The Dynamic Bloom Filters

Deke Guo, *Member, IEEE,* Jie Wu, *Fellow, IEEE,* Honghui Chen, Ye Yuan, and Xueshan Luo

Abstract—A Bloom filter is an effective, space-efficient data structure for concisely representing a set and supporting approximate membership queries. Traditionally, the Bloom filter and its variants just focus on how to represent a static set and decrease the false positive probability to a sufficiently low level. By investigating mainstream applications based on the Bloom filter, we reveal that dynamic data sets are more common and important than static sets. However, existing variants of the Bloom filter cannot support dynamic data sets well. To address this issue, we propose dynamic Bloom filters to represent dynamic sets as well as static sets and design necessary item insertion, membership query, item deletion, and filter union algorithms. The dynamic Bloom filter can control the false positive probability at a low level by expanding its capacity as the set cardinality increases. Through comprehensive mathematical analysis, we show that the dynamic Bloom filter uses less expected memory than the Bloom filter when representing dynamic sets with an upper bound on set cardinality, and also that the dynamic Bloom filter is more stable than the Bloom filter due to infrequent reconstruction when addressing dynamic sets without an upper bound on set cardinality. Moreover, the analysis results hold in stand-alone applications as well as distributed applications.

Index Terms—Bloom filters, dynamic Bloom filters, information representation.

1 Introduction

Information representation and processing of membership queries are two associated issues that encompass the core problems in many computer applications. Representation means organizing information based on a given format and mechanism such that information is operable by a corresponding method. The processing of membership queries involves making decisions based on whether an item with a specific attribute value belongs to a given set. A standard Bloom filter (SBF) is a space-efficient data structure for representing a set and answering membership queries within a constant delay [1]. The space efficiency is achieved at the cost of false positives in membership queries, and for many applications, the space savings outweigh this drawback when the probability of an error is sufficiently low.

The SBF has been extensively used in many database applications [2], for example the Bloom join [3]. Recently, it has started receiving more widespread attention in networking literature [4]. An SBF can be used as a summarizing technique to aid global collaboration in peer-to-peer (P2P) networks [5], [6], [7], to support probabilistic algorithms for routing and locating resources [8], [9], [10], [11], and to share web cache information [12]. In addition, SBFs have great potential for representing a set in main memory [13] in stand-alone applications. For example, SBFs have been used to provide a probabilistic approach for explicit state model checking of finite-state transition systems [13], to summarize the contents of stream data in memory [14], [15], to store the states of flows in the onchip memory at networking devices [16], and to store the statistical values of tokens to speed up the statistical-based Bayesian filters [17].

The SBF has been modified and improved from different aspects for a variety of specific problems. The most important variations include compressed Bloom filters [18], counting Bloom filters [12], distance-sensitive Bloom filters [19], Bloom filters with two hash functions [20], space-code Bloom filters [21], spectral Bloom filters [22], generalized Bloom filters [23], Bloomier filters [24], and Bloom filters based on partitioned hashing [25]. Compressed Bloom filters can improve performance in terms of bandwidth-saving when an SBF is passed on as a message. Counter Bloom filters deal mainly with the item deletion operation. Distance-sensitive Bloom filters, using locality-sensitive hash functions, can answer queries of the form, "Is x close to an item of S?". Bloom filters with two hash functions use a standard technique in hashing to simplify the implementation of SBFs significantly. Space-code Bloom filters and spectral Bloom filters focus on multisets, which support queries of the form, "How many occurrences of an item are there in a given multiset?". The SBF and its mainstream variations are suitable for representing static sets whose cardinality is known prior to design and deployment.

Although the SBF and its variations have found suitable applications in different fields, the following three obstacles still lack suitable and practical solutions.

1) For stand-alone applications that know the upper bound on set cardinality for a dynamic set in advance, a large number of bits are allocated for an SBF to represent all possible items of the dynamic set at the outset. This approach diminishes the space-efficiency of the SBF, and should be replaced by new Bloom filters which always use an appro-

- *D. Guo, H. Chen, and X. Luo are with the Key laboratory of C4ISR Technology, School of Information Systems and Management, National University of Defense Technology, Changsha, China, 410073. E-mail: guodeke@gmail.com, chh0808@gmail.com, xsluo@nudt.edu.cn.*
- *J. Wu is with Department of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL 33431. E-mail: jie@cse.fau.edu.*
- *Y. Yuan is with Department of Computer Science, Northeastern University, Shen Yang, China, 110004. E-mail: linuxyy@gmail.com.*

priate number of bits as set cardinality changes.
2) For stand-alone applications that do not know the upper bound on set cardinality of a dynamic set in advance, it is difficult to accurately estimate a threshold of set size and assign optimal parameters to an SBF in advance. In the event that the cardinality of the dynamic set exceeds the estimated threshold gradually, the SBF might become unusable due to a high false positive probability.
3) For distributed applications, all nodes adopt the same configuration in an effort to guarantee the inter-operability of SBFs between nodes. In this case, all nodes are required to reconstruct their local SBFs once the set size of any node exceeds a threshold value at the cost of large (sometimes huge) overhead. In addition, this approach requires that the nodes with small sets must sacrifice more space so as to be in accordance with nodes with large sets, hence reducing the space-efficiency of SBFs and causing large transmission overhead.

The SBF and variants do not take dynamic sets into account. To address the three obstacles, we propose dynamic Bloom filters (DBF) to represent a dynamic set instead of rehashing the dynamic set into a new filter as the set size changes [26]. DBF can control the false positive probability at a low level by adjusting its capacity[1] as the set cardinality changes. We then compare the performances of SBF and DBF in three categories of stand-alone applications which feature two different types of sets: static sets with known cardinality and dynamic sets with or without an upper bound on cardinality. Moreover, we evaluate the performance of DBFs in distributed applications. The major advantages of DBF are summarized as follows:

1) In stand-alone applications, a DBF can enhance its capacity on demand via an item insertion operation. It can also control the false positive probability at an acceptable level as set cardinality increases. DBFs can shorten their capacities as the set cardinality decreases through item deletion and merge operations.
2) In distributed applications, DBFs always satisfy the requirement of inter-operability between nodes when handling dynamic sets, occupying a suitable amount of memory to avoid unnecessary waste and transmission overhead.
3) In stand-alone as well as distributed applications, DBFs use less expected memory than SBFs when dealing with dynamic sets that have an upper bound on set cardinality. DBFs are also more stable than SBFs due to infrequent reconstruction when dealing with dynamic sets that lack an upper bound on set cardinality.

The rest of this paper is organized as follows. Section 2 surveys standard Bloom filters and presents the algebra operations on them. Section 3 studies the concise representation and approximate membership queries of dynamic sets. Section 4 evaluates the performance of DBFs in stand-alone as well as distributed applications. Section 5 concludes this work.

1. The capacity of a filter is defined as the largest number of items which could be hashed into the filter such that the false match probability does not exceed a given upper bound.

