Filters

The basic idea of the Generalized Bloom Filter (GBF) [51] is to employ two sets of hash functions, one $(g_1, \ldots, g_{k_0})$ for setting bits and another $(h_1, \ldots, h_{k_1})$ to reset bits. A GBF starts out as an arbitrary bit vector set with both 1s and 0s, and information is encoded by setting chosen bits to either 0 or 1, departing thus from the notion that empty bit cells represent the absence of information. As a result, the GBF is a more general binary classifier than the standard Bloom filter. In the GBF, the false-positive probability is upper bounded and it does not depend on the initial condition of the filter. However, the generalization brought by the set of hash functions resetting bits introduces false negatives, whose probability can be upper bounded and does not depend either on the bit filter initial set-up.

Element insertion works by setting to 0 the bits defined by $g_1(x), \ldots, g_{k_0}(x)$ and setting to 1 the k_1 bits at positions $h_1(x), \ldots, h_{k_1}(x)$. In case of a collision, the bit is set to 0. Analogously, membership queries are done by verifying if all bits defined by $g_1(x), \ldots, g_{k_1}(x)$ are set to 0 and all bits determined by $h_1(x), \ldots, h_{k_1}(x)$ are set to 1. The GBF returns false if any bit is inverted, i.e. the queried element does not

belong to the set with a high probability. The false positive and false negative estimates can be traded off by varying the numbers of hash functions, k_0 and k_1.

S. Distance-sensitive Bloom filters

Distance-sensitive Bloom filters (DSBF) [52] were conceived by Kirsch and Mitzenmacher to answer approximate set membership queries in the form of *is x close to an item of S?*, where closeness is measured under a suitable metric. More specifically, given a metric space (U, d), a finite set $S \subset U$, and parameters $0 \leq \epsilon < \delta$, the filter aims to effectively distinguish between inputs $u \in U$ such that $d(u, x) \leq \epsilon$ for some $x \in S$ and inputs $u \in U$ such that $d(u, x) \geq d$ for every $x \in S$.

The DSBF is implemented using locality-sensitive hash functions [53], [54] and allows false positives and false negatives. By comparison, standard Bloom filters are false-negative-free corresponding to the case where $\epsilon = 0$ and δ is any positive constant. While false positives and especially false negatives require special consideration at application design time, a DSBF can provide speed and space improvements for networking and database applications, which can avoid full nearest-neighbor queries or costly comparison operations against entire sets. Moreover, overarching DSBFs can be constructed on top of a collection of conventional BFs to provide a quick (probabilistic) answer to questions of the form, *Are there any sets in the collection very close to this query set*?, which may assist traditional BF-based distributed applications.

T. Data Popularity Conscious Bloom Filters

In many information processing environments, the underlying popularities of data items and queries are not identical, but rather they differ and skewed. For example in many networks data popularity has been observed to be similar to the Zipf distribution. The standard Bloom filter does not utilize information pertaining to the underlying data element distribution. An intuitive approach to take data item popularity into account is to use longer encodings and more hash functions for important elements and shorter encodings and fewer hash functions for less important ones. A larger number of hash functions will result in fewer false positives for popular data elements. It may result in more false positives for unpopular data items; however, since they are requested less frequently this is not expected to become an issue [55].

Thus the Bloom filter construction lends itself well to data popularity-conscious filtering as well; however, this requires the minimization of the false positive rate by adapting the number of hashes used for each element to its popularities in sets and membership queries. To this end, an object importance metric was proposed in [55]. The problem was modeled as a constrained nonlinear integer program and two polynomial-time solutions were presented with bounded approximation ratios. The aim of the optimization problem, modeled as a variant of the knapsack problem, is to find the optimal number of hash functions for each element. The popularities of elements are used to reduce the solution search space.

The results include a 2-approximation algorithm with $O(N^c)$ running time ($c \geq 6$ in practice) and a $(2 + \epsilon)$ approximation algorithm with running time $O(N^2/\epsilon)$, $\epsilon > 0$. Experimental evaluation results indicate that the popularity-conscious Bloom filters can achieve significant false-positive probability reduction (or reduced filter sizes when the false positive rate is kept constant) compared to standard Bloom filters. On the other hand, the popularity-conscious filters require offline computation for estimating input distribution popularities and storage for the custom hash scheme.

U. Memory-optimized Bloom Filter

A memory-optimized Bloom filter was proposed in [56] that uses an additional hash function to select one of the possible k locations in a Bloom filter. Thus only a single bit is set for each element instead of k bits leading to memory savings. The idea of using a separate hash function to make the result of the k hash functions more uniform has also been proposed in [46].

V. Weighted Bloom filter

Bruck et al. [57] propose Weighted Bloom filter (WBF), a Bloom filter variant that exploits the a priori knowledge of the frequency of element requests by varying the number of hash functions (k) accordingly as a function of the element query popularity. Hence, a WBF incorporates the information on the query frequencies and the membership likelihood of the elements into its optimal design, which fits many applications well in which popular elements are queried much more often than others. The rationale behind the WBF design is to consider the filter fpr as a weighted sum of each individual element's false positive probability, where the weight is positively correlated with the element's query frequency and is negatively correlated with the element's probability of being a member. As a consequence, in applications where the query frequencies can be estimated or collected and result for instance in a step or the Zipf distribution, the WBF largely outperforms in fpr the traditional Bloom filter. Even a simple binary classification of elements between hot and cold can result in false positive improvements of a few orders of magnitude.

W. Secure Bloom filters

The hashing nature of Bloom filters provide some basic security means in the sense that the identities of the set elements represented by the BF are not clearly visible for an observer. However, plain BFs allow some leak of information such as the approximate total number of elements inserted. Morever, BFs are vulnerable to correlation attacks where the similarity of BFs' contents can be deduced by comparing BF indexes for overlaps, or lack thereof. Furthermore, in applications where the hash functions are known, a dictionary attack provides probabilistic arguments for the presence of elements in a given BF.

To overcome these limitations, several proposals have suggested secured BF variants as a natural extension of the problem of constructing data structures with privacy guarantees.

TABLE II
KEY FEATURES OF THE BLOOM FILTER VARIANTS, INCLUDING THE ADDITIONAL CAPABILITIES: COUNTING (C), DELETION (D), POPULARITY-AWARENESS (P), FALSE-NEGATIVES (FN), AND THE OUTPUT TYPE.

Filter	Key feature	C	D	P	FN	Output
Standard Bloom filter	Is element x in set S?	N	N	N	N	Boolean
Adaptive Bloom filter	Frequency by increasing number of hash functions	Y	N	N	N	Boolean
Bloomier filter	Frequency and function value	Y	N	N	N	Freq., $f(x)$
Compressed Bloom filter	Compress filter for transmission	N	N	N	N	Boolean
Counting Bloom filter	Element frequency queries and deletion	Y	Y	N	M	Boolean or freq.
Decaying Bloom filter	Time-window	Y	Y	N	N	Boolean
Deletable Bloom filter	Probabilistic element removal	N	Y	N	N	Boolean
Distance-sensitive Bloom filters	Is x close to an item in S?	N	N	N	Y	Boolean
Dynamic Bloom filter	Dynamic growth of the filter	Y	Y	N	N	Boolean
Filter Bank	Mapping to elements and sets	Y	Y	M	N	x, set, freq.
Generalized Bloom filter	Two set of hash functions to code x with 1s and 0s	N	N	N	Y	Boolean
Hierarchical Bloom filter	String matching	N	N	N	N	Boolean
Memory-optimized Bloom filter	Multiple-choice single hash function	N	N	N	N	Boolean
Popularity conscious Bloom filter	Popularity-awareness with off-line tuning	N	N	Y	N	Boolean
Retouched Bloom filter	Allow some false negatives for better false positive rate	N	N	N	Y	Boolean
Scalable Bloom filter	Dynamic growth of the filter	N	N	N	N	Boolean
Secure Bloom filters	Privacy-preserving cryptographic filters	N	N	N	N	Boolean
Space Code Bloom filter	Frequency queries	Y	N	M	N	Frequency
Spectral Bloom filter	Element frequency queries	Y	Y	N	M	Frequency
Split Bloom filter	Set cardinality optimized multi-BF construct	N	N	N	N	Boolean
Stable Bloom filter	Has element x been seen before?	N	Y	N	Y	Boolean
Variable-length Signature filter	Popularity-aware with on-line tuning	Y	Y	Y	Y	Boolean
Weighted Bloom filter	Assign more bits to popular elements	N	N	Y	N	Boolean

The secure indexes [58] by Goh enhance the BF insert and query operations by applying pseudo-random functions twice, first to generate element codewords using a secret key, and second to derive the k index bits after including a set-specific identifier as input to the keyed hash functions.

Finally, Goh proposes a simple technique to further obscure the BF by randomly setting additional bits increasing the bar for attackers at the cost of a fpr increase.

Encrypted Bloom filters by Bellovin and Cheswick [59] propose a privacy-preserving filter variant of Bloom filters which introduces a semi-trusted third party to transform one party's queries to a form suitable for querying the other party's BF, in such a way that the original query privacy is preserved. Instead of undisclosing the keys of all parties and securing the BF operations with keyed hash functions as per Goh [58], Bellovin and Cheswick propose a specialized form of encryption function where operations can be done on encrypted data. More specifically, their proposal is based on the Pohlig-Hellman cipher, which forms an Abelian group over its keys when encrypting any given element.

Yet another refinement on privacy-preserving variants of Bloom filters is the cryptographically secure Bloom filter protocol proposed by [60]. In addition to providing a reasonable security definition, the proposed protocol suite avoids employing third parties by using cryptographic primitives known as blind signature schemes and oblivious pseudorandom functions.

