Identifying Frequent Flows in Large Datasets through Probabilistic Bloom Filters

Yanjun Yao, Sisi Xiong, Jilong Liao, Michael Berry, Hairong Qi, and Qing Cao
Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville, TN, US
Email: {yyao9, sxiong, jliao2, mberry, hqi, cao}@utk.edu

***Abstract*—In many network applications, accurate traffic measurement is critical for bandwidth management with QoS requirements, and detecting security threats such as DoS (Denial of Service) attacks. In such cases, traffic is usually modeled as a collection of flows, which are identified based on certain features such as IP address pairs. One central problem is to identify those "heavy hitter" flows, which account for a large percentage of total traffic, e.g., at least 0.1% of the link capacity. However, the challenge for this goal is that keeping an individual counter for each flow is too slow, costly, and non-scalable. In this paper, we describe a novel data structure called the Probabilistic Bloom Filter (PBF), which extends the classical bloom filter into the probabilistic direction, so that it can effectively identify heavy hitters. We analyze the performance, tradeoffs, and capacity of this data structure. Our study also investigates how to calibrate this data structure's parameters. We also develop two extensions of the basic form of the PBF for more flexible application needs. We use real network traces collected on a web query server and a backbone router to test the performance of the PBF, and demonstrate that this method can accurately keep track of all objects' frequencies, including websites and flows, so that heavy hitters can be identified with constant time computational complexity and low memory overhead.**

I. Introduction

In managing today's complex Internet backbones, accurate traffic monitoring and measurements are crucial for many applications, including short-term purposes such as security needs (e.g., detecting traffic hot-spots, intrusions, and cyber-attacks) and long-term traffic engineering purposes with various QoS requirements (e.g., rerouting common traffic, expanding the capacity for frequently chosen links) [1], [2], [3]. One central problem in such applications is to identify heavy hitters, i.e., those most frequent flows, by keeping track of flow frequencies based on real-time traffic. Given that the number of flows between commercial end host pairs can be extremely large [1], [4], however, keeping a counter for each flow usually requires more memory than available on limited hardware resources, such as routers. Existing methods have addressed this problem through sampling and counting, such as NetFlow [5], where one of N packets is sampled and counted. However, such methods can only sample a small portion of the entire traffic, leading to inaccurate results and over- or under-estimates.

Given such challenges, in this paper, we address the problem on how we can efficiently construct estimates for the frequencies of all traffic flows so that heavy hitters can be identified. To this end, we aim to build an approximate histogram of all traffic flows with limited memory space, so that we can easily identify whether a flow is "heavy" when it is encountered in the ongoing traffic. The key assumption in our approach is that, for traffic management purposes, approximate knowledge on flows' frequency is already sufficient to identify heavy flows, as long as the frequency estimates can provide reliable upper and lower bounds associated with their most likely values. On the other hand, for some flows, if we need to obtain their accurate frequencies, we can add a few extra counters to serve such needs separately.

Our work in this paper is enabled by extending a compact, hashing-based data structure called the *bloom filter*. A Bloom Filter (BF) is a data structure that is designed to answer a query on whether an element exists in a set. Its basic idea is to hash an element to k different locations in a bit array, and sets these locations to all 1s when inserting this element to the set. Being a randomized method, it allows for false positives, but the space savings often outweigh its drawbacks. BFs were originally introduced for database applications, but recently they have received great attention also in the networking area (see [6], [7] as two surveys).

Based on the bloom filter, we investigate how to store *frequency estimations*, rather than set memberships. Note that previous work has addressed the "accurate" version of this problem by proposing *counting bloom filters (CBF)* [8], [6], [7], where the bit vector is replaced by a vector of integer counters. The cost is that the CBF design usually consumes memory space that is one order of magnitude higher than the original BF. In this paper, as we are concerned with the constrained memory of devices such as routers, our extension is based on bit vectors rather than counter vectors. Specifically, we present a probabilistic version of the bloom filter and its operations, so that we can provide estimates of flow frequencies using a small amount of memory, based on which we can provide reliable identification of heavy hitter flows. Formally, we define the problem as follows: given a multi-set S, we would like to identify those items that appear for more than f times. Note that items may be provided in a stream, as is the case of traffic flows, where IP addresses in headers are used to denote flows.

The central idea of our design, by extending the classical bloom filter, is that it performs probabilistic counting operations. Therefore, we call this new data structure as the

978-1-4673-7113-1/15/$31.00 ©2015 IEEE

Probabilistic Bloom Filter (PBF). The key difference is that whenever an item is inserted, instead of flipping the hash locations from 0 to 1 deterministically, we introduce a new parameter p, ranging from 0 to 1, which is the probability to flip the bit. Such a paradigm shift does not need any extra memory space compared to the standard bloom filter. Hence, it is still highly compact and feasible to implement on memory-constrained devices. We then model the performance of the PBF rigorously through probabilistic analysis, and we outline our major contributions as follows:

- We present the Probabilistic Bloom Filter, which allows non-deterministic queries on item existence and frequency in datasets. We provide the PBF's APIs and demonstrate how they can be used by applications.
- We quantitatively study the performance of the PBF through analytical approaches, where we derive closed-form results regarding its capacity, overhead, and performance.
- We study the parameter selection of the PBF for estimation accuracy purpose.
- We develop two extensions of the basic form of the PBF, called the C-PBF and the T-PBF, for additional application needs.
- We evaluate the performance of the PBF with realistic Internet traffic datasets collected from a web query dataset and a backbone router for one hour of time to demonstrate its effectiveness.

While our evaluations are based on network-related datasets such as web query logs and traffic traces, the methods in this paper should be general enough to be applied to several domains. Indeed, counting the number of events based on their types in large-scale datasets is a widely used building block for data analysis needs. For example, the recent discovery of the Higgs boson using the LHC [9], [10] draws conclusions based on statistical analysis for collected particle collision events. Properly configured versions of PBFs can be applied for such purposes when TB-scale of streaming data need to be analyzed in nearly real-time under computational and memory constraints.

The remainder of this paper is organized as follows: We survey related work in Section II. The problem formulation and design are described in Section III. The extensions of the PBF are presented in Section IV, and the performance evaluation is given in Section V. We provide conclusions in Section VI.

II. Related Work

In this section, we describe the related work in three parts: first, the original Bloom Filter design, then, its variants, and finally, recent progress on traffic flow sampling and counting in Internet routers.

The bloom filter, which is proposed by Burton H. Bloom in 1970 [11], is a space-efficient randomized data structure that answers the question on whether an element is in a set. The accuracy of a bloom filter depends on the filter size m, the number of hash functions k, and the number of inserted elements n. The false negatives (an element is reported as not in a set when it is) never happens. Although originally conceived for database applications, the bloom filter recently has also received great attention in the networking area [6], [7], [12], [13], [14], [15], [16], [17], [18].