2 Concise Representation and Membership Queries of Static Sets

2.1 Standard Bloom Filters

A Bloom filter for representing a set $X=\{x_1, ..., x_n\}$ of n items is described by a vector of m bits, initially all set to 0. A Bloom filter uses k independent hash functions $h_1, ..., h_k$ to map each item of X to a random number over a range $\{1, ..., m\}$ [1], [4] uniformly. For each item x of X, we define its Bloom filter address as $Bfaddress(x)$, consisting of $h_i(x)$ for $1 \leq i \leq k$, and the bits belonging to $Bfaddress(x)$ are set to 1 when inserting x. Once the set X is represented as a Bloom filter, to judge whether an element x belongs to X, one just needs to check whether all the $h_i(x)$ bits are set to 1. If so, then x is a member of X (however, there is a probability that this could be wrong). Otherwise, we assume that x is not a member of X. It is clear that a Bloom filter may yield a false positive due to hash collisions, for which it suggests that an element x is in X even though it is not. The reason is that all indexed bits were previously set to 1 by other items [1].

The probability of a false positive for an element not in the set can be calculated in a straightforward fashion, given our assumption that hash functions are perfectly random. Let p be the probability that a random bit of the Bloom filter is 0, and let n be the number of items that have been added to the Bloom filters. Then $p = (1 - 1/m)^{n \times k} \approx e^{-n \times k/m}$ as $n \times k$ bits are randomly selected, with probability $1/m$ in the process of adding each item. We use $f^{BF}_{m,k,n}$ to denote the false positive probability caused by the $(n+1)$th insertion, and we have the expression

$$f^{BF}_{m,k,n} = (1-p)^k \approx (1 - e^{-k \times n/m})^k. \quad (1)$$

In the remainder of this paper, the false positive probability is also called the *false match probability*. We can calculate the filter size and number of hash functions given the false match probability and the set cardinality according to formula (1). From [1], we know that the minimum value of $f^{BF}_{m,k,n}$ is $0.6185^{m/n}$ when $k = (m/n)\ln 2$. In practice, of course, k must be an integer, and smaller k might be preferred since that would reduce the amount of computation required.

For a static set, it is possible to know the whole set in advance and design a perfect hash function to avoid hash collisions. In reality, an SBF is usually used to represent dynamic sets as well as static sets. Therefore, it is impossible to know the whole set and design k perfect hash functions in advance. On the other hand, different perfect hash functions used by an SBF may

cause hash collisions. Thus, the perfect hash functions are not suitable for overcoming hash collisions in SBFs in theory as well as practice.

On the other hand, a static set is typically not allowed to perform data addition and deletion operations once it is represented by an SBF. Thus, the bit vectors of the SBF will stay the same over time, and then the SBF can correctly reflect the set. Therefore, the membership queries based on the SBF will not yield a false negative in this scenario. However, the SBF must commonly handle a dynamic set that is changing over time, with items being added and deleted.

In order to support the data deletion operation, an SBF hashes the item to be deleted and resets the corresponding bits to 0. It may, however, set a location to 0, which is also mapped by other items. In such a case, the SBF no longer correctly reflects the set and will produce false negative judgments with high probability. To address this problem, Fan *et all*. introduced counting Bloom filters (CBF) [12]. Each entry in the CBF is not a single bit but rather a small counter that consists of several bits. When an item is added, the corresponding counters are incremented; when an item is deleted, the respective counters are decremented. The experimental results and mathematical analysis show that four bits for each counter is large enough to avoid overflows [12].

2.2 Algebra Operations on Bloom Filters

We use two standard Bloom filters $BF(A)$ and $BF(B)$ as the representations of two different static sets A and B, respectively.

Definition 1: **(Union of standard Bloom filters)** Assume that $BF(A)$ and $BF(B)$ use the same m and hash functions. Then, the union of $BF(A)$ and $BF(B)$, denoted as $BF(C)$, can be represented by a logical *or* operation between their bit vectors.

Theorem 1: If $BF(A \cup B)$, $BF(A)$ and $BF(B)$ use the same m and hash functions, then $BF(A \cup B) = BF(A) \cup BF(B)$.

Proof: Assume that the number of hash functions is k. We choose an item y from set $A \cup B$ randomly, and y must also belong to set A or B. Bits $hash_i(y)$ of $BF(A \cup B)$ are set to 1 for $1 \leq i \leq k$, and at the same time, bits $hash_i(y)$ of $BF(A)$ or $BF(B)$ are set to 1, thus $BF(A)[hash_i(y)] \cup BF(B)[hash_i(y)]$ are also set to 1. On the other hand, we chose an item x from set A or B randomly, and said x also belongs to set $A \cup B$. Bits $hash_i(x)$ of $BF(A) \cup BF(B)$ are set to 1 for $1 \leq i \leq k$, and at the same time bits $hash_i(x)$ of $BF(A \cup B)$ are also set to 1. Thus, $BF(A \cup B)[i] = BF(A)[i] \cup BF(B)[i]$ for $1 \leq i \leq m$. Theorem 1 is proved to be true. □

Theorem 2: The false positive probability of $BF(A \cup B)$ is not less than that of both $BF(A)$ and $BF(B)$. At the same time, the false positive probability of $BF(A) \cup BF(B)$ is greater than or equal to that of $BF(A)$ as well as $BF(B)$.

Proof: Assume that the sizes of sets A, B, and $A \cup B$ are n_a, n_b, and n_{ab} respectively. According to (1), we can calculate the false positive probability for $BF(A)$, $BF(B)$, and $BF(A \cup B)$.

In fact, given the same k and m, formula (1) is a monotonically increasing function of n. It is true that $|A \cup B| \geq \max(|A|, |B|)$, thus n_{ab} is not less than n_a and n_b. We could infer that the false positive probability of $BF(A \cup B)$ is not less than that of $BF(A)$ and $BF(B)$. According to Theorem 1, we know that $BF(A \cup B) = BF(A) \cup BF(B)$, thus the false positive probability of $BF(A) \cup BF(B)$ is also not less than the value of $BF(A)$ as well as $BF(B)$. Theorem 2 is proved to be true. □

Definition 2: **(Intersection of Bloom filters)** Assume that $BF(A)$ and $BF(B)$ use the same m and hash functions. Then, the intersection $BF(A)$ and $BF(B)$, denoted as $BF(C)$, can be represented by a logical and operation between their bit vectors.

Theorem 3: 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 $(1 - 1/m)^{k^2 \times |A - A \cap B| \times |B - A \cap B|}$.

Proof: Assume the number of hash functions is k. We can derive (2) according to Definition 1, 2, and Theorem 1

$$BF(A) \cap BF(B) = (BF(A - A \cap B) \cap BF(B - A \cap B)) \cup BF(A \cap B). \quad (2)$$

In fact, the items of set $A \cap B$ contribute the same bits whose value is 1 to Bloom filters $BF(A \cap B)$ and $BF(A) \cap BF(B)$. According to (2), it is easy to derive that $BF(A) \cap BF(B)$ equals to $BF(A \cap B)$ only if $BF(A - A \cap B) \cap BF(B - A \cap B) = 0$.