X. Summary and discussion

Table II summarizes the distinguishing features of the Bloom filter variants discussed in this section. The different Bloom filter designs aim at addressing specific concerns regarding space and transmission efficiency, false positive rate, dynamic operation in terms of increasing workload, dynamic operation in terms of insertions and deletions, counting and frequencies, popularity-aware operation, and mapping to elements and sets instead of simple set membership tests. For each variant, table II indicates the output type (e.g., boolean, frequency, value) and whether counting (C), deletion (D), or popularity-awareness (P) are supported (Yes/No/Maybe), or false negatives (FN) are introduced. Bloom filter variants with counting capabilities can also be used to probabilistically encode arbitrary functions by considering the cardinality of each set element being functional value and each set element being a variable.

Bloom filters come in many shapes and forms, and they are widely used in distributed systems due to their compact nature and configurable trade-off between size and accuracy. Making this choice and optimizing the parameters for the expected uses cases are fundamental factors to achieve the desired performance in practice.

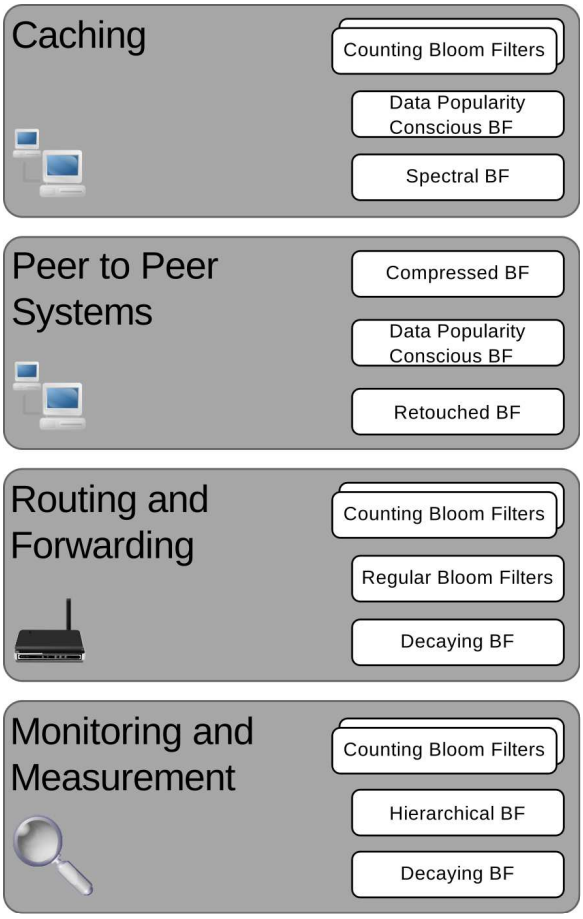

Fig. 9. Bloom filter variants grouped by usage scenarios.

Since there is no Bloom filter that fits all, one key question that application designers should ask is whether false negatives are tolerable or not. Relaxing this constraint can help drastically in reducing the overall false positive rate (cf. retouched Bloom filters [50]), but raises also the question whether the Bloom filter is the right data structure choice despite alternative designs specific to the application domain (cf. [61]), approximate dictionary-inspired approaches [6], [35], cache-efficient variants (blocked Bloom filter) and Golomb coding implementations as proposed by Putze *et al* [62], space-efficient versions of cuckoo hashing [63], and more complex but space-optimal alternatives [5], [6].

Each variant or replacement introduces a specific trade-off involving execution time, space efficiency, and so on. Ultimately, which probabilistic data structure is best suited depends a lot on the application specifics. Indeed, the variations of the standard Bloom filter discussed in this Section are commonly the result of specific requirements of network and distributed system applications, a variety of which we present in the following survey section.

IV. Bloom Filters in Distributed Computing

We have surveyed techniques for probabilistic representation of sets and functions. The applications of these structures are manyfold, and they are widely used in various networking systems, such as Web proxies and caches, database servers, and routers. We focus on the following key usage scenarios:

- Caching for Web servers and storage servers.
- Supporting processing in P2P networks, in which probabilistic structures can be used for summarizing content and caching [28], [64].
- Packet routing and forwarding, in which Bloom filters and variants have important roles in flow detection and classification.
- Monitoring and measurement. Probabilistic techniques can be used to store and process measurement data summaries in routers and other network entities.
- Supporting security operations, such as flow admission and intrusion detection.

Figure 9 shows an overview of Bloom filter variants that can be used in the usage scenarios that this section focuses on. For more detail, see Figure 15 at the end of this article.

A. Caching

Bloom filters have been applied extensively to caching in distributed environments. To take an early example, Fan, Cao, Almeida, and Broder proposed the Summary Cache [27], [28] system, which uses Bloom filters for the distribution of Web cache information. The system consists of cooperative proxies that store and exchange summary cache data structures, essentially Bloom filters. When a local cache miss happens, the proxy in question will try to find out if another proxy has a copy of the Web resource using the summary cache. If another proxy has a copy, then the request is forwarded there.

In order for distributed proxy-based caching to work well, the proxies need to have a way to compactly summarize available content. In the Summary Cache system, proxies periodically transfer the Bloom filters that represent the cache contents (URL lists). Figure 10 illustrates the use of a Bloom filter-based summary cache at a proxy. The summary cache is consulted and used to find nearest servers or other proxies with the requested content.

Dynamic content poses a challenge for caching content and keeping the summary indexes up to date. Within a single proxy, a Bloom filter representing the local content cache needs to be recreated when the content changes. This can be seen to be inefficient and as a solution the Summary Cache uses counting Bloom filters for the maintenance of their local cache contents, and then based on the updates a regular Bloom filter is broadcast to other proxies.

The summary cache-based technique is used in the popular Squid Web Proxy Cache[1]. Squid uses Bloom filters for so-called cache digests. The system uses a 128-bit MD5 hash of the key, a combination of the URL and the HTTP method, and splits the hash into four equal chunks. Each chunk modulo the digest size is used as the value for one of the Bloom filter hash functions. Squid does not support deletions from the digest and thus the digest must be periodically rebuilt to remove stale information.

Bloom filters have been applied extensively in distributed storage to minimize disk lookups. As an example, we consider

[1] www.squid-cache.org

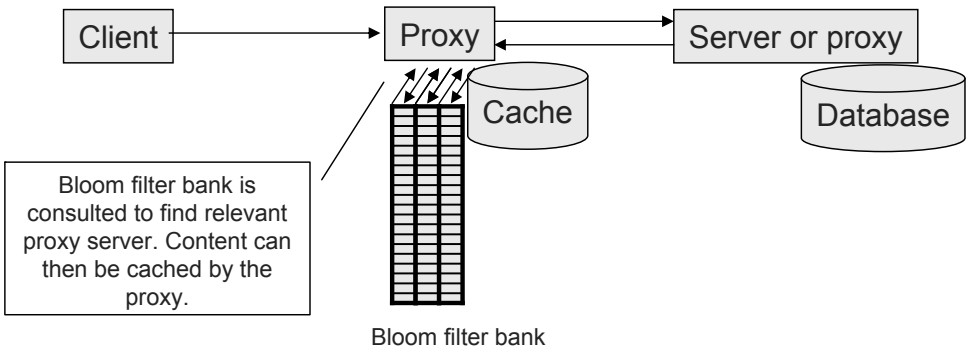

Fig. 10. Bloom filters for caching proxies

Google's Bigtable system that is used by many massively popular Google services, such as Google Maps and Google Earth, and Web indexing. Bigtable is a distributed storage system for structured data that has been designed with high scalability requirements in mind, for example capability to store and query petabytes of data across thousands of commodity servers [65].

A Bigtable is a sparse multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp. Each value in the map is an uninterpreted array of bytes. Bigtable uses Bloom filters to reduce the disk lookups for non-existent rows or columns [65]. As a result the query performance of the database has to rely less on costly disk operations and thus performance increases.

Apache Hadoop [66] is a framework for running applications on clusters of commodity hardware. Hadoop implements the map/reduce paradigm in which an application is divided into smaller fragments in order to achieve parallel efficiency. The Hadoop implementation uses various Bloom filter structures to optimize the reduce stage.

B. P2P Networks

Bloom filters have been extensively applied in P2P environments for various tasks, such as compactly storing keyword-based searches and indices [67], synchronizing sets over network, and summarizing content.

In [68], the applications and parameters of Bloom filters in P2P networks are discussed. The applications identified by the authors include peer content summarization and the filter length, compression, and hash types used, semantic overlays using peer Bloom filter similarity, and query routing by Bloom filter similarity. Updating of peer Bloom filters is also discussed.

The exchange of keyword lists and other metadata between peers is crucial for P2P networks. Ideally, the state should be such that it allows for accurate matching of queries and takes sublinear space (or near constant space). The later versions of the Gnutella protocol use Bloom filters [68] to represent the keyword lists in an efficient manner. In Gnutella, each leaf node sends its keyword Bloom filter to an ultra-node, which

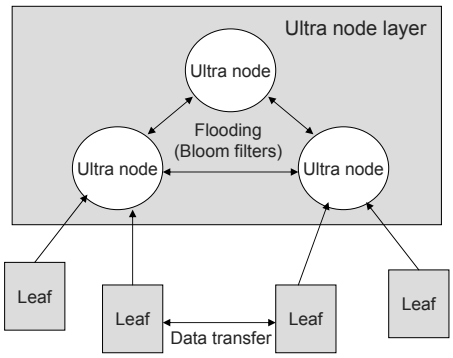

Fig. 11. 2-tier Gnutella

can then produce a summary of all the filters from its leaves, and then sends it to neighbouring ultra-nodes. The ultra-nodes are hubs of connectivity, each being connected to more than 32 other ultra-nodes. Figure 11 illustrates this two-tier Gnutella architecture.

Rhea and Kubiatowicz [69] designed a probabilistic routing algorithm for P2P location mechanisms in the OceanStore project. Their aim was to determine when a requested file has been replicated near the requesting system. This system uses a construction called *Attenuated Bloom filter*, which is simply an array of d basic Bloom filters. The ith basic filter keeps record of what files are reachable within i hops in the network. The attenuated Bloom filter only finds files within d hops, but the returned paths are likely to be the shortest paths to the replica. In the distributed system, a node maintains attenuated filters for each neighbour separately, and updates are broadcast periodically.