In its initial design, the BF did not address the issues of element duplicates, as it only considers simple sets. No matter how many times an item appears in a set, it is counted only once in the constructed BF. Counting Bloom Filters (CBFs) [8], [6], [7] have been designed to address this issue. They are based on the same idea as BFs, but they adopted fixed size counters (also called bins) instead of single bits in its vector design. When an item is inserted, the corresponding counters are increased, hence, duplicate information is maintained rather than lost. However, CBFs are different from our approach in that they are fundamentally deterministic, as they keep accurate counts of the number of duplicates for an item. Therefore, CBFs require memory overhead that is usually an order of magnitude higher than common BFs. In another approach, Kumar and others [19] proposed the Space-Code Bloom Filter (SCBF) (later extended to a multi-resolution version called the MRSCBF), which used a filter made up of a fixed number of groups of hash functions. During the insertion operation, one group of hash functions was randomly chosen for the element. For query, the number of groups containing the element was counted to estimate the frequency. However, as only open formulas were provided, estimating frequency was done by looking up a pre-computed table, which was very computationally intensive to be built. Such efforts differ in our work in that we present closed-form results on modeling the performance of the proposed data structures. In our evaluation, we compare with both the CBF and the MRSCBF as they are the state-of-the-art baselines.

In recent years, the bloom filter has been widely used in network measurements [1]. Estan and others [20] applied Counting Bloom Filters to traffic measurement problems inside routers. The approach was based on the simple idea that if the counter for a flow increases beyond a threshold, it should be considered as a frequent flow. Zhao and others [21] used distributed Bloom Filters to find local icebergs (items whose frequency is larger than a given threshold), and then estimated global icebergs in a central server. Finally, Liu and others [22] proposed the Reversible MultiLayer Hashed Counting Bloom Filter(RML-HCBF), whose hash functions select a set of consecutive bits from the original strings as hash values, so that it may find elephant flows (large and continuous flow) using the counter values and thresholds. In contrast, the PBF we propose is based on approximate counting methods rather than accurate ones, thus saving on the memory overhead and processing speed.

III. Probabilistic Bloom Filter

In this section, we describe the design of the probabilistic bloom filter (PBF). We first present its programming interfaces, followed by the operations between multiple PBFs, and finally, an analysis of its properties, capacity, and performance.

A. Programming Interfaces

There are several operations for PBFs, which we summarize as the following APIs:

- `CREATE`(m, k, p) //Create a new PBF based on configuration parameters
- `INSERT`(x, d) //Insert element x into a PBF d
- `FREQUENCY`(x, d) //Query on the frequency of x in a PBF d
- `JOIN`(d_1,d_2) //Join two PBFs d_1 and d_2 together
- `COMPRESS`(d) //Compress a PBF d into half of the original size

Algorithm 1 The PBF Insert Algorithm

```
procedure INSERT(x)                ▷ Insert operation
    for j = 1 → k do
        i ← h_j(x)
        random_i ← Uniform(0, 1)
        if random_i < p then
            B_i ← 1
        end if
    end for
end procedure
```

We first describe the insert operation, whose details are illustrated in Algorithm 1. For this operation, the primary change compared to a conventional bloom filter is that it uses a parameter p to decide whether to flip a bit from 0 to 1, when items are inserted. For example, as shown in Figure 1, we choose $m = 25, k = 3, p = 2/3$, when inserting an item e_1, only two bits are flipped due to the new parameter p. Note that as an optimization, we do not need to read the bit's value before we set it to 1, thereby reducing the number of memory accesses for the insert operation. For the frequency query algorithm, it will simply add up the number of 1s in the k bits as determined by the hashing functions, and uses statistical inference methods to obtain an approximate frequency of the data item in the data set. We will describe the details of this estimation operation in Section III-B.

For multiple PBFs, we provide two APIs, the joining and compression, which are facilitated by the bit-vector nature of PBFs. Given two multi-sets S_1 and S_2, suppose that they are represented by two PBFs, $B1$ and $B2$. We can calculate the PBF that represents the union set $S = S_1 \cup S_2$ by taking the OR of their PBFs: $B = B1 \vee B2$ assuming that the bit vector length m and the hash functions are identical. The merged filter B represents the aggregate frequency of an item belonging to S_1 or S_2 as belonging to the set S.

The second operation is compression. If the PBF size m is divisible by 2, the compression operation allows us to store the original multiset in a shorter bit vector. This can be achieved by bit-wise ORing the first and second halves of the PBF's bit vector together. To insert or query the new PBF, the range of the hashing functions also needs to be updated by applying the $mod(m/2)$ operation to their outputs.

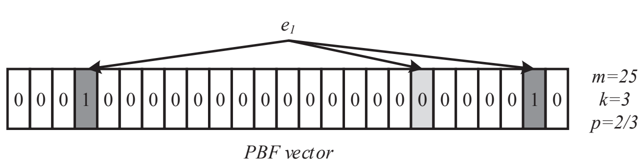

Fig. 1. Insert operation of PBF

TABLE I
SYMBOLS USED IN ANALYSIS

f	The frequency threshold of heavy-hitter items
k	The number of hashing functions
m	The length of the bit-vector
n	The total number of flows or items in a dataset
p	The probability for setting a bit to 1
y	The expected number of 1s out of k hashed positions
$\hat{y}$	The observed number of 1s out of k hashed positions
θ	The probability that a bit has been set to 1

B. Performance Modeling of PBFs

Because the PBF introduces one additional parameter p, it has different properties compared to the original BF. In this section, we model the performance of the PBF by studying the relationship between the frequency of items and the number of bits that are flipped out of k hashed positions. Table I shows the notations we use in the following analysis.

We first consider what happens if there is just one item. We assume that this item has hashed positions that are distributed uniformly, as is true for the hashing functions we choose in our implementation. In addition, among k hash functions, the results are independent from each other. We start from considering one hash function at a time. The probability that it flips a particular bit is p/m. Therefore, for a given bit, the probability that it is not set to 1 is given by $1 - \frac{p}{m}$. As there are k hashing functions for inserting any item, the probability that none of k hashed positions will set a specific bit to 1 is given by $(1 - \frac{p}{m})^k$. After inserting n items into the bloom filter, the probability that a given bit is still zero is going to be $P(n, 0) = (1 - \frac{p}{m})^{kn}$. Thus, the probability of a bit being set to 1 is

$$P(n, 1) = 1 - (1 - \frac{p}{m})^{kn}. \tag{1}$$

According to Equation 2, we could simplify most of our equations. Note that in following analysis, we consider the value of p is small and the value of n is large enough, as this is true in real applications since we are dealing with large amount of data.

$$\lim_{a \to \infty} (1 + \frac{x}{a})^a = e^x \tag{2}$$

The expected number of 1s of these k bits, denoted as $g(p, m, n, k)$, is

$$g(p, m, n, k) = (1 - (1 - \frac{p}{m})^{kn}) * k \approx (1 - e^{-\frac{kpn}{m}}) * k. \tag{3}$$

Note that this approximation for the expected value is true only when p/m is sufficiently small. This constraint is true because our picked m is usually large, and p is usually much smaller than 1. Observe that $P(n,1)$ exists for every bit regardless of what items are inserted. Therefore, this value corresponds to a "background noise", which means that some bits will be set due to other items being inserted. Notice that when m increases, the background noise will decrease. If n increases, the noise increases. To obtain the frequency estimation of items, we have to take this noise into account.