For any item $z \in (B - A \cap B)$, the probability that bits $hash_1(z), \ldots, hash_k(z)$ of $BF(A - A \cap B)$ are 0 should be $p^k = (1 - 1/m)^{k^2 \times |A - A \cap B|}$. Thus, we can infer that the probability that $BF(B - A \cap B) \cap BF(A - A \cap B) = 0$ should be $(1 - 1/m)^{k^2 \times |A - A \cap B| \times |B - A \cap B|}$. Theorem 3 is true. □

2.3 Related Works

The most closely-related work is split Bloom filters [27]. They increase their capacity by allocating a fixed $s \times m$ bit matrix instead of an m-bit vector as used by the SBF to represent a set. A certain number of s filters, each with m bits, are employed and uniformly selected when inserting an item of the set. The false match probability increases as the set cardinality grows. An existing split Bloom filter must be reconstructed using a new bit matrix if the false match probability exceeds an upper bound. In practice, the split Bloom filters also need to estimate a threshold of set cardinality, and encounter the same problems faced by SBF. Although dynamic Bloom filters adopt a similar structure as split Bloom filters, they are different in the following aspects. First, split Bloom filters always consume $s \times m$ bits and waste too much memory before the set cardinality reaches $(m \times \ln 2)/k$, whereas dynamic Bloom filters allocate memory in an

incremental manner. Second, split Bloom filters do not support the data deletion operation, which is required in order to really support dynamic sets, whereas dynamic Bloom filters do support dynamic sets. Third, dynamic Bloom filters propose dedicated solutions for four different scenarios, as shown in Section 4.

Another related work is scalable Bloom filters [28] which address the same problem and adopt a similar solution proposed by DBFs [26]. A scalable Bloom filter also employs a series of SBFs in an incremental manner, but uses a different method to allocate memory for each SBF. It allocates $m \times a^{i-1}$ bits for its i^{th} SBF where a is a given positive integer and $1 \leq i \leq s$, while a DBF allocates m bits for each DBF. It achieves a lower false positive probability than a DBF that holds the same number of SBFs by using more memory, but suffers from a drawback due to the use of heterogeneous SBFs featuring different sizes and hash functions. It causes large overhead due to the need to calculate the Bloom filter address for items in each SBF when performing a membership query operation. In a DBF, the Bloom filter address of items in each SBF is the same. This makes it possible to optimize the storage and retrieval of a DBF by using the bit slice approach, as shown in Section IV-E. On the other hand, it also lacks an item deletion operation and solutions for dedicated application scenarios.

3 Concise Representation and Membership Queries of Dynamic Set

DBF focuses on addressing dynamic sets with changing cardinality rather than static sets, which were addressed by the previous version. It should be noted that DBFs can support static sets. Throughout this paper, an SBF is called *active* only if its false match probability does not reach a designed upper bound; otherwise it is called *full*. Let n_r be the number of items accommodated by an SBF. The n_r is equal to the capacity c for a full SBF, and less than c for an active SBF. In the rest of this paper, we use SBF to imply counting Bloom filters for the sake of supporting the item deletion operation.

3.1 Overview of Dynamic Bloom Filters

A DBF consists of s homogeneous SBFs. The initial value of s is one, and the initial SBF is active. The DBF only inserts items of a set into the active SBF and appends a new SBF as an active SBF when the previous active SBF becomes full. The first step to implement a DBF is initializing the following parameters: the upper bound on false match probability of the DBF, the largest value of s, the upper bound on false match probability of the SBF, the filter size m of the SBF, the capacity c of the SBF, and number of hash functions k of the SBF. As we will discuss further on in this paper, the approaches used to initialize these parameters are not identical in different scenarios. For more information, readers may refer to Section 4.

Algorithm 1 Insert (x)

Require: x is not null
1: $ActiveBF \leftarrow GetActiveStandardBF()$
2: **if** ActiveBF is null **then**
3: $ActiveBF \leftarrow CreateStandardBF(m, k)$
4: Add ActiveBF to this dynamic Bloom filter.
5: $s \leftarrow s + 1$
6: **for** $i = 1$ to k **do**
7: $ActiveBF[hash_i(x)] \leftarrow ActiveBF[hash_i(x)] + 1$
8: $ActiveBF.n_r \leftarrow ActiveBF.n_r + 1$

GetActiveStandardBF()
1: **for** $j = 1$ to s **do**
2: **if** $StandardBF_j.n_r < c$ **then**
3: Return $StandardBF_j$
4: Return null

Given a dynamic set X with n items, we will first show how a DBF is represented through a series of item insertion operations. Algorithm 1 contains the details regarding the process of the item insertion operation. It is clear that the DBF should first discover an active SBF when inserting an item x of X. If there are no active SBFs, the DBF creates a new SBF as an active SBF and increments s by one. The DBF inserts x into the active SBF, and increments n_r by one for the active SBF. If X does not decrease after deployment, only the last SBF of the DBF will be active, whereas the other SBFs are full. Otherwise, those full SBFs may become active if some items are removed from the set X.

It is convenient to represent X as a DBF by invoking Algorithm 1 repeatedly. After achieving the DBF, we can answer any set membership queries based on the DBF instead of X. The detailed process is illustrated in Algorithm 2, which uses an item x as input. If all the $hash_j(x)$ counters are set to a non-zero value for $1 \leq j \leq k$ in the first SBF, then the item x is a member of X. Otherwise, the DBF checks its second SBF, and so on. In summary, x is not a member of X if it is not found in all SBFs, and is a member of X if it is found in any SBF of the DBF.

Algorithm 2 Query (x)

Require: x is not null
1: **for** $i = 1$ to s **do**
2: $counter \leftarrow 0$
3: **for** $j = 1$ to k **do**
4: **if** $StandardBF_i[hash_j(x)] = 0$ **then**
5: break
6: **else**
7: $counter \leftarrow counter + 1$
8: **if** $counter = k$ **then**
9: Return true
10: Return false

If an item x is removed from X, the corresponding DBF must execute Algorithm 3 with x as the input in order to reflect X as consistently as possible. First of all, the DBF must identify the SBF in which all the $hash_j(x)$

Algorithm 3 Delete (x)

Require: x is not null
1: $index \leftarrow null$
2: $counter \leftarrow 0$
3: **for** $i = 1$ to s **do**
4: **if** BF[i].Query(x) **then**
5: $index \leftarrow i$
6: $counter \leftarrow counter + 1$
7: **if** $counter > 1$ **then**
8: break
9: **if** $counter = 1$ **then**
10: **for** $i = 1$ to k **do**
11: $BF[index][hash_i(x)] \leftarrow BF[index][hash_i(x)] - 1$
12: $BF[index].n_r \leftarrow BF[index].n_r - 1$
13: Merge()
14: Return true
15: **else**
16: Return false

Merge()
1: **for** $j = 1$ to s **do**
2: **if** $StandardBF_j.n < c$ **then**
3: **for** $k = j + 1$ to s **do**
4: **if** $StandardBF_j.n_r + StandardBF_k.n_r < c$ **then**
5: $StandardBF_j \leftarrow StandardBF_j \cup StandardBF_k$
6: $StandardBF_j.n_r + \leftarrow StandardBF_k.n_r$
7: Clear $StandardBF_k$ from the dynamic Bloom filter.
8: Break

counters are set to a non-zero for $1 \leq j \leq k$. If no SBF exists that satisfies the constraint in the DBF, the item deletion operation will be rejected since x does not belong to X. If there is only one SBF satisfying the constraint, the counters $hash_j(x)$ for $1 \leq j \leq k$ are decremented by one. If there are multiple SBFs satisfying the constraint, then x may appear to be in multiple SBFs of the DBF. Thus, it is impossible for the DBF to know which is the right one. If the DBF persists in removing membership information of x from it, the wrong SBF may perform the item deletion operation with given probability. The wrong item deletion operation destroys the DBF, and leads to at most k potential false negatives. To avoid producing false negatives, the membership information of such items is kept by the DBF but removed from X.