The OceanStore system uses a two-tiered model, in which the attenuated filter is part of the first tier. If the probabilistic search fails, the search can then fallback to a deterministic overlay search using Tapestry.

In [70], the authors propose to exploit two-dimensional locality to improve P2P system search efficiency. They present a locality-aware P2P system architecture called Foreseer, which

explicitly exploits geographical locality and temporal locality by constructing a neighbor overlay and a friend overlay, respectively. Each peer in Foreseer maintains a small number of neighbors and friends along with their content filters used as distributed indices.

Exponentially Decaying Bloom filters probabilistically encode routing tables in a highly compressed way that allows for efficient aggregation and propagation of routing information in unstructured peer-to-peer networks [71].

Bloom filters can be applied for approximate set reconciliation and data synchronization [72]. This application is important for P2P systems, in which a peer may send a compact data structure to another peer that represents items that the peer already has. Bloom filters are not directly ideal for this kind of set reconciliation applications, because of the possibility for false positives. Therefore a number of Bloom filter-based structures have been developed [73], [74].

Bloom filters have also been used in social networks, for example in Tribler [75], a social P2P file sharing system. Tribler uses Bloom filters to keep the databases that maintain the social trust network synchronized between peers. The Bloom filters are used to filter out peers already known by message destination nodes from swarm discovery messages. Tribler can reach common friends–of–friends of two peers by using a Bloom filter of 260 bytes in size, enabling a peer to exchange information with thousands of others in a short time.

C. Packet Routing and Forwarding

Bloom filters have been used to improve network router performance [76]. Song et al. used a Counting Bloom Filter to optimize a hash table used in network processing, such as maintaining per-flow context, IP route lookup, and packet classification. The small, on-chip Bloom filter eliminates slow, off-chip lookups when the searched flow is not found, and minimizes the number of lookups required when the flow is found. This is done by associating a hash table bucket with each Bloom filter counter. The bucket associated with the counter with the lowest value and lowest index is then always accessed, and the corresponding item is stored in that bucket. Counters are also artificially incremented to eliminate collisions. This leads to one worst-case off-chip lookup for flows stored.

In [77], Bloom filters are used for high-speed network packet filtering. A regular Bloom filter with a collision list is implemented in kernel space in a Linux network driver. The filter is populated by signatures of (protocol, IP address, port)–tuples. Incoming packets are matched against the filter and matches given to a user-space network monitoring program. Wildcards are supported by setting one of the tuple fields to zero when populating the filter, and on input packets when querying. The authors also implement a threaded network packet processor to offload packet processing from the Linux kernel to a separate thread. With the Bloom filter the authors almost quadruple the performance of the existing driver, as compared to when capturing all packets and filtering in user-space only.

In the remainder of the subsection, we focus on important uses of Bloom filter variants in routing and forwarding tasks. These cases include IP lookups, loop and duplicate detection, forwarding engines, and deep packet scanning. We also briefly discuss the use of Bloom filters for content-based publish/subscribe and multicast, which is an active research area.

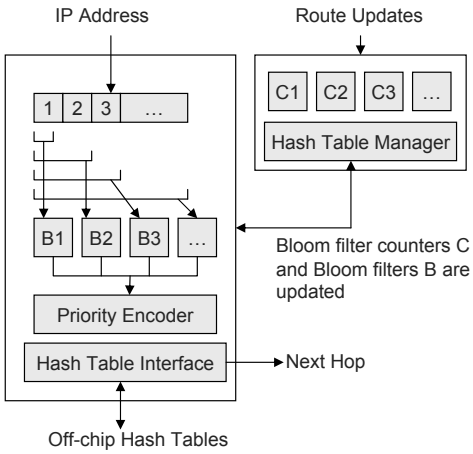

Fig. 12. Longest Prefix Matching with Bloom filters

1) IP Lookups: Bloom filters can be applied in various parts in a routing and forwarding engine. Probabilistic techniques have been used for efficient IP lookups. IP routers forward packets based on their address prefixes. Each prefix is associated with the next hop destination. CIDR-based routing and forwarding uses the longest prefix match for finding the next hop destination. This is commonly solved using a binary search, a trie search, or a TCAM. IP lookups can be made more efficient by dividing the addresses into tables based on their length and then utilizing binary search to find the longest common prefix. The d-left hashing technique has been used to make this lookup more compact and efficient [78].

Many different probabilistic structures have been developed for fast packet forwarding. To take one example, an algorithm that uses Bloom filters for *Longest Prefix Matching (LPM)* was introduced in [79]. The algorithm performs parallel queries on Bloom filters, to determine address prefix membership in sets of prefixes sorted by prefix length. This work indicates that Bloom filter–based forwarding engines can offer favorable performance characteristics compared to TCAMs used by many routers. Figure 12 illustrates this design for high–speed prefix matching. The idea is to have different regular Bloom filters for different address prefixes. These BFs are implemented in hardware and updated by a route computation process. The route manager uses counting Bloom filters to keep track of how the regular BFs should be instrumented.

Asymmetric Bloom filters that allocate memory resources according to prefix distribution have been proposed for LPM. By using direct lookup array and *Controlled Prefix Expansion (CPE)*, worst-case performance is limited to two hash probes and one array access per lookup. Performance analysis indicates that average performance approaches one hash probe per lookup with less than 8 bits per prefix [79].

The system employs a set of W Counting Bloom Filters

where W is the length of input addresses, and associates one filter with each unique prefix length. A hash table is also constructed for each distinct prefix length. Each hash table is initialized with the set of corresponding prefixes, where each hash entry is a (prefix, next hop)–pair.

Based on the analysis, the expected number of hash probes per lookup depends only on the total amount of memory resources, M, and the total number of supported prefixes, N. The number of required hash probes is given by $(\frac{1}{2})^{\frac{M/N}{\ln 2}}$. The result is independent of the number of unique prefix lengths and the distribution of prefixes among the prefix lengths.

2) Loop Detection: Bloom filters can be used for loop detection in network protocols. IP uses the Time-To-Live (TTL) field to detect and drop packets that are in a forwarding loop. The TTL counter is incremented for each network hop. For small loops, TTL may still allow a substantial amount of looping traffic to be generated.

Icarus is a system that uses Bloom filters for preventing unicast loops and multicast implosions. The idea is straightforward, namely to use a Bloom filter in the packet header as a probabilistic loop detection mechanism. Each node has a corresponding mask that can be ORed with the Bloom filter in the header of a packet, and then determine whether or not a loop has occurred. Detection accuracy can be traded off against space required in the packet header [80].

3) Duplicate Detection: In [41], Deng and Rafiei introduce the Stable Bloom filter (SBF), which is a modified Counting Bloom Filter. In the update process, p randomly chosen counter values are decremented by 1, and then the k counters of the added element are set to Max, the maximum counter value. This causes a probabilistic aging of counters and eventual convergence of the fpr. This also results in false negatives. The authors use the SBF in stream duplicate detection, and achieve an improved false positive rate as compared to a regular Bloom filter, and an improved false negative rate compared to simple buffering.

Decaying Bloom filters (DBF) developed in [38] can also be used for duplicate detection in an unbounded data stream. The DBF is a Counting Bloom filter, in which the k counters that map to a new element are set to W, the sliding window size, when adding. Before adding, all counters are decremented by one. The authors further improved the performance of the DBF by dividing the DBF into blocks (b_DBF) so that each addition only takes $m/T+k$ operations, where T is the number of blocks and m the number of counters. Unfortunately the authors examine the false positive ratio with a much smaller sliding window than in [41], so [38] and [41] are not directly comparable. However, DBF appears, by interpolation, to have a much lower false positive rate than SBF: less than 2% at 4096 bits, compared to SBF's 8.2% at 16384 bits. Furthermore, DBF does not suffer from false negatives.

4) Forwarding Engines: Bloom filters can also be used in multicast forwarding engines. A multicast packet is sent through a multicast tree. A multicast router maps an incoming multicast packet to outgoing interfaces based on the multicast address. Initially, Grönvall suggests an alternative multicast forwarding technique using Bloom filters [81]. In this technique, a router has a Bloom filter for each outgoing interface. The filters contain the addresses associated with the interfaces. When a multicast packet arrives on one interface, the Bloom filters of each outgoing interface are checked for matches. The packet is forwarded to all matching interfaces. This technique is interesting, because it does not store any addresses at the router; however, the addition and removal of multicast addresses requires that the Bloom filters are updated, e.g., using any BF variant supporting deletions.

A similar idea has been recently proposed for content-centric networks [82], where packet forwarding decisions may be based on a new identifier space for information objects (e.g., 256-bit flat labels) or novel forwarding identifiers. An abstract switching element can be built by querying in parallel a bank of Bloom filters, one for each possible port-out (physical and virtual). The evaluation of the SPSwitch in [82] argues for a simpler system design and enhanced flexibility by relying on a fingerprint-based d-left hash table. The unifying Bloom principle of information-centric networking applications is to reduce the state requirements and simplify multicast support by tolerating some overdeliveries due to false positives.

A similar tradeoff can be applied to enterprise and data center networks, where the scalability of the data plane becomes increasingly challenging with the growth of forwarding tables and link speeds. Simply building switches with larger amounts of faster memory is not appealing, since high-speed memory is both expensive and power hungry. Implementing hash tables in SRAM is not appealing either because it requires significant over-provisioning to ensure that all forwarding table entries fit. The BUFFALO architecture [83] proposes Bloom filters stored in a small SRAM to compress the information of the addresses associated with each outgoing link. Leveraging the flattening of IP addresses and the shortest-path routing, BUFFALO proposes a practical switch design that gracefully handles false positives without reducing the packet-forwarding rate, while guaranteeing that packets reach their destinations with bounded stretch with high probability. Routing changes are handled by dynamically adjusting the filter sizes based on Counting Bloom Filters stored in slow memory.