Next, for a certain item that appears f times, it will invoke the *insert* API f times. Therefore, the probability for any of the k bits mapped by hash functions to still be zero is $P(f,0) = (1-p)^f \approx e^{-pf}$, and the probability of this bit to be 1 is $P(f,1) = 1-(1-p)^f \approx 1-e^{-pf}$.

Again, we assume that $p \ll 1$, which is true for our selection of p values. Clearly, whether a bit is set to 1 is determined by two factors: the probability that it is set to 1 by an item's insert operation as illustrated by $P(f,1)$, and the probability of the background noise as illustrated by $P(n-f,1)$ (in Eq. 1). These two factors are independent of each other. Therefore, the probability for an element to remain 0 as (since neither the background noise nor the repeated items have set it to 1) is given by

$$P(n-f,0) \times P(f,0) \approx e^{-\frac{pk(n-f)}{m}} \times e^{-pf}. \tag{4}$$

Next, as each bit can be considered as a Bernoulli experiment, its "success" probability θ can be considered as the event that a bit has been set as 1. Here we denote that

$$\theta = 1 - P(n-f,0) \times P(f,0) \tag{5}$$

Therefore, denote the total number of 1s as Y, we have that

$$Y|\theta \sim Bin(k,\theta). \tag{6}$$

Therefore, we know that $E(Y|\theta) = k\theta$ and $V(Y|\theta) = k\theta(1-\theta)$. If we denote the expected number of bits that are set to 1 in the k mapped bits for an element in the PBF as y, we have

$$y = (1 - P(n-f,0) \times P(f,0)) \times k = k(1 - e^{-\frac{pk(n-f)}{m}} \times e^{-pf}). \tag{7}$$

Based on the observation results for the proportion of bits that have been set, we can denote its value as $\hat{y}$, and define $\hat{\theta} = \hat{y}/k$. We can use $\hat{\theta}$ as an unbiased estimator of θ, and since θ is derived based on the frequency f, we can then estimate f as follows (by solving the Equation 7 for f)

$$f = \frac{knp + m\ln\left[1 - \frac{\hat{y}}{k}\right]}{(k-m)p}. \tag{8}$$

Next, we calculate the confidence interval for f, by approximating it using a normal distribution based on the central limit theorem. This is also the so-called Wald Method, whose formula for confidence interval is given by

$$\hat{\theta} \pm z_{\frac{1}{2}\alpha}\sqrt{\frac{1}{n}\hat{\theta}\left(1-\hat{\theta}\right)} \tag{9}$$

where $\hat{\theta}$ is the proportion of successes, and $z_{\frac{1}{2}\alpha}$ is the critical z value with a tail area of $\frac{1}{2}\alpha$ of the standard normal curve. Based on this formula, we can derive the lower and upper bounds for f as shown below:

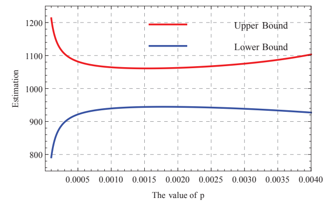

Fig. 2. Relation between p and the estimation bounds.

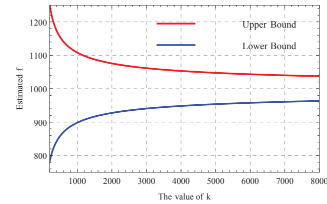

Fig. 3. Relation between k and the estimation bounds.

$$f_{min} = \frac{knp + m\ln\left[\frac{k-\hat{y}}{k} + \sqrt{\frac{\left(1-\frac{k-\hat{y}}{k}\right)(k-\hat{y})}{k^2}}\, z_{\frac{1}{2}\alpha}\right]}{(k-m)p} \tag{10}$$

$$f_{max} = \frac{knp + m\ln\left[\frac{k-\hat{y}}{k} - \sqrt{\frac{\left(1-\frac{k-\hat{y}}{k}\right)(k-\hat{y})}{k^2}}\, z_{\frac{1}{2}\alpha}\right]}{(k-m)p} \tag{11}$$

As an illustrative example, suppose we want to filter data traffic with $100K$ flows, where frequent flows are defined as those with a frequency to be at least one percent of the total, i.e., $1K$ flows. We pick a bloom filter size of $m = 2M$ bits ($1M = 10^6$), and let $k = 1000$. We select p as 0.0006 (we will explain this later in parameter selection). In this case, the frequent flow is expected to have 467 bits in 1000 bits set as 1. Conversely, if indeed, 467 bits are set, the estimated number of flows is 999, with a 95% confidence interval as $[905, 1098]$. On the other hand, if somehow the value of $\mu - 2\sigma = 435$ bits are set, the number of estimated flows is 902, with a 95% confidence interval as $[813, 995]$. Note that, however, this interval does not contain 1000 as it is a 95% confidence interval. One may want to use the 99% confidence interval to capture a larger range. In this case, the 99% interval gives the bound as $[797, 1013]$, which indeed contains the 1000 in its range.

C. Selection of Parameters for PBFs

One critical challenge in using the PBF for analyzing datasets is that it needs to set several parameters properly, such as m, k, and p. Choosing such parameters improperly will reduce its capabilities and increase errors. Further, due to the probabilistic nature of the PBF, its parameters need to be set differently compared to conventional bloom filters. Therefore, in this section, we study how to set the parameters for the PBF, and define its capacity.

Problem Formulation: Given a known number of items n and the threshold for frequent flows f, how do we choose m, k, and p properly? Similarly, given m, k, and p, how do we estimate the PBF's capacity to handle large n and f?

To answer this question, we first find the constraints for m, k, and p, and try to optimize the model performance by minimizing m and k, which correspond to the memory overhead (m) and computational overhead (k).

The first constraint for choosing the right parameters is to limit the background noise as shown in Equation 3. As m increases, the background noise will decrease, assuming a fixed k. Therefore, to keep the noise below a certain threshold ϵ, we require:

$$g(p, m, n, k)/k \leq \epsilon,$$

so that

$$m \geq \frac{-knp}{\log(1-\epsilon)}. \tag{12}$$

Note that we assume that ϵ is chosen as an appropriately low threshold, e.g., 0.1.