Furthermore, two active SBFs should be replaced by the union of them if the addition of their n_r is not greater than the capacity c of one SBF. The union operation of counting Bloom filters is similar to that of standard Bloom filters, which performs the addition operation between counter vectors instead of the logical *or* operation between bit vectors. Note that there is at most one pair of SBFs which satisfy the constraint of union operation after an item is removed from the DBF.

The average time complexity of adding an item x to an SBF and a DBF is the same: $O(k)$, where k is the number of hash functions used by them. The average time complexities of membership queries for SBF and DBF are $O(k)$ and $O(k \times s)$, respectively. The average time complexity of a member deletion for SBF and DBF are $O(k)$ and $O(k \times s)$, respectively.

3.2 False Match Probability of Dynamic Bloom Filters

In this subsection, we analyze the false positive probability of a DBF under two scenarios. Items of X are not allowed to be deleted from X in the first scenario, whereas they are allowed to in the second scenario.

As discussed above, a DBF with $s = \lceil n/c \rceil$ SBFs can represent a dynamic set X with n items. If we use the DBF instead of X to answer a membership query, we may meet a false match at a given probability. We will evaluate the probability with which the DBF yields a false positive judgment for an item x not in X. The reason is that all counters of $bfaddress(x)$ in any SBF might have been set to a non-zero value by items of X.

If the cardinality of X is not greater than the capacity of an SBF ($n \leq c$), then the false match probability of the DBF can be calculated according to formula (1) since the DBF is just an SBF. Otherwise, the false match probability of the DBF can be calculated in a straightforward way. The false positive probability of the first $s - 1$ SBFs is $f^{BF}_{m,k,c}$, and that of the last SBF is f^{BF}_{m,k,n_l} with $n_l = n - c \times \lfloor n/c \rfloor$. Then, the probability that not all counters of $bfaddress(x)$ in each SBF of the DBF are set to a non-zero value is $(1-f^{BF}_{m,k,c})^{\lfloor n/c \rfloor}(1-f^{BF}_{m,k,n_l})$. Thus, the probability that all the counters of $bfaddress(x)$ in at least one SBF of the DBF are set to a non-zero value can be denoted as

$$\begin{aligned} f^{DBF}_{m,k,c,n} &= 1 - (1 - f^{BF}_{m,k,c,c})^{\lfloor n/c \rfloor}(1 - f^{BF}_{m,k,c,n_l}) \\ &= 1 - (1 - (1 - e^{-k \times c/m})^k)^{\lfloor n/c \rfloor} \\ &\quad (1 - (1 - e^{-k \times (n - c \times \lfloor n/c \rfloor)/m})^k). \end{aligned} \quad (3)$$

In the following discussion, we will use DBF as well as SBF to represent X, and observe the change trend of $f^{DBF}_{m,k,c,n}$ and $f^{BF}_{m,k,n}$ as n increases continuously. For $1 \leq n \leq c$, the false positive probability of DBF equals that of SBF. In this case, the DBF becomes the SBF, and formula (3) also becomes formula (1). For $n > c$, the false positive probability of the DBF increases gradually with n, while that of the SBF increases quickly to a high value and then slowly increases to almost one. For example, when n reaches $10 \times c$, $f^{BF}_{m,k,10\times c}$ becomes about one hundred times larger than $f^{BF}_{m,k,c}$, but $f^{DBF}_{m,k,c,10\times c}$ is about ten times larger than $f^{BF}_{m,k,c}$. We can draw a conclusion from the formulas (1), (3), and Figure 1 that DBF scales better than SBF after the actual size of X exceeds the capacity of one SBF.

Furthermore, we use multiple DBFs and SBFs to represent X, and study the trend of $f^{BF}_{m,k,n}/f^{DBF}_{m,k,c,n}$ as the cardinality n of X increases continuously. In our experiments, we chose four kinds of DBFs using four SBFs different

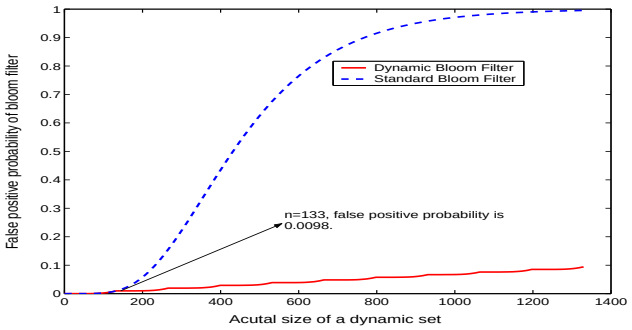


Fig. 1. False positive probability of dynamic and standard Bloom filters are functions of the actual size n of a dynamic set, where $m = 1280$, $k = 7$, and $c = 133$.

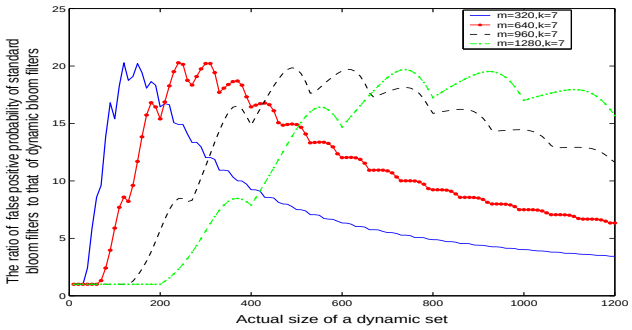


Fig. 2. The ratio of false positive probability of a standard Bloom filter to the value of a DBF is a function of the actual size n of a dynamic set, where $k = 7$ and $c = 133$.

with respect to size. For all four SBFs, the number of hash functions is 7, and the predefined upper bound on false positive probability is 0.0098. The experimental results are shown in Figure 2. It is obvious that all four curves follow a similar trend. The ratio $f^{BF}_{m,k,n}/f^{DBF}_{m,k,c,n}$ is a function of the actual size n of X. For $1 \leq n \leq c$, the ratio equals 1. For $n > c$, the ratio quickly reaches to the peak due to the slow increase in $f^{DBF}_{m,k,c,n}$ and the quick increase in $f^{BF}_{m,k,n}$, and then decreases slowly. After n exceeds c, the DBF with a different parameter m scales better than the corresponding SBF. In fact, the value of m has no effect on the trend of $f^{BF}_{m,k,n}/f^{DBF}_{m,k,c,n}$.

We know that $f^{BF}_{m,k,n}$ and $f^{DBF}_{m,k,c,n}$ are monotonically decreasing functions of m according to formulas (1) and (3). In other words, $f^{DBF}_{m_1,k,c,n} < f^{DBF}_{m_2,k,c,n}$ for $m_1 > m_2$, this means that the curve of $f^{DBF}_{m_1,k,c,n}$ is always lower than the curve of $f^{DBF}_{m_2,k,c,n}$ as n increases. In fact, so does $f^{BF}_{m,k,n}$. We also conduct experiments to confirm this conclusion, and illustrate the result in Figure 4.