The other extreme approach to support multicast is to move state from the network elements to the packets themselves in form of Bloom filter-based representations of the multicast trees. This notion has been exploited by Ratnasamy et al. when revisiting IP multicast [84] and by Jokela et al. [85] to provide a scalable forwarding plane for publish/subscribe networks (See Fig. 13). While [84] insert the inter-domain AS path information into a 800-bit Bloom filter-based header (called shimheader), LIPSIN [85] departs from the IP internetworking model and handles link identifiers more generally, from network interfaces to virtual links spanning multiple hops. Link IDs take a Bloom filter form (i.e., m bits with only k bits set to 1) that can be ORed together to build a source-routing Bloom filter. Forwarding nodes maintain a small Link ID table whose entries are checked for presence in the routing BF to take the forwarding decision. In a typical WAN topology, using 256-bit BFs, multicast trees containing around 40 links can be constructed to reach in a stateless fashion up to 24 users while maintaining the false positive rate ($\approx 3\%$) and the associated forwarding efficiency within

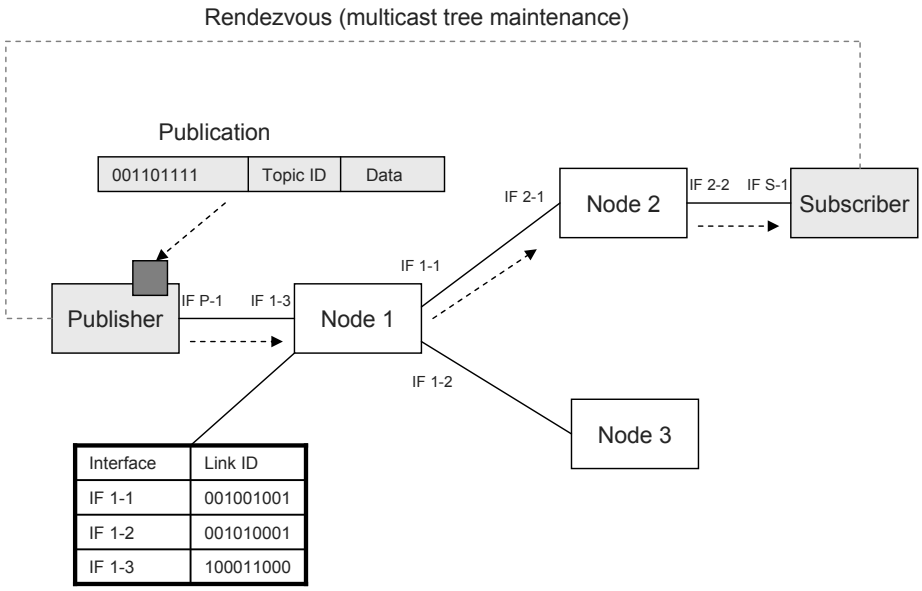

Fig. 13. Example of zFilter routing and forwarding

reasonable performance levels.

Applying the core idea of compressing source routes into packet headers, the Switching with in-packet Bloom filters (SiBF) architecture [86] proposes a Valiant load balanced forwarding service tailored for data center networks. Based on OpenFlow-capable switches, iBFs are carried in the Ethernet source and destination fields which are re-written at Top-of-Rack switches.

tian et al. have proposed an application-oriented multicast (aom) protocol [87]. each router uses the standard unicast ip routing table to determine necessary multicast copies and next-hop interfaces. all the multicast membership and addressing information traversing the network is encoded with bloom filters for low storage and bandwidth overhead. the paper goes on to prove that the aom service model is loop-free and incurs no redundant traffic. the false positive performance of the bloom filter implementation was also analyzed.

5) Deep Packet Scanning and Packet Classification: Bloom filters have found applications also in deep packet scanning, in which applications need to search for predefined patterns in packets at high speeds. Bloom filters can be used to detect predefined signatures in packet payloads. When a suspect packet is encountered, it can then be moved for further investigation. One advantage of Bloom filters is that they can be efficiently implemented in hardware and parallelized [88], [46], [89], which can result in high-performance and energy-efficient operation.

The storage requirements of the well-known crossproduct algorithm used in packet classification can be significantly reduced by using on-chip Bloom filters. For packets that match p rules in a rule set, a proposed algorithm requires $4 + p + e$ independent memory accesses to return all matching rules, where e is a small constant that depends on the false positive rate of the Bloom filters [90].

Packet classification continues to be an important challenge in network processing. It requires matching each packet against a database of rules and forwarding the packet according to the highest priority matching rule. Within the hash-based packet classification algorithms, an algorithm that is gaining interest is the tuple space search algorithm that groups the rules into a set of tuple spaces according to their prefix lengths. An incoming packet can now be matched to the rules in a group by taking into consideration only those prefixes specified by the tuples. More importantly, matching of an incoming packet can now be performed in parallel over all tuples. Within these tuple spaces, a drawback of utilizing hashing is that certain rules will be mapped to the same location, also called a collision. The negative effect of such a collision is that it will result in multiple memory accesses and subsequently longer processing time. The authors of [91] propose a pruned Counting Bloom Filter to reduce collisions in the tuple space packet classification algorithm. The approach decreases the number of collisions and memory accesses in the rule set hash table in comparison to a traditional hashing system. They investigate several well-known hashing functions and determine the number of collisions and show that utilizing the pruned Counting Bloom Filter can reduce the number of collisions at least 4% and by at most 32% for real rule sets.

6) Content-based Publish/Subscribe: The content-based publish-subscribe (pub-sub) paradigm for system design is becoming increasingly popular, offering unique benefits for many data-intensive applications. Coupled with peer-to-peer technology, it can serve as a central building block for developing data-dissemination applications deployed over a large-scale network infrastructure. A key open problem in creating large-scale content-based pub-sub infrastructures relates to

efficiently and accurately matching subscriptions with various predicates to incoming events [92], [93]. A Bloom filter-based approach has been proposed for general content-based routing with predicates [93].

Bloom filters and additional predicate indices were used in a mechanism to summarize subscriptions [94], [95]. An Arithmetic Attribute Constraint Summary (AACS) and a String Attribute Constraint Summary (SACS) were used to summarize constraints, because Bloom filters cannot directly capture the meaning of other operators than equality. The subscription summarization is similar to filter merging, but it is not transparent, because routers and servers need to be aware of the summarization mechanism. In addition, the set of attributes needs to be known a priori by all brokers and new operators require new summarization indices. The benefit of the summarization mechanism is improved efficiency, since a custom-matching algorithm is used that is based on Bloom filters and the additional indices.

D. Monitoring and Measurement

Network monitoring and measurement are key application areas for Bloom filters and their variants. We briefly examine some key cases in this domain, for example detection of heavy flows, Iceberg queries, packet attribution, and approximate state machines. Key functions for monitoring include flow classification [96], [97] and approximate counting and summarization of flows and packets [98], [99].

1) Heavy Flows: Bloom filters have found many applications in measurement of network traffic. One particular application is the detection of heavy flows in a router. Heavy flows can be detected with a relatively small amount of space and small number of operations per packet by hashing incoming packets into a variant of the counting Bloom filter and incrementing the counter at each set bit with the size of the packet. Then if the minimum counter exceeds some threshold value, the flow is marked as a heavy flow [100].

2) Iceberg Queries: Iceberg queries [101] have been an active area of research development. An Iceberg query is such that identifies all items with frequency above some given threshold. Bloom filter variants that are able to count elements are good candidate structures for supporting Iceberg queries. In networking, low-memory approximate histogram structures are needed for collecting network statistics at runtime. For example, in some applications it is necessary to track flows across domains and perform, to name a few examples, congestion and security monitoring. Iceberg queries can be used to detect Denial-of-Service attacks.

Packet and payload attribution is another application area in measurement for Bloom filters. The problem in payload attribution is as follows. Given a payload, the system reduces the uncertainty that we have about the actual source and destination(s) of the payload, within a given target time interval. The goodness of the system is directly related with how much this uncertainty can be reduced. The implementation of a payload attribution system has two key components, namely a payload processing component and a query-processing component.

3) Packet Attribution: The current Internet architecture allows a malicious node to disguise its origin during denial-of-service attacks with IP spoofing. A well-known solution to identify these nodes is IP traceback. The main types of traceback techniques are (1) to mark each packet with partial path information probabilistically, and (2) to store packet digests in the form of Bloom filters at routers and reconstruct attack paths by checking neighboring routers iteratively.

The *Source Path Isolation Engine (SPIE)* [102] implements a packet attribution system, in which the system keeps track of incoming and outgoing packets at a router. Simply storing all the resulting information is not feasible. Therefore, Snoeren et al. proposed to use Bloom filters to reduce the state requirements. A Bloom filter stores a summary of packet information in a probabilistic way. One key observation is that each router maintains its own Bloom filters and thus their hash functions are independent.

A SPIE-capable router creates a packet digest for every packet it processes. The digest is based on the packet's non-mutable header fields and a prefix of first 8 bytes of the payload. These digests are then maintained by a network component for a predefined time.

When a security component, such as an intrusion detection system, detects that the network is under attack, it can use SPIE to trace the packet's route through the network to the sender. A single packet can be traced to its source given that the routers on the route still have the packet digest available. A false positive in this setting means that a packet is incorrectly reported as having been seen by a router. When the source of a packet is traced, false positives mean that the reverse path becomes a tree (essentially branches to multiple points due to false positives).

The packet attribution was extended to payload attribution by Shanmugasundaram et al. [31] with the Hierarchical Bloom filter. As discussed in this survey, this structure allows the query of a part of a string. SPIE uses the non-mutable headers and a prefix of the payload, whereas with Hierarchical Bloom filters it is sufficient to have only the payload to perform a traceback.