The second constraint concerns the estimation accuracy as shown in Equation 8. To ensure the estimation accuracy, the observed value of $\hat{y}$ should not be too small or too large compared to k. Since Equation 7 gives the expected value of y, we require that the ratio between y and k should lie between ϵ and $1-\epsilon$, therefore, $\epsilon \leq \frac{k(1-e^{-\frac{pk(n-f)}{m}} \times e^{-pf})}{k} \leq 1-\epsilon$.

Based on this result, we can then obtain the estimation range of f, which is denoted by the lowest f value and the highest f value that can be accurately estimated, as a function of k, m, and p as

$$f_{lowerbound} = \frac{knp + \log(1-\epsilon)m}{(k-m)p}, \tag{13}$$

$$f_{upperbound} = \frac{knp + \log(\epsilon)m}{(k-m)p}, \tag{14}$$

also, observe that the real value of f should be located within this estimation range, we have that

$$f_{lowerbound} \leq f \leq f_{upperbound}. \tag{15}$$

This formula, therefore, gives the third constraint. To illustrate the meanings of these constraints, especially on how they affect the choices of p, we consider the following way to illustrate possible choices of parameter values. To minimize m, we simply set m as the lower bound, using Equation 12, i.e., $m = \frac{-knp}{\log(1-\epsilon)}$. Then we can establish the constraints for p as

$$-\frac{\log(1-\epsilon)}{n} \leq p \leq \frac{(n-f)\log(1-\epsilon) - n\log(\epsilon)}{nf}. \tag{16}$$

On the other hand, the value of k can be chosen based on two considerations. First, if k is too large, it will incur too much computational overhead. Second, the value of k will affect the confidence interval calculated in Equations 10 and 11. The reason is that different values of k will lead to different lower and upper bounds, and their difference will be varying. To illustrate this, assume that we have chosen p as the upper bound, and m as the lower bound, and we set $\epsilon = 0.1$, we can calculate the ratio between the confidence interval of f to the estimated f as follows

$$Ratio = \frac{f_{max} - f_{min}}{f}.$$

Assuming $f \leq \epsilon n$, meaning that the frequency under estimation is not too large compared to the total number of items, n, we can find that

$$Ratio \approx 0.46 \ln\left(0.1 + 0.59\sqrt{\frac{1}{k}}\right) - 0.46 \ln\left(0.1 - 0.59\sqrt{\frac{1}{k}}\right).$$

We can verify that under this setting, this ratio decreases monotonically with k increases. In particular, if we let $Ratio \leq 1/2$, we can find that $k \geq 141$. Therefore, we suggest k must be chosen not lower than 150 to ensure good performance. Note that this value of k only incurs moderate computation overhead. Recent research on fast string hashing algorithms [23] has demonstrated optimized hashing functions can achieve a hashing throughput of a fraction of a CPU cycle per byte. Therefore, we can use a k value of larger than 1000 on multi-core workstations without having performance bottlenecks, as hashing a 100 byte string (e.g., a packet header) 1000 times only takes less than 0.1ms.

So far, we have outlined the way we should select parameters. To summarize, the procedure is as follows:

Algorithm 2 The PBF Parameter Selection Algorithm

```
1: procedure SELECT(n,f)                      ▷ n and f as known
2:     Calculate the bounds for p,
           and use its upper bound if possible (Equation 16)
3:     Select a modest value for k (assuming k ≥ 150)
4:     Calculate the lower bound for m (Equation 12)
5:     return p, k, and m
6: end procedure
```

Note that we require p to be chosen as its upper bound because a larger p will increase the accuracy of estimation, as long as this p value does not go beyond its upper bound. We now use a numerical example to illustrate possible choices for p, k, and m. Assume that $n = 100K$ and $f = 1000$. If $\epsilon = 0.1$, according to Equation 16, we have $1.05 \times 10^{-6} \leq p \leq 0.0022$. We now evaluate the effects of p. By keeping m bounded according to Equation 12 and $k = 2000$ (k can also take any other constant value), we change the value of p and study its effects on the confidence interval. The results are shown in Figure 2. As illustrated, a larger p indeed leads to better confidence intervals, as long as the value of p does not go over its upper bound. In fact, the narrowest confidence interval in Figure 2 is precisely when $p = 0.0022$. After p goes over this upper bound (specified by Equation 16), the confidence interval becomes larger again. The underlying reason for this interesting phenomenon is that when p is too large, it will over-saturates the bit vector too early, hence impairing the performance of estimations.

Next we investigate the effects of m. Suppose that we have chosen a moderate value of p for general cases, where p is chosen as 0.0006. We hope to see how k can affect confidence intervals, by changing k from 200 to 8000. As shown in Figure 3, a larger k will lead to better confidence intervals, at the cost of more computational overhead.

D. The Maximum Estimatable Frequency

Once the PBF parameters are chosen, given properly chosen p, k, and m, one critical problem is its estimation errors. Given that our goal of this paper is to identify heavy-hitters, we are most concerned about the errors that are caused by over-saturation of bit vectors, i.e., when the frequency of the flow is too high. Motivated by this observation, we next define the "Maximum Estimatable Frequency (MEF)", which is defined as follows:

Maximum Estimatable Frequency (MEF): Given a set of PBF parameters, what is the maximum value of f that it can still estimate accurately, where the k bit vector is at most saturated for $1 - \epsilon$?

To derive MEF, we use Equation 14, and replace m with the result from Equation 12. Therefore, the maximum value that f can reach is given by

$$f_{MEF} = \frac{n(\log(1-\epsilon) - \log(\epsilon))}{np + \log(1-\epsilon)}. \tag{17}$$

In reality, for specific PBF settings, the value of p is only a single value. Hence, if we set p as a constant, for a large n, we then have

$$\lim_{n \to +\infty} f_{MEF} = \frac{\log(1-\epsilon) - \log(\epsilon)}{p}.$$

Hence, this approximation can be used to estimate the real MEF that a PBF setting can handle. If $\epsilon = 0.1$, the capacity is roughly $2.2/p$. For example, if $p = 0.001$, this PBF can count up to cardinality of around $2,200$. For all items with a frequency above this threshold, the PBF can still report that they are *at least* $2,200$, but their real value is not reported due to bit vector saturations.

IV. EXTENSIONS OF THE PBF

To further expand the ability of PBF, we have developed two extensions, a counting PBF (C-PBF) and a time-decaying PBF (T-PBF). Due to page limit, we briefly discuss C-PBF and then provide more detailed descriptions of T-PBF.