3.3 Algebra Operations on Dynamic Bloom Filters

Given two different dynamic sets A and B, we can use two dynamic Bloom filters $DBF(A)$ and $DBF(B)$ or two standard Bloom filters $BF(A)$ and $BF(B)$ to represent them, respectively.

Definition 3: **(Union of dynamic Bloom filters)** Given the same SBF, we assume that $DBF(A)$ and $DBF(B)$ use $s_1 \times m$ and $s_2 \times m$ bit matrixes, respectively. $DBF(A) \cup DBF(B)$ could result in a $(s_1 + s_2) \times m$ bit matrix. The ith line vector equals the ith line vector of $DBF(A)$ for $1 \leq i \leq s_1$, and the $(i - s_1)$th line vector of $DBF(B)$ for $s_1 < i \leq (s_1 + s_2)$.

Theorem 4: The false positive probability of $DBF(A) \cup DBF(B)$ is larger than that of $DBF(A)$ as well as $DBF(B)$.

Proof: Assume that $DBF(A) \cup DBF(B)$, $DBF(A)$, and $DBF(B)$ uses the same SBF with parameters m, k, c, and the actual sizes of dynamic set A and B are n_a and n_b, respectively. The false positive probability of $DBF(A) \cup DBF(B)$ is

$$f^{DBF}_{m,k,c,n_a+n_b} = 1 - (1 - f^{BF}_{m,k,c})^{(\lfloor n_a/c \rfloor + \lfloor n_b/c \rfloor)} \times (1 - (1 - e^{-k \times (n_a - c \times \lfloor n_a/c \rfloor)/m})^k) \times (1 - (1 - e^{-k \times (n_b - c \times \lfloor n_b/c \rfloor)/m})^k). \quad (4)$$

The false positive probability of $DBF(A)$ and $DBF(B)$ are f^{DBF}_{m,k,c,n_a} and f^{DBF}_{m,k,c,n_b}, respectively. In fact, given the same k, m, and c, the value of formula (4) minus f^{DBF}_{m,k,c,n_a} is larger than 0, and the value of formula (4) minus f^{DBF}_{m,k,c,n_b} is also larger than 0. Thus, we can easily derive that the false positive probability of $DBF(A) \cup DBF(B)$ is larger than that of $DBF(A)$ as well as $DBF(B)$. □

Theorem 5: If the size of A and B is not zero and less than c, the false positive probability of $DBF(A) \cup DBF(B)$ is less than that value of $BF(A) \cup BF(B)$.

Proof: The false positive probability of $DBF(A) \cup DBF(B)$ is denoted as f^{DBF}_{m,k,c,n_a+n_b}, and that of $BF(A) \cup BF(B)$ is denoted as f^{BF}_{m,k,n_a+n_b}. Because the size of A and B is less than c, formula (4) can be simplified as formula (5). Let $x = e^{-k \times n_a/m}$ and $y = e^{-k \times n_b/m}$, we can obtain formula (6) which denotes f^{BF}_{m,k,n_a+n_b} minus f^{DBF}_{m,k,c,n_a+n_b} according to formula (1) and formula (5).

$$f^{DBF}_{m,k,c,n_a+n_b} = 1 - (1 - (1 - e^{-k \times n_a/m})^k)(1 - (1 - e^{-k \times n_b/m})^k) \quad (5)$$

$$f(x, y) = (1 - xy)^k + ((1 - x)(1 - y))^k - (1 - x)^k - (1 - y)^k \quad (6)$$

$$f(a) - f(d) = f(d)(a - d) + \ldots + f^{k-1}(d)(a - d)^{k-1}/(k - 1)! + f^k(\xi)(a - d)^k/k!, \; d < \xi < a \quad (7)$$

$$f(c) - f(b) = f(c)(c - b) + \ldots + f^{k-1}(b)(c - b)^{k-1}/(k - 1)! + f^k(\xi)(c - b)^k/k!, \; b < \xi < c \quad (8)$$

Let $a = 1 - xy$, $b = (1 - x) \times (1 - y)$, $c = 1 - x$, and $d = 1 - y$. Thus, $b < c < a$, $b < d < a$ because of $0 < x < 1$ and $0 < y < 1$. If $c < d$, then we obtain formulas (6) and (7) according to the Taylor formula. $f(z) = z^k, 0 < z < 1$, is a monotonically increasing function of z and has

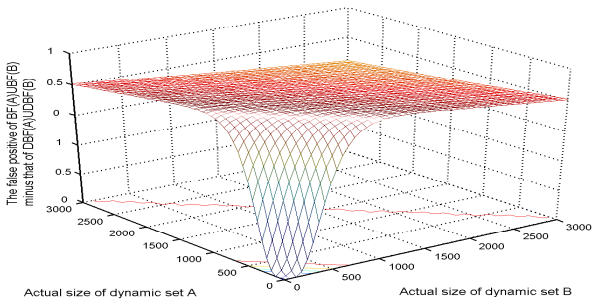


Fig. 3. False positive probability of $BF(A) \cup BF(B)$ minus that of $DBF(A) \cup DBF(B)$ is a function of size n_a of dynamic set A and n_b of set B, where $m = 1280$, $k = 7$, and $c = 133$.

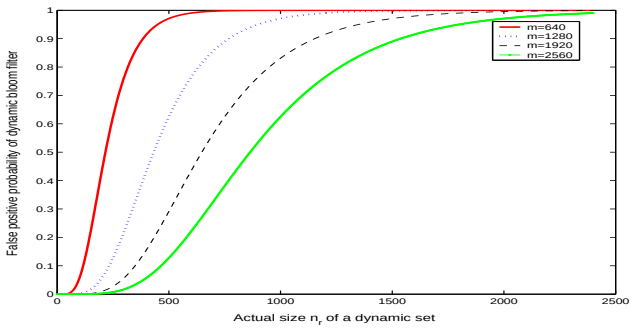


Fig. 4. The false positive probability of four kinds of DBFs are functions of the actual size n of a dynamic set, where $k = 7$, and the predefined threshold of false positive probability of each DBF is 0.0098.

a continuous k-rank derivative. The ith derivative is a monotonically increasing function for $1 < i \leq k$. It is obvious that $a - D = c - b, d < c < a, b < d < a$. Thus, each item of $f(a)$ is larger than the corresponding item of $f(c)$, and so (6) is larger than 0. If $c > d$, the result is the same. Theorem 5 is proved to be true. □

On the other hand, we used MATLAB to calculate the result of f^{BF}_{m,k,n_a+n_b} minus f^{DBF}_{m,k,c,n_a+n_b}. As shown in Figure 3, the false positive probability of $DBF(A) \cup DBF(B)$ is also less than that of $BF(A) \cup BF(B)$, even though the size of A and B exceeds c.

3.4 Evaluations of Item Deletion Algorithm

3.4.1 Mathematical Analysis

As mentioned above, multiple SBFs tend to allocate $Bfaddress(x)$ for an item $x \in X$ in a DBF. It is clear that only one SBF ever truly represented x during its input process; the other SBFs are false positives. The item deletion operation of DBF always omits such items, keeping their set membership information. The motivation is to prevent the DBF from producing potential false negatives caused by an incorrect item deletion. As a direct result of the item deletion operation, queries of such items will yield false positives.