The key idea of the IP traceback in [103] is to sample only a small percentage (e.g., 3%) of the digests of the sampled packets. Relying on a low sampling rate is critical to relax the storage and computational requirements and allow link speeds to scale to OC-192 or higher rates.

The Generalized Bloom filter (GBF) [51], introduced in Sec. III-R, was conceived to address single-packet IP traceback in a stateless fashion by probabilistically encoding a packet's route into the packets themselves. The key feature of the GBF is the double set of hash functions to set and reset bits hop-by-hop, which provides built-in protection against Bloom filter tampering at the cost of some false negatives.

Counter braids [104] revisits the problem of accurate per-flow measurement. The authors present a counter architecture, called Counter Braids, inspired by sparse random graph codes. In a nutshell, Counter Braids "compresses while counting". It solves the central problems (counter space and flow-to-counter association) of per-flow measurement by "braiding" a hierarchy of counters with random graphs. Braiding results in

drastic space reduction by sharing counters among flows; and using random graphs generated on-the-fly with hash functions avoids the storage of flow-to-counter association.

While the problem of high-performance packet classification has received a great deal of attention in recent years, the research community has yet to develop algorithmic methods that can overcome the drawbacks of TCAM-based solutions. A hybrid approach, which partitions the filter set into subsets that are easy to search efficiently, is introduced in [105]. The partitioning strategy groups filters that are close to one another in tuple space, which makes it possible to use information from single-field lookups to limit the number of subsets that must be searched. Running time can be traded off against space consumption by adjusting the coarseness of the tuple space partition. The authors find that for two-dimensional filter sets, the method finds the best-matching filter with just four hash probes while limiting the memory space expansion factor to about 2. They also introduce a novel method for Longest Prefix Matching (LPM), which is used as a component of the overall packet classification algorithm. The LPM method uses a small amount of on-chip memory to speed up the search of an off-chip data structure, but uses significantly less on-chip memory than earlier methods based on Bloom filters.

4) Approximate State Machines: Efficient and compact state representation is needed in routers and other network devices, in which the number and behaviour of flows needs to be tracked. The *Approximate Concurrent State Machine (ACSM)* approach was motivated by the observation that network devices, such as NATs, firewalls, and application level gateways, keep more and more state regarding TCP connections [106]. The ACSM construction was proposed to track the simultaneous state of a large number of entities within a state machine. ACSMs can return false positives, false negatives, and 'do not know' answers. Their construction follows the Bloom filter principle and proposes a space-efficient fingerprint compressed d-left hash table design.

E. Security

The hashing nature of the Bloom filter makes it a natural fit for security applications. Spafford (1992) was perhaps the first person to use Bloom filters to support computer security. The OPUS system [107] uses a Bloom filter which efficiently encodes a wordlist containing poor password choices to help users choose strong passwords. Two years later, Manber and Wu [108] presented two extensions to enhance the Bloom-filter-based check for weak passwords.

The privacy-preserving secure Bloom filters by Bellovin and Cheswick [59], described in Sec. III-W, allows parties to perform searches against each other's document sets without revealing the specific details of the queries. The system supports query restrictions to limit the set of allowed queries.

Bloom filters have been used by Aguilera et al. [109] to detect hash tampering in a network-attached disks (NADs) infrastructure. Also in the field of forensic filesystem practices, the *md5bloom* manipulation tool [110] employs Bloom filters to efficiently aggregate and search hashing information, demonstrating its practicality of identifying object versioning in Linux libraries.

Moving over to the field of network security, Attig, Dharmapurikar and Lockwood [111] describe an FPGA implementation of an array of Bloom filters and a hash table used for string matching to scan malicious Internet packets. The system searches 25 Bloom filters with string signature lengths from 2 to 26 bytes in parallel. False positives are resolved by exact match search using the hash table. Matches generate UDP packets that notify the user, a monitoring process, or a network administrator.

Antichi et al. [112] used Counting Bloom Filters to detect TCP and IP fragmentation evasion attacks. Attack signatures were split to 3-byte substrings which were inserted into a CBF. One CBF per attack signature string per flow was used. Incoming fragmented packet data was then matched against the CBF's and attack substrings detected. Each substring detected was removed from the corresponding CBF. Corresponding full string matchers were also enabled when a substring was detected. When the CBF was empty to the degree α, the attack string was considered detected, and the full string matcher was used to check for false positives. In case the full string matcher detected the attack, the flow was blocked. The authors report a greater than 99% detection rate and false positive ratios of 1% or less.

Bloom filters are used in the Trickles stateless network stack and transport protocol for preventing replay attacks against servers. Two Bloom filters of identical size and using the same family of hash functions are used to simplify the periodic purge operation [113]. The counting variant (CBF) is used in [114] to provide a lightweight route verification mechanism that enables a router to discover route failures and inconsistencies between advertised Internet routes and the actual paths taken by the data.

Focusing on the distributed denial-of-service (DDoS) issues, Ballani et al. [115] were among the first to use in-network Bloom filters to pro-actively filter out attacks, allowing each host to explicitly declare to the network routing infrastructure what traffic it wants routed to it. In addition to performing the standard longest-prefix match before forwarding packets, a router performs a reachability check using Bloom filters. Similar in their reliance on Bloom filters, Phalanx [116] combines the notion of capabilities with a multi-path-aware overlay, implementing Bloom filters to reduce state requirements while still providing probabilistic guarantees for in-network security. Wang et al. [117] propose *congestion puzzles* to mitigate bandwidth-exhaustion attacks. Congested routers challenge clients to generate hashes that match certain criteria in order to obtain bandwidth. Basic Bloom filters are maintained at routers to detect duplicate solutions.

In [118], Wolf presents a mechanism where packet forwarding is dependent on credentials represented as a packet header size Bloom filter. Credentials are issued by en-route routers on flow initiation and later verified on a packet-basis. Also based on in-packet Bloom filters (iBF), the self-routing capabilities in [119] enhance the security properties of LIPSIN [85] by using iBFs as forwarding identifiers that act simultaneously as path designators, i.e. define which path the packet should take, and as capabilities, i.e. effectively allowing the forwarding nodes along the path to enforce a security policy where

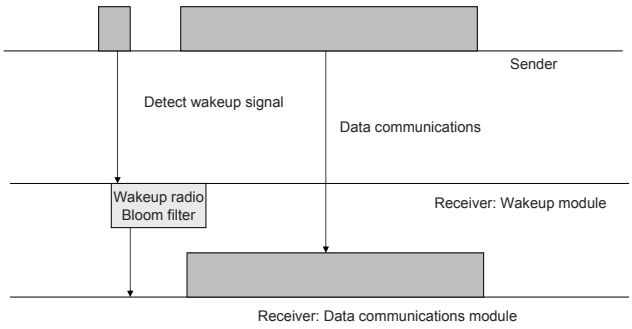

Fig. 14. Overview of device wakeup using a Bloom filter

only explicitly authorized packets are forwarded. Link IDs are dynamically computed at packet forwarding time using a loosely synchronized time-based shared secret and additional in-packet flow information (e.g., invariant packet contents). The capabilities are thus expirable and flow-dependent, but do not require any per-flow network state or memory look-ups, which are traded-off for additional, though amenable, per-packet computation.

In wireless sensor networks (WSNs), a typical attack by compromised sensor nodes consists of injecting large quantities of bogus sensing reports, which, if undetected, are forwarded to the data collector(s). The statistical en-route filtering approach [120] proposes a detection method based on a Bloom filter representation of the report generation (collection of keyed message authentications), that is verified probabilistically and dropped en-route in case of incorrectness. In order to address the problem of multiuser broadcast authentication in WSNs, Ren et al. [121] propose a neat integration of several cryptographic techniques, including Bloom filters, the partial message recovery signature scheme and the Merkle hash tree.

F. Other Applications

This section summarizes use of Bloom filters in several other interesting applications.

In web services, Counting Bloom Filters have been used for accelerated service discovery [122]. To manage a large number of services based on quantified service features, the features were stored in text form and mapped into the Bloom filter.

A Bloom filter-based wakeup mechanism has recently been proposed [123]. This work proposes an identifier-matching mechanism that uses a Bloom filter for wake-up wireless communication. The devices and services agree on wake-on wireless identifiers beforehand. The simulation results suggest that this approach can be used to reduce mobile device energy consumption. The identifier-matching mechanism can be implemented with a simple circuit using a Bloom filter, in which a query only uses an AND circuit. Figure 14 shows an overview of device wakeup using a Bloom filter.

The authors of [124] introduce a novel approximate method for XML data filtering, in which a group of Bloom filters represented a routing table entry and filtered packets according to XPath queries encoded to it. In this method, millions of path queries can be stored efficiently. At the same time, it is easy to deal with the change of these path queries. Performance is improved by using Prefix Filters to decrease the number of candidate paths. This Bloom filter-based method takes less time to build a routing table than an automaton-based method. The method has a good performance with acceptable fpr when filtering XML packets of relatively small depth with millions of path queries.

Achieving expressive and efficient content-based routing in publish/subscribe systems is a difficult problem. Traditional approaches prove to be either inefficient or severely limited in their expressiveness and flexibility. The authors of [93] present a novel routing method, based on Bloom filters, which shows high efficiency while simultaneously preserving the flexibility of content-based schemes. The resulting implementation is a fast, flexible and fully decoupled content-based publish/subscribe system.

As pervasive computing environments become popular, RFID tags are introduced into our daily life. However, there exists a privacy problem that an adversary can trace users' behavior by linking the tag's ID. Although a hash-chain scheme can solve this privacy problem, the scheme needs a long identification time or a large amount of memory. The authors of [125] propose an efficient identification scheme using Bloom filters. Their Bloom pre-calculation scheme provides high-speed identification with a small amount of memory by storing pre-calculated outputs of the tags in Bloom filters.