A. C-PBF: Counting PBF Design

In this section, we introduce the C-PBF, a counting variant of the PBF that extends its capability with more memory usage. The idea of the C-PBF is simple: it replaces each bit in the PBF with a w-bit counter. Whenever an item is inserted, instead of deciding on whether flipping one bit from 0 to 1, it will determine with a probability of p whether to increase this w-bit counter. Formally, this updated algorithm is shown in Algorithm 3.

Algorithm 3 C-PBF Insert Algorithm

procedure INSERT(x) ▷ Insert operation
    for $j = 1 \to k$ **do**
        $i \leftarrow h_j(x)$
        $Counter_i \leftarrow B[i]$
        $random_i \leftarrow Uniform(0,1)$
        if $random_i < p$ **then**
            $Counter_i = Counter_i + 1$
        end if
    end for
end procedure

B. T-PBF: Time-decaying PBF Design

In some scenarios, large stream data would saturated the PBF vector as data continues to accumulate and the accuracy would be affected for long term estimation. Therefore, we propose the T-PBF, which allows it to *forget* those frequent items that appeared in the distant past.

Algorithm 4 T-PBF Decaying Algorithm

procedure DECAY(x) ▷ Decay operation
    for Every T seconds **do**
        for $j = 1 \to k$ **do**
            $i \leftarrow h_j(x)$
            if then$B[i] == 1$
                $random_i \leftarrow Uniform(0,1)$
                if $random_i < q$ **then**
                    $B[i] \leftarrow 0$
                end if
            end if
        end for
    end for
end procedure

The idea of the T-PBF is to introduce a new operation, decaying, which flips bits from 1 to 0 with a probability q over each time epoch T. This can be considered as an approximation for the *delete* operation. Formally, this updated algorithm is shown in Algorithm 4.

We now demonstrate the long term behavior of the T-PBF. For simplicity, we consider operations in epochs, and the decaying operation only occurs at the end of each epoch. Observe that for each bit, for each epoch, it may either start with bit 1 or 0. If it starts with 1, it may be flipped to 0 at the end of the epoch with a probability of q. However, if it starts with 0, it may be flipped to 1 first with a probability shown in Equation 7, and then flipped back to 0 with a probability of q. Therefore, we can use a discrete time Markov chain to describe these operations. In particular, because the transitions exhibit different probabilities at the beginning and the end of each epoch, we can model it with a time-inhomogeneous chain with a seasonal variation. In this case, we have that the seasonal period $d = 2$, and the transition probability as the following (assuming $n \geq 0$, while $2n$ and $2n+1$ stand for the index of epochs)

$$P(2n) = \begin{array}{c c} & \begin{array}{cc} 0 & 1 \end{array} \\ \begin{array}{c} 0 \\ 1 \end{array} & \left[\begin{array}{cc} 1-\alpha & \alpha \\ 0 & 1 \end{array} \right] \end{array}, \quad P(2n+1) = \begin{array}{c c} & \begin{array}{cc} 0 & 1 \end{array} \\ \begin{array}{c} 0 \\ 1 \end{array} & \left[\begin{array}{cc} 1 & 0 \\ \beta & 1-\beta \end{array} \right] \end{array},$$

where we have:

$$\alpha = 1 - e^{-\frac{pk(n-f)}{m}} \times e^{-pf}, \text{and} \tag{18}$$

$$\beta = q. \tag{19}$$

To analyze this seasonal chain, we can add a supplementary variable to create a homogeneous chain. The new chain contains four states: $A(0,a), B(0,b), C(1,a), D(1,b)$. Its tran-

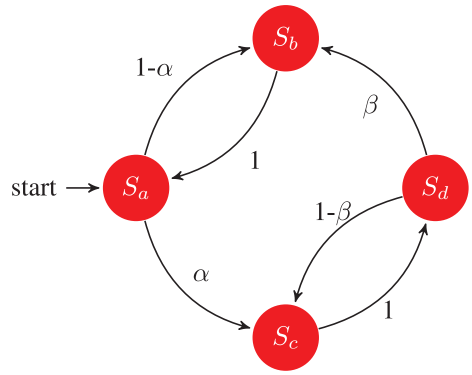

Fig. 4. The Markov Transition for the T-PBF

sition matrix is shown below, followed by the corresponding Markov chain illustration.

$$P(n) = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1-\alpha & \alpha & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \beta & 1-\beta & 0 \end{bmatrix} \end{array}$$

This Markov chain has four communicating classes, A, B, C, and D. They are all recurrent classes, as well. In the long term, this chain has the following stationary distribution:

$$P(s) = \begin{cases} \frac{\beta}{2\alpha+2\beta} & \text{for } s \equiv A, \\ \frac{\beta}{2\alpha+2\beta} & \text{for } s \equiv B, \\ \frac{\alpha}{2\alpha+2\beta} & \text{for } s \equiv C, \\ \frac{\alpha}{2\alpha+2\beta} & \text{for } s \equiv D. \end{cases}$$

As expected, the results show that in the long term, if a flow is no longer available, f will decrease to 0, and its α will quickly converge to $1 - e^{-\frac{pkn}{m}}$, which is smaller. Therefore, in the long run, most bits will be reset to 0 (as demonstrated by the increased probability of states A and B in Figure 4. This validates the design of the time-decaying nature of the T-PBF. On the other hand, if a flow re-appears with a large f, its transition will mostly stay in states C and D, which means that more bits will be set due to the flow existence. Note that the epoch of the T-PBF can also be dynamic: as the PBF will lose the ability of prediction when all k bits are set to 1s, when a large percentage of k bits for any element are set to 1, the decaying process can be triggered.

V. Evaluation

In this section, we evaluate the performance of the PBF, using one web query dataset and one real Internet traffic dataset. We compare the performance of our proposed algorithms with the following three baselines, in terms of estimation accuracy and memory usage: Counting Bloom Filter (CBF) [8], Multi-Resolution Space-Code Bloom Filter (MRSCBF) [19], and Random Sampled Netflow (RSN) [5].

A. Dataset A: Web Query Log Analysis

Our first evaluation dataset is a public web query dataset [24], where web query logs with $20M$ web queries are collected from $650K$ users over a period of three months. Each web query contains the user ID, searching keyword(s), a timestamp, and the web link that this user picks. The daily workload pattern is plotted in Figure 5, which shows a seven-day repeating pattern on the number of queries per day. In the following, we will use the PBF to detect popular websites.

Note that due to the limited size of the dataset, this task can be finished with any conventional method. However, our goal is to demonstrate that our proposed methods can achieve a better memory usage while introducing only limited errors. The limited size of this dataset allows us to evaluate the performance of these methods easily.

1) Detecting Popular Websites: The dataset includes a total of $378,087$ websites selected by users, where around 0.45% of them have a frequency above 100. We define the popular websites as those that appear for more than 100 times.