Recall that X has n items, and the DBF uses $s = \lceil n/c \rceil$ SBFs. After the representation of X, we can consider the event where a particular item x of X appears to be in multiple SBFs. If x was represented by one of the first $s-1$ SBFs during the item insertion process, the probability of this event can be calculated by

$$f_1(n) = 1 - (1 - f^{BF}_{m,k,c})^{s-2}(1 - f^{BF}_{m,k,n_l}). \quad (9)$$

If item x was represented by the sth SBF during the item insertion process, this event means that at least one of the first $s-1$ SBFs produce a false positive judgment for x, and the probability of this event can be calculated by

$$f_2(n) = 1 - (1 - f^{BF}_{m,k,c})^{s-1}. \quad (10)$$

It is easy to know that the value of formula (10) is larger than that of formula (9). We use formula (10) as an estimated upper bound on the probability that x of X appears to be in multiple SBFs, and then achieve an estimated upper bound $n \times f_2(n)$ on the number of such items which have more than one Bloom filter address. Our experimental results show that the real number is less than the estimated upper bound. If all items of X are deleted, the DBF will try to perform the same operation. However, the DBF cannot guarantee deletion of all items due to its special item deletion operation. In reality, the DBF still holds the membership information of at most $n \times f_2(n)$ items. If other items join X at the same time that the original items are deleted, the DBF can reflect the membership information of all items of X and at most $n \times f_2(n)$ remaining items. It is logical that the false positive probability of the DBF is always larger than the theoretical value. Our experimental findings show similar results, and the difference between the real value and theoretical value is small. In other words, the negative impact of the item deletion operation on a DBF can be controlled at an accepted level.

3.4.2 Experimental Analysis

In this section, we will first describe the implementation of k random and independent hash functions. Then, we will compare the analytical model to the experimental results for the number of items which have multiple Bloom filter addresses. One critical factor in our experiments is creating a group of k hash functions. In our experiments, they will be generated as

$$h_i(x) = \big(g_1(x) + i \times g_2(x)\big) \bmod m. \quad (11)$$

$g_1(x)$ and $g_2(x)$ are two independent and random integers in the universe with range $\{1, 2, \ldots, m\}$. i ranges from 0 to $k-1$. We use the $SDBM\_MersenneTwister$ method to generate the two random integers for any item x. Let the output of the SDBM Hash function as the seed of the random number generator (RNG) MersenneTwister. Then, the MersenneTwister will produce the two desired random integers. The SDBM hash function seems to have a good over-all distribution for many different sets. It also works well in situations where there are high variations in the MSBs of the items in a set. The MersenneTwister is capable of quickly producing very

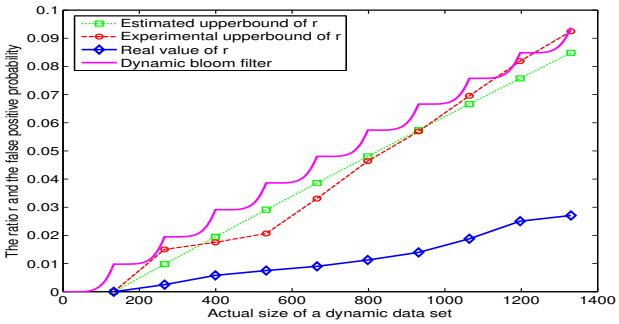


Fig. 5. False positive probability of dynamic Bloom filters and the percentage of data which has multiple Bloom filter addresses, where $m = 1280$, $k = 7$, and $c = 133$.

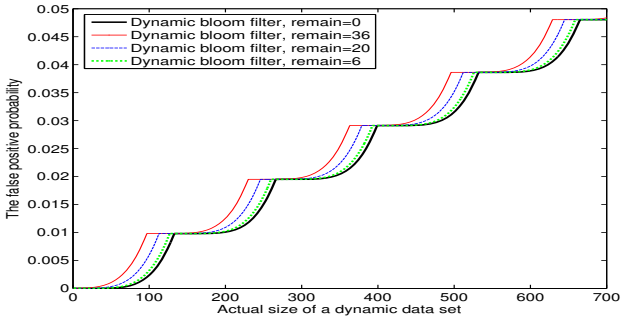


Fig. 6. False positive probability of dynamic Bloom filters are functions of the actual size n of a dynamic set and the remaining data, where $m = 1280$, $k = 7$, and $c = 133$.

TABLE 1
Experimental upper bound and real value of r with $c = 133$

n/c	2	3	4	5	6	7	8	9	10
Upperbound	4	7	11	22	37	53	74	98	123
Real value	3	4	4	6	9	13	20	30	36

high-quality pseudo-random numbers. This mechanism requires one hash function and one random number generator to run $k-1$ rounds of (11) in order to generate a Bloom filter address $Bfadddress(x)$ for item x. It provides a considerable amount of processing reduction compared to using k actual hashes. Kirsch *et al.* shows that this method does not increase the probability of false positives [29].

The multiple address problem causes some items to remain in a DBF even after they have been deleted from set X. The ratio of the number of such items to the cardinality of set X is denoted as r. Recall that $f_2(n)$ is an estimated upper bound on r based on mathematical analysis. The experimental upper bound on r and real value of r are the average values achieved from 100 rounds of simulations with different sets in each round.

Note that dynamic Bloom filters are designed to represent many possible sets, and there are no benchmark sets in the field of Bloom filters. Our experiments do not seek particular sets but simply use names of files at some peers in a peer-to-peer trace as the data source. The P2P trace is a snapshot of files that were shared by eDonkey peers between Dec 9, 2003 and Feb 2, 2004, and record 11014603 distinct files. We initialize ten sets with cardinality $i \times c$ for $1 \leq i \leq 10$ using the trace data, and then implement ten DBFs. In our experiment, the parameters of each SBF are $m = 1280$, $k = 7$, and $c = 133$. For each DBF, the number of items possessing multiple Bloom filter addresses from the corresponding set is determined, and then the experimental upper bound of r is calculated. For each DBF, the item deletion algorithm mentioned above is performed, and the ratio of the number of remaining items to the original cardinality of the corresponding set is the real value of r. Table 1 shows the experimental results under different dynamic sets.

Figure 5 shows that the real value of r is less than the experimental upper bound, the estimated upper bound and the false positive probability of a DBF. All four curves increase as the size of the dynamic set increases. The curve for the real value r increases smoothly and maintains a low level when the real size of the set is less than ten times that of the estimated threshold. On the other hand, the frequency of deleting all items from a set and its corresponding DBF is often low, and the period is usually long.

Let's consider the dynamic sets with $n/c = 10, 8, 5$. We first delete all items of those sets based on the item deletion algorithm, and then continuously add new items to the set. It is easy to understand that the remaining items can increase the false match probability of the DBF. Figure 6 shows that the more remaining items there are, the greater the false positive probability will be. In practice, the frequency of deleting all items from a set and its corresponding DBF is very low; the false positive probability of a DBF is often larger than the expected value, but still maintains a lower, more stable level. On the other hand, the real capacity of a DBF decreases with the number of those remaining items. Applications can use the change in the real capacity as a metric to evaluate the influence of the item deletion operation and to decide whether to represent the updated set again. In the future, we will seek a better method to solve the multiple address problem for an item.