The authors of [126] propose a simple but elegant modification to the Bloom filter algorithm for hardware implementations that uses banking combined with special hash functions that guarantee all hash indexes fall into non-conflicting banks. They evaluate several applications of this Banked Bloom filter (BBF) in prediction in processors: BBF branch prediction, BBF load hit/miss prediction, and BBF last-tag prediction. The BBF predictors can provide accurate predictions with substantially less cost than previous techniques.

A power management proxy for P2P applications used N sets of hash functions and picked the Bloom filter with the least 1 bits to improve the false positive rate [127]. The hash functions were generated from a seed hash using a RNG. The system was used to allow a smart NIC to answer peer queries, and the computer was only woken up for download and upload tasks to conserve energy.

Bloom filters have been used for differential file access in a DBMS [128]. The differential file, with updated records, would be accessed only when the record to fetch was contained in the Bloom filter, indicating that the record in the database is not up-to-date. Otherwise the system would know that the record has not been changed, and it is sufficient to read the record from the database.

Bloom filters were used in probabilistic finite state transition system verification in [129]. The authors optimize hash calculation by shortening the state name using hashing, and then re-hashing the resulting value to obtain the k Bloom filter indices. A Bloom filter allows all states to be kept in memory in a compact manner so that verification can proceed without swapping.

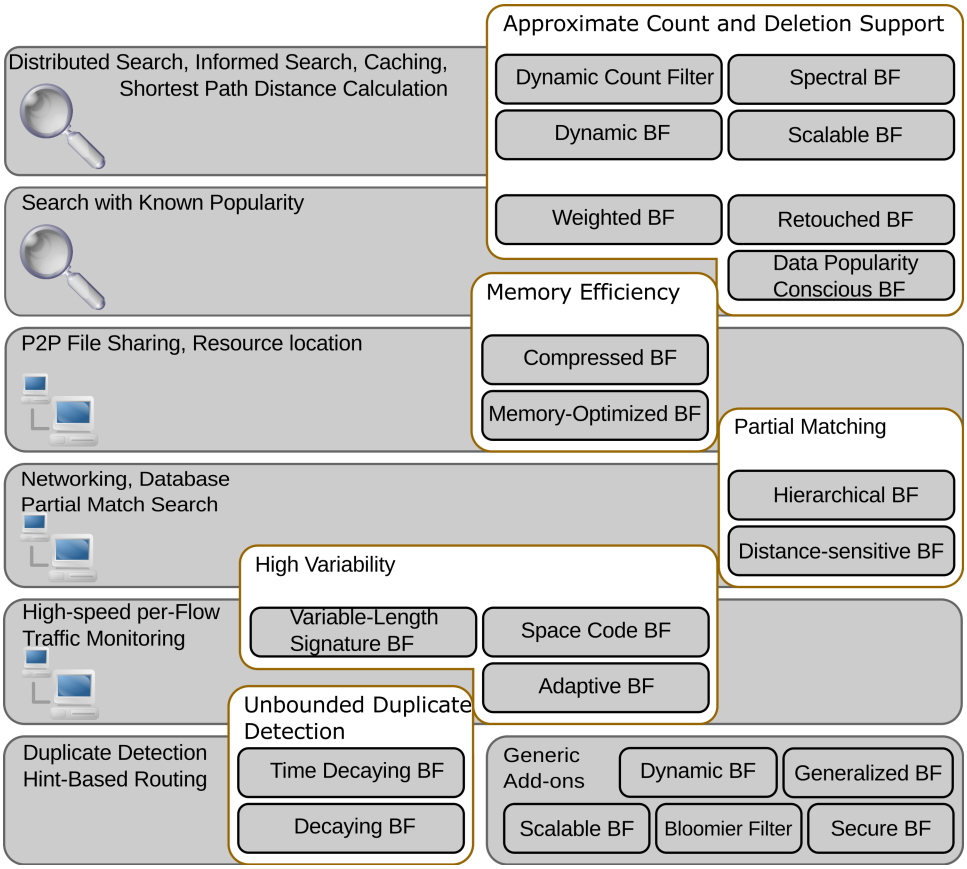

Fig. 15. Summary of Bloom filter variants

In [130], Bloom filters are used to represent and query ranges of multi-dimensional data. Range queries are handled by segmenting the attribute range into separate Bloom filters that represent membership in that segment.

V. Summary

Bloom filters are a general aid for network processing and improving the performance and scalability of distributed systems. In Figure 15, The Bloom filter variants introduced in this paper are categorized by application domain and supported features. The Figure aims to help domain experts select an appropriate Bloom filter based on their application. An expert need only find their domain on the left side and pick a Bloom filter on its right. Each rectangular bubble represents a Bloom filter variant. Variants that support a certain feature are found inside a highlighted area labeled with the name of that feature. *Approximate count and deletion support* refers to the ability to support approximate multiplicity and deletion of elements. The variants that support this are derived from the Counting Bloom Filter and include an array of fixed or variable size counters. *Memory efficiency* means that the variant optimizes the memory use of a Bloom filter in some fashion. These are recommended for applications in which memory is scarce. *Partial matching* means the ability to answer the question if x is near an element contained in the filter. These allow for example in-word matches for text search. *High variability* variants allow rapid changes in the set of items stored in the filter, such as those required by per-flow traffic monitoring. Finally, *Unbounded duplicate detection* is a class of Bloom filter that aims to represent a continuous stream of incoming elements and detect duplicate elements in the stream. The Figure also includes five variants that have been grouped into *General add-ons*. These Bloom filter techniques can be employed alone, or combined with another variant in the Figure. For example, many Bloom filters can be combined with *Scalable Bloom Filter* by increasing their length with a new block of space after the false positive ratio reaches a certain value.

Acknowledgements

This work was supported by TEKES as part of the Future Internet program of TIVIT (Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT).

References

[1] A. Z. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Mathematics*, vol. 1, no. 4, 2003.
[2] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[3] A. Ostlin and R. Pagh, "Uniform hashing in constant time and linear space," in *STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 2003, pp. 622–628.
[4] C. Henke, C. Schmoll, and T. Zseby, "Empirical evaluation of hash functions for multipoint measurements," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 39–50, 2008.

[5] A. Pagh, R. Pagh, and S. S. Rao, "An optimal Bloom filter replacement," in *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2005, pp. 823–829.

[6] E. Porat, "An optimal Bloom filter replacement based on matrix solving," in *CSR '09: Proceedings of the Fourth International Computer Science Symposium in Russia on Computer Science - Theory and Applications*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 263–273.

[7] M. Mitzenmacher, "Compressed Bloom filters," in *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2001, pp. 144–150.

[8] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and Y. Tang, "On the false-positive rate of Bloom filters," *Inf. Process. Lett.*, vol. 108, no. 4, pp. 210–213, 2008.

[9] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and network applications of dynamic Bloom filters," in *Proceedings of the 25th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, Apr 2006.

[10] L. F. Mackert and G. M. Lohman, "R* Optimizer Validation and Performance Evaluation for Distributed Queries," in *VLDB'86 Twelfth International Conference on Very Large Data Bases*, aug 1986, pp. 149–159.

[11] G. Marsaglia and W. W. Tsang, "Some difficult-to-pass tests of randomness," *Journal of Statistical Software*, vol. 7, no. 3, pp. 37–51, 2002.

[12] G. Varghese, *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices (The Morgan Kaufmann Series in Networking)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[13] A. Kirsch, M. Mitzenmacher, and G. Varghese, *Algorithms for Next Generation Networks, Computer Communications and Networks*. Springer-Verlag, Feb 2010, ch. Hash-Based Techniques for High-Speed Packet Processing, pp. 181–218.

[14] J. L. Carter and M. N. Wegman, "Universal classes of hash functions (extended abstract)," in *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1977, pp. 106–112.

[15] G. Antichi, D. Ficara, S. Giordano, G. Procissi, and F. Vitucci, "Blooming trees for minimal perfect hashing," in *Proceedings of the Global Communications Conference (GLOBECOM)*. IEEE, Nov 2008, pp. 1567–1571.

[16] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: building a better Bloom filter," in *ESA'06: Proceedings of the 14th annual European symposium on Algorithms*. London, UK: Springer-Verlag, 2006, pp. 456–467.

[17] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, "Balanced allocations," *SIAM J. Comput.*, vol. 29, no. 1, pp. 180–200, 2000.

[18] B. Vöcking, "How asymmetry helps load balancing," *J. ACM*, vol. 50, no. 4, pp. 568–589, 2003.

[19] A. Z. Broder and A. R. Karlin, "Multilevel adaptive hashing," in *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1990, pp. 43–53.

[20] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An Improved Construction for Counting Bloom Filters," in *14th Annual European Symposium on Algorithms, LNCS 4168*, 2006, pp. 684–695.

[21] F. Bonomi, M. Mitzenmacher, R. Panigrah, S. Singh, and G. Varghese, "Bloom filters via d-left hashing and dynamic bit reassignment," in *44th Allerton Conference*, Sep 2006.

[22] S. Lumetta and M. Mitzenmacher, "Using the power of two choices to improve Bloom filters," *Internet Mathematics*, vol. 4, no. 1, pp. 17–33, 2007.

[23] F. Hao, M. Kodialam, and T. V. Lakshman, "Building high accuracy Bloom filters using partitioned hashing," in *SIGMETRICS '07*. New York, NY, USA: ACM, 2007, pp. 277–288.

[24] M. V. Ramakrishna, "Practical performance of Bloom filters and parallel free-text searching," *Commun. ACM*, vol. 32, no. 10, pp. 1237–1239, 1989.

[25] M. Mitzenmacher and S. Vadhan, "Why simple hash functions work: exploiting the entropy in a data stream," in *SODA '08: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2008, pp. 746–755.

[26] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A linear-time probabilistic counting algorithm for database applications," *ACM Trans. Database Syst.*, vol. 15, no. 2, pp. 208–229, 1990.