To detect popular websites, we first feed the dataset to a PBF with $k = 150$, $p = 0.001$, and $m = 6M$, where the parameters are selected based on the method in Section III-C. Note that the total memory use is only $0.75MB$ ($6M$ bits). Figure 6 compares the estimated frequency and the real frequency of the first 500 popular websites. Observe that the estimations are matching real frequencies closely, with an average estimation error computed using the formula $\frac{f_{estimated} - f_{real}}{f_{real}}$ as 4.7%. The primary source of inaccuracies comes from the randomness when flipping a bit from 0 to 1. Furthermore, note that Figure 6 shows the upper and lower bounds of f estimations, which have much larger errors compared to the estimated value of f (the estimate of f is calculated based on Equation 8).

We also test the same dataset with the C-PBF, and the performance is comparable. Specifically, for C-PBF, we set up $k = 50$, $p = 0.03$, $m = 0.6M$. The counter size of C-PBF is set to 10 bits, so that the total memory overhead is the same as PBF. The average estimation error of C-PBF is 4.9%, as shown in Figure 7. With this setting of C-PBF, the maximum frequency that can be estimated is $34,105$, which is sufficient for this dataset.

Next, we compare the performance of PBF/C-PBF with the baselines. The first baseline is CBF, where we set it up with suitable parameters. We choose to set the counter size to be 16 bits to accommodate the most frequent flows. Compared to the PBF, the CBF consumes *16 times of the memory* as the PBF and C-PBF under similar settings. The advantages of CBF lie in that it does not provide approximate answers, as demonstrated by the Figure 8, where no confidence intervals are plotted.

The second baseline is MRSCBF, where we set it up with $l = 32$, $r = 5$, and $m_i = \frac{k_i nl}{\ln 2}$, where $k = \{3, 4, 6, 6, 6\}$ as mentioned in [19]. As shown in Figure 9, the average estimation error of MRSCBF is 10.1%, which is twice of the error of PBF and C-PBF. On the other hand, besides a long and offline pre-computation phase to get the lookup table, the MRSCBF takes $\sum_{i=1}^{5} m_i = 436,371,391$ bits memory space, which is *72.7 times of the memory* as the PBF and C-PBF. The only advantage of MRSCBF is that once the lookup table is built, it takes a few CPU cycles to estimate the frequency.

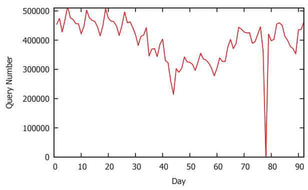

Fig. 5. The daily pattern for the web query dataset [24].

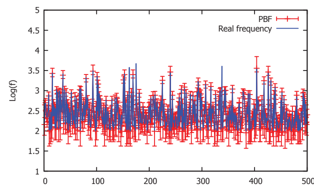

Fig. 6. Estimations of frequency results of the PBF.

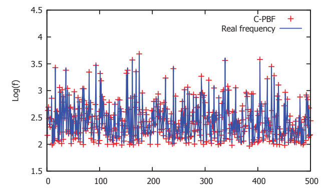

Fig. 7. Estimations of frequency results of the C-PBF.

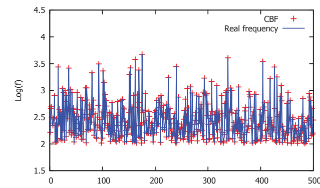

Fig. 8. Estimations of frequency results of the CBF.

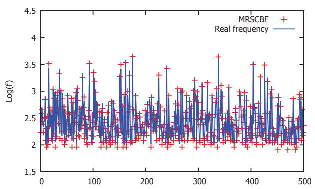

Fig. 9. Estimations of frequency results of the MRSCBF.

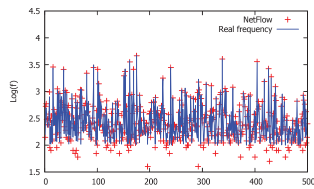

Fig. 10. Estimations of frequency results of the RSN.

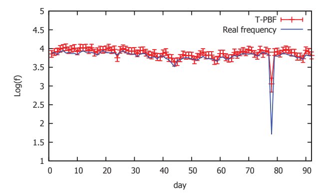

Fig. 11. Daily frequency trend estimation.

In PBF, however, a constant k hashing functions need to be calculated.

The third baseline is RSN, where we randomly sample one packet out of ten packets. As a result of the low sample rate, RSN only takes $11,063,808$ bits memory space, which is *1.8 times of the memory* as the PBF and C-PBF. According to Figure 10, the average estimation error of RSN is 11.3%. The major sources of errors are the randomness of sampling, and frequency estimation based on the sample counters and the predefined sample rate. Reduced estimation errors can be achieved, at the cost of taking more memory space. All the comparisons could also be inferred in Table II.

2) Frequency Trend of a Popular Website: For the long-term detection, we can use T-PBF to reveal the trend of frequency of a popular website. Here we choose Google as our observing target. For the T-PBF, we set up $k = 150$, $p = 0.001$, $m = 6M$, and $q = 0.8$. We trigger the decay operation every time we finish processing one day's data. As shown in Figure 11, T-PBF can estimate the frequency trend of the Google website well.

B. Dataset B: Network Measurement Dataset Analysis

In our second case study, we use PBF to analyze heavy-hitters from Internet traffic traces. Our evaluation dataset contains passive traffic traces from CAIDA's equinix-chicago and equinix-sanjose monitors on high-speed Internet backbone links [25]. The dataset spans one hour of activity. First, we analyze the general traffic pattern of the trace, by counting the total number of packets collected in each minute. The traffic

TABLE II
COMPARISON WITH BASELINES

Approaches	Average Error	Memory Usage
PBF	4.7%	0.75MB
CBF	0	12MB
MRSCBF	10.1%	54.55MB
RSN	11.3%	1.38MB

patterns are shown in Figure 12. In the following, we will use the PBF and the C-PBF to detect the heavy hitter flows, and use the T-PBF to detect their long-term trends.

To differentiate between flows, we use pairs of source and destination IP addresses as the key for each flow. We then count the frequency of each flow in the whole dataset, where the results are shown in Figure 13. Observe that most of the flows have a frequency of only once or twice, but a small number of flows exist with a large frequency. In our following experiments, we define a threshold of $1,000$ for those heavy hitters. These flows account for 0.12% of the total $96,854,555$ flows, where the maximum frequency is $32,404,064$.

1) Detecting Heavy Hitter Flows: First, we evaluate the estimation accuracy of the PBF. We focus on the data traces of the first 5 minutes when we apply PBF and C-PBF in this study. Later, we use T-PBF to handle the entire hour of data. The reason is that networked devices such as routers will most likely process datasets in a streaming manner, exactly as what T-PBF does, instead of processing the dataset in one operation.