3.5 Optimizations of Dynamic Bloom Filters

In this subsection, we consider cases in which applications do not require DBFs to provide the item deletion operation. In those cases, DBFs use SBFs as a component and may be optimized from the following aspects. None of the optimization strategies proposed below are suitable to other cases in which DBFs use CBFs as a component.

3.5.1 Improvement of Item Insertion Operation

The item insertion algorithm proposed previously could be optimized in two scenarios. First, sometimes it seems to a new item of a normal set that it has been represented by a DBF, even if it is not. In this case, the new item is still

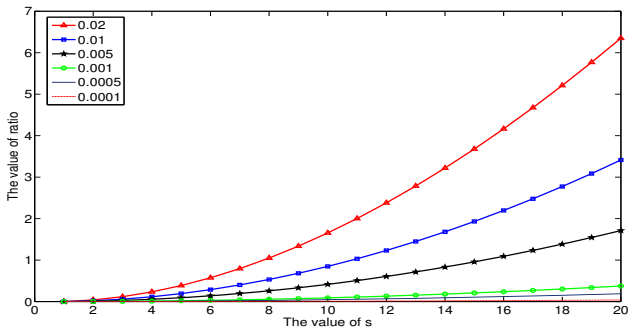


Fig. 7. The ratio of the number of items which have at least one Bloom filter address before they are represented to the threshold c is a function of s and the threshold f.

inserted into an active SBF of the DBF. This kind of item may cause the DBF to allocate unnecessary SBFs. Second, duplicate insertions of an identical item in a multi-set do not necessarily harm an SBF, but they may cause a DBF to extend unnecessarily [30]. Solving this issue requires a membership query before each insert; however, doing so increases the insertion complexity from k to $O(k \times s)$. Considering the additional computational costs, DBFs should only adopt the improved item insertion algorithm if the previous Algorithm 1 causes at least one unnecessary SBF.

As discussed in Section 3.4, $n \times f_2(n)$ denotes an estimation of the number of items which have multiple Bloom filter address after all items are represented, and are very similar to the experimental results. The experimental results, as well as the estimation, are larger than the number of items which already have at least one Bloom filter address before they are represented in the two scenarios. Thus, the improved algorithm should be used only if

$$ratio = \frac{n \times f_2(n)}{c} \geq 1,$$

which implies that Algorithm 1 incurs at least one unnecessary SBF. The $ratio$ is a monotonically increasing function of s and f which denotes an upper bound on the false match probability of an SBF. As shown in Figure 7, the value of $ratio$ is always less than one if $f \leq 0.01, s \leq 10$ or $f \leq 0.001, s \leq 20$; hence, it is not necessary to use the improved algorithm. Under other conditions, Algorithm 1 should be replaced by the improved algorithm since the former causes at least one unnecessary SBF.

The above discussions fail to consider the impact of a multi-set. If the distribution of duplicate items in a multi-set is known in advance, a multi-set could be treated as a normal set where those duplicate items act as identical items in a normal set, and could be analyzed in the same way. Otherwise, we recommend adopting the improved item insertion algorithm, which could be optimized by a better method of storing dynamic Bloom filters, as discussed below.

3.5.2 Compressed Dynamic Bloom Filters

In some distributed applications, an SBF at a node is usually delivered to one or more nodes as a message. In this case, besides the three metrics of SBFs we have seen so far, (1) the computational overhead of an item query operation, (2) the size of the filter in memory, and (3) the false match rate, a fourth metric can be used: the size of a message used to transmit an SBF across the network. The compressed Bloom filters might significantly save bandwidth at the cost of larger uncompressed filters and some additional computation to compress and decompress the filter sent across the network. In the idealized setting, using compression always reduces the false positive probability by adopting a larger Bloom filter size and fewer hash functions than an SBF uses. Interested readers may obtain details concerning all of the theoretical and practical issues of compressed Bloom filters in [18].

It is reasonable to compress a DBF by using compressed Bloom filters instead of SBFs. The compressed DBFs and MDDBFs could reduce both the transmission size and false positive probability of the uncompressed versions at the cost of larger memory and additional computation overheads.

3.5.3 Approach for Storing Dynamic Bloom Filters

There are two ways of storing a dynamic Bloom filter with a set number of s SBFs. These are referred to as the bit string and bit slice methods, respectively [31]. The bit string approach stores the s SBFs dependently and sequentially. Instead of storing a DBF as an $s \times m$ long bit strings, the bit slice method stores a DBF as an $m \times s$ long bit slices. We know that only a subset of the bit positions in each SBF need to be examined on a query. One problem with the bit string approach is that all $s \times m$ bits need to be retrieved on a query. For the bit slice approach, only a fraction of the bit slices need to be retrieved on a query.

4 Performance Evaluations

We use α to denote an upper bound on the false match probability of an SBF representing a static set with fixed cardinality n. Given α and n, the parameters k and m could be optimized for the SBF with $m = \lceil n \times \log(\alpha)/\log(0.61285) \rceil$ and $k = \lceil (m/n) \ln 2 \rceil$. Given α and m, the capacity c can be optimized for the SBF with $c = \lceil m \times \log(0.61285)/\log(\alpha) \rceil$. It is clear that the amount of memory allocated to an SBF increases linearly with n. SBF and its variations are practical approaches to represent static sets; however, most applications often encounter dynamic sets without fixed cardinality besides static sets. According to the structure of DBFs, we know that DBFs can represent dynamic sets as well as static sets. In this section, we will first evaluate the performances of DBFs and SBFs in stand-alone applications with three different sets, and then discuss the distributed applications of DBFs.

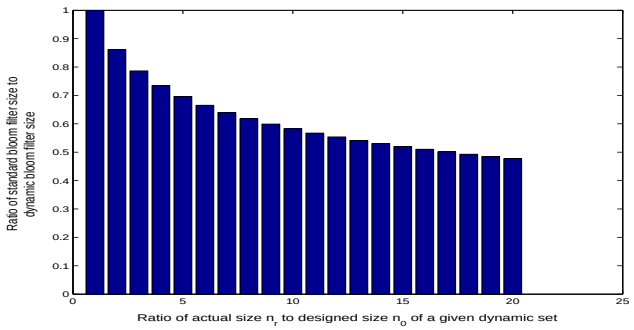


Fig. 8. The ratio of size of a Bloom filter to that of a DBF is a function of a non-negative integer, which denotes the ratio of n to c. The experimental condition is the same as that in Figure 3.

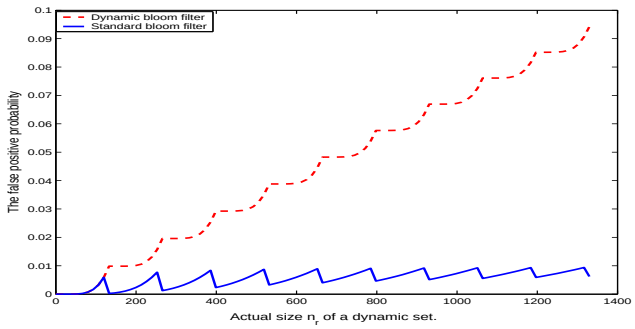


Fig. 9. False positive probability of dynamic and standard Bloom filters are functions of the actual size of a dynamic set. Standard Bloom filters can expand the filter size m to $\lceil n/c \rceil \times m$. $m = 1280$, $k = 7$, and $c = 133$.