[27] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, pp. 254–265, 1998.

[28] ——, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, 2000.

[29] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci, "Multilayer compressed counting Bloom filters," in *Proceedings of the 27th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*. IEEE, 2008, pp. 311–315.

[30] C. E. Rothenberg, C. A. B. Macapuna, F. L. Verdi, and M. Magalhães, "The deletable Bloom filter: a new member of the Bloom family," *IEEE Communications Letters*, vol. 14, no. 6, pp. 557–559, June 2010. [Online]. Available: http://arxiv.org/abs/1005.0352

[31] K. Shanmugasundaram, H. Brönnimann, and N. Memon, "Payload attribution via hierarchical Bloom filters," in *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2004, pp. 31–41.

[32] S. Cohen and Y. Matias, "Spectral Bloom filters," in *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2003, pp. 241–252.

[33] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier filter: an efficient data structure for static support lookup tables," in *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2004, pp. 30–39.

[34] D. Charles and K. Chellapilla, "Bloomier filters: A second look," in *ESA '08: Proceedings of the 16th annual European symposium on Algorithms*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 259–270.

[35] M. Dietzfelbinger and R. Pagh, "Succinct data structures for retrieval and approximate membership (extended abstract)," in *ICALP '08: Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part I*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 385–396.

[36] L. Golab and M. T. Özsu, "Issues in data stream management," *SIGMOD Rec.*, vol. 32, no. 2, pp. 5–14, 2003.

[37] A. Arasu and G. S. Manku, "Approximate counts and quantiles over sliding windows," in *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. New York, NY, USA: ACM, 2004, pp. 286–296.

[38] H. Shen and Y. Zhang, "Improved approximate detection of duplicates for data streams over sliding windows," *J. Comput. Sci. Technol.*, vol. 23, no. 6, pp. 973–987, 2008.

[39] X. Li, J. Wu, and J. J. Xu, "Hint-based routing in wsns using scope decay Bloom filters," in *IWNAS '06: Proceedings of the 2006 International Workshop on Networking, Architecture, and Storages*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 111–118.

[40] K. Cheng, L. Xiang, M. Iwaihara, H. Xu, and M. M. Mohania, "Time-decaying Bloom filters for data streams with skewed distributions," in *RIDE '05: Proceedings of the 15th International Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 63–69.

[41] F. Deng and D. Rafiei, "Approximately detecting duplicates for streaming data using stable Bloom filters," in *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2006, pp. 25–36.

[42] A. Kumar, J. J. Xu, L. Li, and J. Wang, "Space-code Bloom filter for efficient traffic flow measurement," in *IMC '03: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. New York, NY, USA: ACM, 2003, pp. 167–172.

[43] Y. Matsumoto, H. Hazeyama, and Y. Kadobayashi, "Adaptive Bloom filter: A space-efficient counting algorithm for unpredictable network traffic," *IEICE Trans. Inf. Syst.*, vol. E91-D, no. 5, pp. 1292–1299, 2008. [Online]. Available: http://iplab.naist.jp/research/traceback/Matsumoto_IEICE-ED200805.pdf

[44] Y. Lu, B. Prabhakar, and F. Bonomi, "Bloom filters: Design innovations and novel applications," in *Proceedings of the Forty-Third Annual Allerton Conference*, sep 2005.

[45] M. Yoon, "Aging Bloom filter with two active buffers for dynamic sets," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 1, pp. 134–138, 2010.

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

[47] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison, "Scalable Bloom filters," *Inf. Process. Lett.*, vol. 101, no. 6, pp. 255–261, 2007.

[48] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The dynamic Bloom filters," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 1, pp. 120–133, 2010.

[49] M. Xiao, Y. Dai, and X. Li, "Split Bloom Filter," *Acta Electronica Sinica*, vol. 32, no. 2, pp. 241–245, 2004.

[50] B. Donnet, B. Baynat, and T. Friedman, "Retouched Bloom filters: allowing networked applications to trade off selected false positives against false negatives," in *CoNEXT '06: Proceedings of the 2nd international conference on Emerging networking experiments and technologies*. New York, NY, USA: ACM, 2006, pp. 1–12.

[51] R. P. Laufer, P. B. Velloso, D. d. O. Cunha, I. M. Moraes, M. D. D. Bicudo, M. D. D. Moreira, and O. C. M. B. Duarte, "Towards stateless single-packet IP traceback," in *LCN '07: Proceedings of the 32nd IEEE Conference on Local Computer Networks*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 548–555.

[52] A. Kirsch and M. Mitzenmacher, "Distance-sensitive Bloom filters," in *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments and the Third Workshop on Analytic Algorithmics and Combinatorics (Proceedings in Applied Mathematics)*. SIAM, 2006.

[53] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM*, vol. 51, no. 1, pp. 117–122, 2008.

[54] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 518–529.

[55] M. Zhong, P. Lu, K. Shen, and J. Seiferas, "Optimizing data popularity conscious Bloom filters," in *PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2008, pp. 355–364.

[56] M. Ahmadi and S. Wong, "A memory-optimized Bloom filter using an additional hashing function." in *Proceedings of the Global Communications Conference (GLOBECOM)*. IEEE, Nov 2008, pp. 2479–2483.

[57] J. Bruck, J. Gao, and A. Jiang, "Weighted Bloom filter," in *2006 IEEE International Symposium on Information Theory (ISIT'06)*, July 2006.

[58] E.-J. Goh, "Secure indexes," Cryptology ePrint Archive, Report 2003/216, 2003, http://eprint.iacr.org/2003/216/.

[59] S. M. Bellovin and W. R. Cheswick, "Privacy-enhanced searches using encrypted Bloom filters," Columbia University and AT&T, Tech. Rep. CUCS-034-07, 2004.

[60] R. Nojima and Y. Kadobayashi, "Cryptographically secure Bloom-filters," *Transactions on Data Privacy*, vol. 2, no. 2, pp. 131–139, 2009.

[61] P. Hurley and M. Waldvogel, "Bloom filters: One size fits all?" *Proceedings of the Annual IEEE Conference on Local Computer Networks (LCN)*, pp. 183–190, 2007.

[62] F. Putze, P. Sanders, and J. Singler, "Cache-, hash- and space-efficient Bloom filters," in *WEA'07: Proceedings of the 6th international conference on Experimental algorithms*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 108–121.

[63] R. Pagh, "Cuckoo hashing," in *Encyclopedia of Algorithms*, M.-Y. Kao, Ed. Springer, 2008.

[64] H. Cai, P. Ge, and J. Wang, "Applications of Bloom filters in peer-to-peer systems: Issues and questions," in *NAS '08: Proceedings of the 2008 International Conference on Networking, Architecture, and Storage*, Washington, DC, USA, 2008, pp. 97–103.

[65] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: a distributed storage system for structured data," in *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 15–15.

[66] D. Borthakur, *The Hadoop Distributed File System: Architecture and Design*, The Apache Software Foundation, 2007.

[67] J. Risson and T. Moors, "Survey of research towards robust peer-to-peer networks: search methods," *Comput. Netw.*, vol. 50, no. 17, pp. 3485–3521, 2006.

[68] H. Cai, P. Ge, and J. Wang, "Applications of Bloom filters in peer-to-peer systems: Issues and questions," in *NAS '08: Proceedings of the 2008 International Conference on Networking, Architecture, and Storage*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 97–103.

[69] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao, "OceanStore: an architecture for global-scale persistent storage," *SIGARCH Comput. Archit. News*, vol. 28, no. 5, pp. 190–201, 2000.

[70] H. Cai and J. Wang, "Exploiting geographical and temporal locality to boost search efficiency in peer-to-peer systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 10, pp. 1189–1203, 2006.

[71] A. Kumar, J. Xu, and E. W. Zegura, "Efficient and scalable query routing for unstructured peer-to-peer networks," in *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2005, pp. 1162–1173.

[72] D. Starobinski, A. Trachtenberg, and S. Agarwal, "Efficient pda synchronization," *IEEE Transactions on Mobile Computing*, vol. 2, no. 1, pp. 40–51, 2003.

[73] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost, "Informed content delivery across adaptive overlay networks," in *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2002, pp. 47–60.

[74] P. Reynolds and A. Vahdat, "Efficient peer-to-peer keyword searching," in *Middleware '03: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*. New York, NY, USA: Springer-Verlag New York, Inc., 2003, pp. 21–40.

[75] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. Reinders, M. R. van Steen, and H. J. Sips, "TRIBLER: a social-based peer-to-peer system," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 2, pp. 127–138, 2008.

[76] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended Bloom filter: an aid to network processing," in *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2005, pp. 181–192.

[77] L. Deri, "High-speed dynamic packet filtering," *J. Netw. Syst. Manage.*, vol. 15, no. 3, pp. 401–415, 2007.

[78] A. Z. Broder and M. Mitzenmacher, "Using multiple hash functions to improve IP lookups," in *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2001, pp. 1454–1463.

[79] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using Bloom filters," in *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2003, pp. 201–212.

[80] A. Whitaker and D. Wetherall, "Forwarding without Loops in Icarus," in *Proceedings of Open Architectures and Network Programming (OPENARCH)*, 2002, pp. 63–75.

[81] B. Grönvall, "Scalable multicast forwarding," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 1, pp. 68–68, 2002.

[82] C. Esteve, F. L. Verdi, and M. F. Magalhães, "Towards a new generation of information-oriented internetworking architectures," in *CoNEXT 08: Proceedings of the 4th international conference on Emerging networking experiments and technologies*. New York, NY, USA: ACM, 2008.

[83] M. Yu, A. Fabrikant, and J. Rexford, "BUFFALO: Bloom filter forwarding architecture for large organizations," in *CoNEXT '09: Proceedings of the 5th international conference on Emerging networking experiments and technologies*. New York, NY, USA: ACM, 2009, pp. 313–324.