For the first 5 minutes, there are $187,116,831$ packets, which belong to $15,454,076$ different flows. The number of flows with a frequency larger than $1,000$ is $13,569$. As there are limited resources on networking devices, we set $p = 0.00005$, $k = 4,000$, and $m = 800M$ bits, according to the method in Section III-C. The comparison between the real frequency and the estimated frequency of the first 500 frequent flows (sorted by time) are shown in Figure 14. Observe that the estimated frequency and the real frequency are almost perfect matching each other. The only exceptions are those cases when the PBF loses its estimation ability as it's k hash bits are saturated with 1s, leading it to miss certain data points. However, we argue that such missing points have no effects on the identification of heavy hitters, as these missing points correspond to heavy flows with a high certainty.

Next, we use the same first 5 minute data to evaluate the estimation accuracy of the C-PBF. To fit the C-PBF into networking devices, we set up the size of the counter to be 10 bits, so that $m = 80M$, $k = 400$, and $p = 0.0015$. As shown in Figure 15, the C-PBF can estimate the frequency of flows

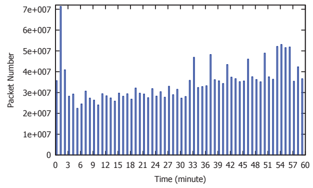

Fig. 12. Dataset traffic pattern, generated based on [25]

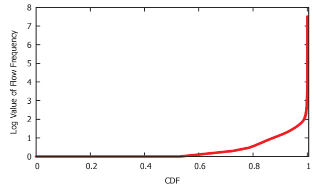

Fig. 13. Dataset flow frequency, generated based on [25]

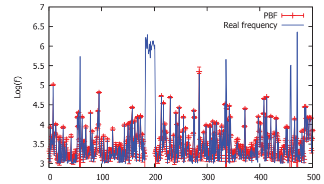

Fig. 14. Comparison between the real frequency and the estimated frequency with the PBF.

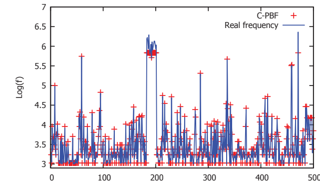

Fig. 15. Comparison between the real frequency and the estimated frequency with the C-PBF.

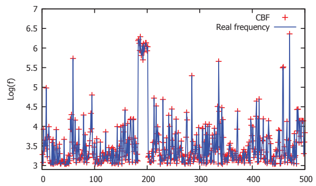

Fig. 16. Comparison between the real frequency and the estimated frequency with the CBF.

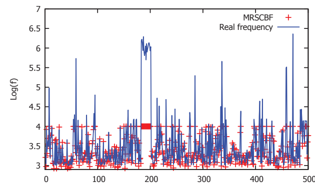

Fig. 17. Comparison between the real frequency and the estimated frequency with the MRSCBF.

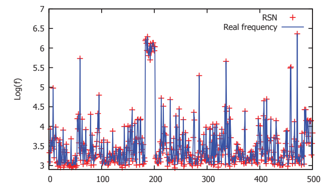

Fig. 18. Comparison between the real frequency and the estimated frequency with the RSN.

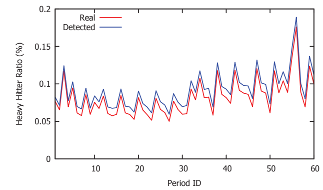

Fig. 19. Comparison of the real heavy hitter ratio and the detected heavy hitter ratio of the T-PBF.

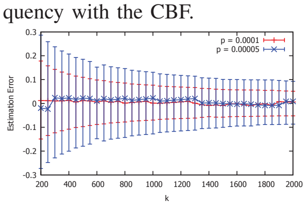

Fig. 20. The average error rate of the PBF with $p = 0.0001$, and different k and m values.

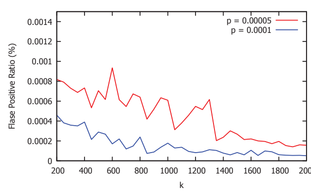

Fig. 21. The false positive ratio of the PBF.

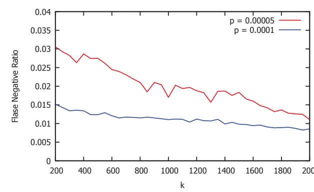

Fig. 22. The false negative ratio of the PBF.

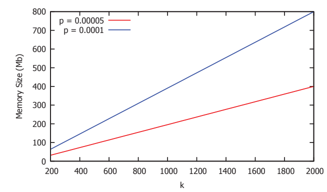

Fig. 23. The memory overhead of the PBF with different k values.

equally accurately compared to PBF. On average, the estimated frequency is 3.83% larger than the real frequency, caused by the inherent approximate nature of C-PBF counting designs. This accuracy is comparable to the original PBF design.

Next, we compare with the baselines including the CBF, the MRSCBF, and the RSN. For the CBF method, we set $k = 4000$ and $m = 800M$. As shown in Figure 16, the estimated frequencies are also close to the real frequencies, with an average error as 3.76%. However, to prevent counter overflows, we have to set each counter to occupy 22 bits, which means that in terms of memory overhead, the CBF takes *22 times of the memory* compared to PBF and C-PBF to deliver accurate counting performance. For the second baseline MRSCBF method, we set $l = 32$, $r = 6$, and $k = \{3, 4, 6, 6, 6, 6\}$ to accommodate the most frequent flows. To reduce the computation time of the look-up table, we set the maximum estimated frequency as $10,000$, which has no effects on detecting heavy hitter flows. This is why the estimated frequency is never larger than $10,000$ in the Figure 17. Excluding the flows with frequencies more than $10,000$, the average estimation error of MRSCBF is 9.3%, which is more than twice of the error of PBF and C-PBF. On the other hand, it needs $\sum_{i=1}^{6} m_i = 22,117,154,656$ bits of memory to hold this MRSCBF, which means that the MRSCBF takes *27.6 times of the memory* compared to PBF and C-PBF. For the RSN method, we set sample rate as 0.1. We can observe from Figure 18 that the average estimation error is 4.61%. On the other hand, RSN takes $1,304,986,970$ bits of memory space for this large dataset, which is *1.63 times of the memory*.

2) Long-term Heavy Flow Detection with T-PBF: We next investigate the performance of long-term flow detection with the T-PBF on the dataset. For the T-PBF, we set $p = 0.00005$, $k = 4000$, $m = 800M$, and $q = 0.3$. The decay operation is triggered after one minute's traffic data. We calculated the detected heavy hitter ratio (the number of detected heavy flows over the total number of flows), the false-positive ratio, and the false-negative ratio for each period. For both the false-positive ratio and the false-negative ratio, the threshold on frequency is set as 1000. As shown in Figure 19, the detected heavy hitter ratio on average is 6.48% larger than the real heavy hitter ratio. The source of this inaccuracy comes from the accumulation of flow frequencies in the previous periods. In general, the T-PBF works as expected, as we can observe that it triggers small estimation errors and low memory overhead.

3) Effects of Parameter Selections: We next perform experiments to evaluate the effects of k on estimation errors, the false-positive ratio, and the false-negative ratio. With a given k, the computational overhead and query delay is constant. In the experiments, we choose p to be 0.00005 and 0.0001, respectively, and change k from 200 to 2000 with a step size 50. The value of m is computed from Equation 12 with different k and p values. As shown in Figure 20, the estimation error decreases with the increasing k values. This observation is consistent with our theoretical evaluation shown in Figure 3.

On the other hand, as shown in Figure 21, the maximum false-positive ratio is 0.000009, which is acceptably small. The false negative ratio shown in Figure 22 has a maximum value of 0.03, which is also acceptably small. Note that the false negative ratio is larger than the false-positive ratio, because the number of heavy hitter flows is much smaller than the number of non-heavy-hitter flows. We also observe that by increasing k, the false-positive and false-negative ratios are decreasing. Finally, with the increasing of k and p, the needed memory size is increasing as shown in Figure 23. This explains the tradeoff between the error rates and the memory overhead: a larger memory overhead typically leads to a better performance.

VI. Conclusion

In this paper, we develop the probabilistic bloom filter (PBF), which extends conventional bloom filters to perform probabilistic counting operations. We provide the PBF's APIs to demonstrate how they can be used by applications, and quantitatively investigate the performance of the PBF through analytical approaches. The derived closed-form results answer our questions regarding the capacity, accuracy, and parameter selection of the PBF. Finally, we also extend the PBF into two variants: a counting PBF (C-PBF) and a time-decaying PBF (T-PBF), for additional application needs. Our evaluation results based on two realistic datasets show that this design outperforms the existing, state-of-the-art approaches.

To the best of our knowledge, our work in this paper is the first probabilistic bloom filter that is designed to count large volume of data with adjustable capacity and accuracy. We hope our work can stimulate future work in this direction, and provide a basis for investigations towards better methods based on probabilistic counting and bloom filters.

VII. Acknowledgment

We appreciate the highly insightful comments we received from Dr. Robinson E. Pino of the Department of Energy on the design of the algorithms in this paper. This work was supported by the National Science Foundation under Grant CNS-1239478 and Grant CNS-0953238.

References

[1] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang, "Leveraging Zipf S Law For Traffic Offloading," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 1, pp. 16–22, 2012.

[2] T. Li, S. Chen, and Y. Ling, "Per-Flow Traffic Measurement Through Randomized Counter Sharing," *IEEE/ACM Transactions on Networking*, vol. 20, no. 5, pp. 1622–1634, 2012.

[3] C. Estan and G. Varghese, "New Directions in Traffic Measurement and Accounting," in *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, vol. 32, no. 1, Jan. 2002, p. 75.

[4] C. Estan, G. Varghese, and M. Fisk, "Bitmap Algorithms for Counting Active Flows on High-Speed Links," *IEEE/ACM Transactions on Networking*, vol. 14, no. 5, pp. 925–937, Oct. 2006.

[5] Cisco, "Cisco IOS NetFlow," http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html.

[6] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2003.

[7] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and Practice of Bloom Filters for Distributed Systems," *IEEE Communications Surveys and Tutorials*, vol. 14, no. 1, pp. 131–155, 2012.

[8] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[9] ATLAS collaboration, "Observation of a New Particle in the Search for the Standard Model Higgs boson with the {ATLAS} Detector at the {LHC}," *Physics Letters B*, vol. 716, no. 1, pp. 1 – 29, 2012.

[10] CMS Collaboration, "Observation of a New Boson at a Mass of 125 GeV with the {CMS} Experiment at the {LHC}," *Physics Letters B*, vol. 716, no. 1, pp. 30 – 61, 2012.

[11] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[12] T. Chen, D. Guo, Y. He, H. Chen, X. Liu, and X. Luo, "A Bloom Filters Based Dissemination Protocol In Wireless Sensor Networks," *Journal of Ad Hoc Networks*, vol. 11, no. 4, pp. 1359–1371, 2013.

[13] O. Rottenstreich and I. Keslassy, "The Bloom Paradox: When Not To Use A Bloom Filter?" in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2012.

[14] B. Donnet, B. Gueye, and M. A. Kaafar, "Path Similarity Evaluation Using Bloom Filters," *Journal of Computer Networks*, vol. 56, no. 2, pp. 858–869, 2012.

[15] M. Moreira, R. Laufer, P. Velloso, and O. Duarte, "Capacity And Robustness Tradeoffs In Bloom Filters For Distributed Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2219–2230, 2012.

[16] H. Chen, H. Jin, X. Luo, Y. Liu, T. Gu, K. Chen, and L. M. Ni, "BloomCast: Efficient And Effective Full-Text Retrieval In Unstructured P2P Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 2, pp. 232–241, 2012.

[17] M. Sarela, C. E. Rothenberg, T. Aura, A. Zahemszky, P. Nikander, and J. Ott, "Forwarding Anomalies In Bloom Filter-based Multicast," in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2011.

[18] B. K. Debnath, S. Sengupta, J. Li, D. J. Lilja, and D. H.-C. Du, "BloomFlash: Bloom Filter On Flash-Based Storage," in *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2011.

[19] A. Kumar, J. Xu, and J. Wang, "Space-code bloom filter for efficient per-flow traffic measurement," *IEEE J.Sel. A. Commun.*, vol. 24, no. 12, pp. 2327–2339, Dec. 2006. [Online]. Available: http://dx.doi.org/10.1109/JSAC.2006.884032

[20] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 323–336, Aug. 2002.

[21] Q. G. Zhao, M. Ogihara, H. Wang, and J. J. Xu, "Finding global icebergs over distributed data sets," in *Proceedings of the PODS*. New York, NY, USA: ACM, 2006, pp. 298–307.

[22] W. Liu, W. Qu, Z. Liu, K. Li, and J. Gong, "Identifying elephant flows using a reversible multilayer hashed counting bloom filter," in *Proceedings of the HPCC-ICESS*, 2012, pp. 246–253.

[23] O. Kaser and D. Lemire, "Strongly Universal String Hashing is Fast," *CoRR-arXiv*, vol. abs/1202.4961, 2012.

[24] "500k user session collection," 2006. [Online]. Available: http://www.gregsadetsky.com/aol-data/

[25] "Internet trace data from caida," 2013. [Online]. Available: https://data.caida.org/datasets/passive-2013/