[84] S. Ratnasamy, A. Ermolinskiy, and S. Shenker, "Revisiting IP multicast," in *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, Pisa, Italy, Sept. 2006.

[85] P. Jokela, A. Zahemszky, C. Esteve, S. Arianfar, and P. Nikander, "LIPSIN: Line speed Publish/Subscribe Inter-Networking," in *SIGCOMM '09: Proceedings of the 2009 conference on Applications, technologies, architectures, and protocols for computer communications*, Barcelona, Spain, August 2009.

[86] C. E. Rothenberg, C. Macapuna, F. Verdi, M. Magalhães, and A. Zahemszky, "Data center networking with in-packet Bloom filters," in *28th Brazilian Symposium on Computer Networks (SBRC)*, May 2010.

[87] X. Tian, Y. Cheng, and B. Liu, "Design of a scalable multicast scheme with an application-network cross-layer approach," *IEEE Transactions on Multimedia*, vol. 11, no. 6, pp. 1160–1169, 2009.

[88] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood, "Deep Packet Inspection using Parallel Bloom Filters," *IEEE Micro*, vol. 24, no. 1, pp. 52–61, 2004.

[89] Y. Chen and O. Oguntoyinbo, "Power efficient packet classification using cascaded Bloom filter and off-the-shelf ternary cam for wdm networks," *Comput. Commun.*, vol. 32, no. 2, pp. 349–356, 2009.

[90] S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood, "Fast packet classification using Bloom filters," in *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*. New York, NY, USA: ACM, 2006, pp. 61–70.

[91] M. Ahmadi and S. Wong, "Modified collision packet classification using counting Bloom filter in tuple space," in *PDCN'07: Proceedings of the 25th IASTED conference on parallel and distributed computing and networks*. Anaheim, CA, USA: ACTA Press, 2007, pp. 315–320.

[92] I. Aekaterinidis and P. Triantafillou, "Publish-subscribe information delivery with substring predicates," *IEEE Internet Computing*, vol. 11, no. 4, pp. 16–23, 2007.

[93] Z. Jerzak and C. Fetzer, "Bloom filter based routing for content-based publish/subscribe," in *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*. New York, NY, USA: ACM, 2008, pp. 71–81.

[94] P. Triantafillou and A. Economides, "Subscription summaries for scalability and efficiency in publish/subscribe systems," in *Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, J. Bacon, L. Fiege, R. Guerraoui, A. Jacobsen, and G. Mühl, Eds., 2002.

[95] ——, "Subscription summarization: A new paradigm for efficient publish/subscribe systems," in *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 562–571.

[96] A. Soule, K. Salamatian, N. Taft, R. Emilion, and K. Papagiannaki, "Flow classification by histograms: or how to go on safari in the internet," in *SIGMETRICS*, 2004, pp. 49–60.

[97] J. Aguilar-Saborit, P. Trancoso, V. Muntes-Mulero, and J. L. Larriba-Pey, "Dynamic adaptive data structures for monitoring data streams," *Data Knowledge Engineering*, vol. 66, no. 1, pp. 92–115, 2008.

[98] A. Cvetkovski, "An algorithm for approximate counting using limited memory resources," in *SIGMETRICS*, 2007, pp. 181–190.

[99] Q. Zhao, J. Xu, and Z. Liu, "Design of a novel statistics counter architecture with optimal space and time efficiency," in *SIGMETRICS '06/Performance '06: Proceedings of the joint international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 2006, pp. 323–334.

[100] W.-c. Feng, K. G. Shin, D. D. Kandlur, and D. Saha, "The BLUE active queue management algorithms," *IEEE/ACM Trans. Netw.*, vol. 10, no. 4, pp. 513–528, 2002.

[101] Q. G. Zhao, M. Ogihara, H. Wang, and J. J. Xu, "Finding global icebergs over distributed data sets," in *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. New York, NY, USA: ACM, 2006, pp. 298–307.

[102] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, B. Schwartz, S. T. Kent, and W. T. Strayer, "Single-packet IP traceback," *IEEE/ACM Trans. Netw.*, vol. 10, no. 6, pp. 721–734, 2002.

[103] M. Sung, J. Xu, J. Li, and L. Li, "Large-scale IP traceback in high-speed internet: practical techniques and information-theoretic foundation," *IEEE/ACM Trans. Netw.*, vol. 16, no. 6, pp. 1253–1266, 2008.

[104] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter braids: a novel counter architecture for per-flow measurement," in *SIGMETRICS*, 2008, pp. 121–132.

[105] H. Song, J. Turner, and S. Dharmapurikar, "Packet classification using coarse-grained tuple spaces," in *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*. New York, NY, USA: ACM, 2006, pp. 41–50.

[106] F. Bonomi, M. Mitzenmacher, R. Panigrah, S. Singh, and G. Varghese, "Beyond Bloom filters: from approximate membership checks to approximate state machines," in *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2006, pp. 315–326.

[107] E. H. Spafford, "OPUS: preventing weak password choices," *Comput. Secur.*, vol. 11, no. 3, pp. 273–278, 1992.

[108] U. Manber and S. Wu, "An algorithm for approximate membership checking with application to password security," *Inf. Process. Lett.*, vol. 50, no. 4, pp. 191–197, 1994.

[109] M. K. Aguilera, M. Ji, M. Lillibridge, J. MacCormick, E. Oertli, D. Andersen, M. Burrows, T. Mann, and C. A. Thekkath, "Block-level security for network-attached disks," in *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2003, pp. 159–174.

[110] V. Roussev, Y. Chen, T. Bourg, and G. G. R. III, "md5bloom: Forensic filesystem hashing revisited." *Digital Investigation*, vol. 3, no. Supplement-1, pp. 82–90, 2006.

[111] C. Dixon, T. Anderson, and A. Krishnamurthy, "Phalanx: withstanding multimillion-node botnets," in *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008, pp. 45–58.

[112] G. Antichi, D. Ficara, S. Giordano, G. Procissi, and F. Vitucci, "Counting Bloom filters for pattern matching and anti-evasion at the wire speed," *IEEE Network*, vol. 23, no. 1, pp. 30–35, 2009.

[113] A. Shieh, A. C. Myers, and E. G. Sirer, "A stateless approach to connection-oriented protocols," *ACM Trans. Comput. Syst.*, vol. 26, no. 3, pp. 1–50, 2008.

[114] E. L. Wong, P. Balasubramanian, L. Alvisi, M. G. Gouda, and V. Shmatikov, "Truth in advertising: lightweight verification of route integrity," in *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2007, pp. 147–156.

[115] H. Ballani, Y. Chawathe, S. Ratnasamy, T. Roscoe, and S. Shenker, "Off by default!" in *Proc. 4th ACM Workshop on Hot Topics in Networks (Hotnets-IV)*, College Park, MD, Nov. 2005.

[116] C. Dixon, T. Anderson, and A. Krishnamurthy, "Phalanx: withstanding multimillion-node botnets," in *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008, pp. 45–58.

[117] X. Wang and M. K. Reiter, "Mitigating bandwidth-exhaustion attacks using congestion puzzles," in *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2004, pp. 257–267.

[118] T. Wolf, "A credential-based data path architecture for assurable global networking," in *Proc. of IEEE MILCOM*, Orlando, FL, October 2007.

[119] C. E. Rothenberg, P. Jokela, P. Nikander, M. Sarela, and J. Ylitalo, "Self-routing denial-of-service resistant capabilities using in-packet Bloom filters," in *the 5th European Conference on Computer Network Defense (EC2ND)*, 2009, pp. 46–51.

[120] F. Ye, H. Luo, S. Lu, L. Zhang, and S. Member, "Statistical en-route filtering of injected false data in sensor networks," in *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2004, pp. 839–850.

[121] K. Ren, W. Lou, and Y. Zhang, "Multi-user broadcast authentication in wireless sensor networks," in *Proceedings of the Fourth Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, 2007, pp. 223–232.

[122] S. Cheng, C. K. Chang, and L.-J. Zhang, "An efficient service discovery algorithm for counting bloom filter-based service registry," in *ICWS '09: Proceedings of the 2009 IEEE International Conference on Web Services*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 157–164.

[123] T. Takiguchi, S. Saruwatari, T. Morito, S. Ishida, M. Minami, and M. Morikawa, "A novel wireless wake-up mechanism for energy-efficient ubiquitous networks," in *Proceedings of the 1st International Workshop on Green Communications (GreenComm'09)*, 2009.

[124] X. Gong, W. Qian, Y. Yan, and A. Zhou, "Bloom filter-based xml packets filtering for millions of path queries," in *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 890–901.

[125] Y. Nohara, S. Inoue, and H. Yasuura, "A secure high-speed identification scheme for rfid using bloom filters," in *ARES '08: Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 717–722.

[126] M. Breternitz, G. H. Loh, B. Black, J. Rupley, P. G. Sassone, W. Attrot, and Y. Wu, "A segmented bloom filter algorithm for efficient predictors," in *SBAC-PAD '08: Proceedings of the 2008 20th International Symposium on Computer Architecture and High Performance Computing*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 123–130.

[127] M. Jimeno, K. Christensen, and A. Roginsky, "A power management proxy with a new best-of-n Bloom filter design to reduce false positives," *IEEE Performance, Computing, and Communications Conference*, pp. 125–133, 2007.

[128] L. L. Gremillion, "Designing a Bloom filter for differential file access," *Commun. ACM*, vol. 25, no. 9, pp. 600–604, 1982.

[129] P. C. Dillinger and P. Manolios, "Bloom filters in probabilistic verification," in *Formal Methods in Computer-Aided Design (FMCAD)*, vol. 3312. Springer-Verlag Heidelberg, 2004, pp. 367–381.

[130] Y. Hua, D. Feng, and T. Xie, "Multi-dimensional range query for data management using Bloom filters," in *IEEE International Conference on Cluster Computing*. Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 428–433.