4.1 Static Set with Fixed Cardinality

A dynamic set could be regarded as a series of static sets over a sequence of discrete time. In this section, we use SBF and DBF to represent a dynamic set, and compare them from two aspects at any given discrete time. The SBF is reconstructed according to its optimal configuration, as determined by the set cardinality.

For a dynamic set X, let s be $\lceil n/c \rceil$ where n and c denote the cardinality of X and capacity of SBF used by DBF, respectively. For a DBF representing the set, formula of its false positive probability is simplified as formula (12). For an SBF representing that set, it calculates how many bits it must consume in order to achieve the same false positive probability as the DBF. Finally, we establish the relationship between formulas (1) and (12), and achieve formula (13) to denote the ratio of the number of bits m_1 used by an SBF to the number of bits $s \times m$ used by a DBF.

$$f^{DBF}_{m,k,c,n} = 1 - (1 - (1 - e^{-k \times c/m})^k)^{\lceil n/c \rceil} \quad (12)$$

$$\frac{m_1}{s \times m} = \frac{-k \times c}{m \times ln(1 - \sqrt[k]{1 - (1-y)^s})} \quad (13)$$

The following conclusions can be drawn from formula (13) and Figure 8. To obtain the same false match probability, SBF and DBF use the same bits to represent X if $n \leq c$. However, the SBF consumes fewer bits than the DBF if $n > c$. The difference of bits used by DBF and SBF is small if s is not too large.

Let us compare the false positive probability of an SBF and a DBF which use the same bits to represent an identical dynamic set. That is, an SBF is allowed to expand its size to $s \times m$ and to re-represent the dynamic set as the set cardinality grows. In this case, the standard Bloom filter is defined as NBF. The false match probability of an SBF could be calculated according to (1). The false match probability of a DBF should still be (3). It is necessary to compare $f^{DBF}_{m,k,c,n}$ and $f^{NBF}_{m,k,c,n}$ under this situation, and we have

$$f^{NBF}_{m,k,c,n} = (1 - e^{-k \times n/(m \times \lceil n/c \rceil)})^k \quad (14)$$

The experimental results are shown in Figure 9. We conclude that $f^{DBF}_{m,k,c,n} = f^{NBF}_{m,k,c,n} \leq f^{BF}_{m,k,c}$ for $n \leq c$. For $n > c$, $f^{DBF}_{m,k,c,n}$ grows as the set cardinality increases, and $f^{NBF}_{m,k,c,n}$ fluctuates between $i \times c$ and $(i+1)c$, where i is any non-negative integer. Let $n_x < c$ be any non-negative integer, thus $f^{NBF}_{m,k,c,n_x+(i-1)\times c}$ is not larger than $f^{NBF}_{m,k,c,n_x+i\times c}$. In fact, $f^{NBF}_{m,k,c,n}$ grows as the set cardinality increases in the whole range, but the increase rate is slower than that of $f^{DBF}_{m,k,c,n}$.

In summary, to achieve the same false match probability, an SBF never uses more bits than a DBF to represent any static version of that dynamic set if the SBF is actively reconstructed as the increase of set cardinality. It is clear that the SBF produces large, even huge, overhead due to frequent reconstructions. On the contrary, a DBF is not required to be reconstructed if its false match probability is controlled at an acceptable level with the increase of set cardinality. Note that the false match probability of a DBF might sometimes become too large to be tolerated by many applications. It is necessary to occasionally reconstruct the DBF. In the next subsection, we will compare DBFs and SBFs in the whole lifetime of a dynamic set instead of comparing them in a series of discrete time.

4.2 Dynamic Set with an Upper Bound on Set Cardinality

In this section, we use SBF as well as DBF to represent a dynamic set X with an upper bound N on set cardinality. Let α denote the upper bound on false match probability of the SBF as well as DBF. In many applications, the distribution of set cardinality covers a large range [32], [33]. In such a distribution, the upper bound is sometimes several orders of magnitude larger than the mean or minimum cardinality. Applications usually allocate a large number of bits for an SBF at the outset with $m = \lceil N \times log(\alpha)/log(0.61285) \rceil$. These bits are large enough for the SBF to accommodate all possible items of X, while decreasing the space-efficiency of the SBF. DBF, however, allocates enough bits in an incremental and on-demand fashion. A common objective of SBF and DBF is to guarantee that the false match probability never

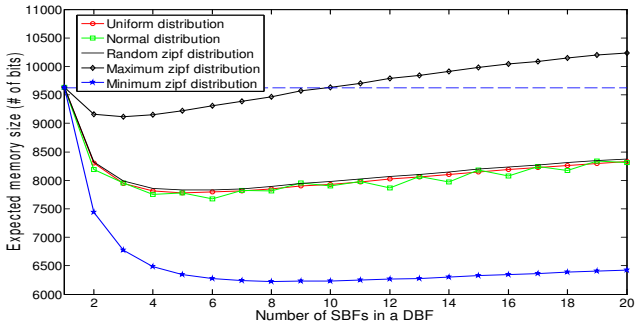


Fig. 10. The memory size of a DBF under different set cardinality distributions: a uniform distribution, a normal distribution with $u = \lceil N/2 \rceil$ and $\sigma^2 = 20$, and a Zipf distribution with 0.4 as parameter, where N=1330 and α=0.0098.

exceeds the α and to make sure that they are not required to be reconstructed as the set cardinality changes.

We now address the problem of designing a minimum size DBF when the probability density function of the set cardinality is known. We assume that s homogeneous SBFs make up the DBF, and δ denotes the upper bound on the false match probability of each SBF. The overall false match probability α for the DBF has to be apportioned among the individual SBFs. According to formula (3), we know that $\alpha = 1 - (1 - \delta)^s$. As a result, we can derive the value of parameter δ with $\delta = 1 - (1 - \alpha)^{1/s}$.

The capacity of the DBF is N since at most N items of the set are accommodated by it. Since the N items are allocated to a certain number of s SBFs evenly, the capacity of the i^{th} SBF is defined as $c_i = \lceil N/s \rceil$ for $1 \leq i \leq s$. In order to solve this problem, we must determine the parameters m, k, δ and s such that the false match probability DBF never exceeds the upper bound α.

Let p_i represent the probability that X has i items where $1 \leq i \leq N$, i.e., $\sum_{i=1}^{N} p_i = 1$. We associate the i^{th} SBF of the DBF with a r_i which implies an upper bound on the probability that this SBF is used. Let $r_1 = 1$ and $r_i = \sum_{j=c\times(i-1)+1}^{N} p_j$ for $i = 2, 3, ..., s$, where c denotes the capacity of any SBF. The expected number of bits used by the DBF is upper bounded by $\sum_{i=1}^{s} m \times r_i$. Recall that the bits used by each SBF is $m = \lceil (N/s) \times \log(\delta)/\log(0.6185)) \rceil$. We formulate the problem to minimize $\sum_{i=1}^{s} r_i \times (N/s) \times \log(\delta)/\log(0.6185)$.

$$\begin{aligned} \text{Minimize} \quad & \sum_{i=1}^{s} r_i \times (N/s) \times \log(\delta)/\log(0.6185) \\ \text{Subject to} \quad & \\ \delta \quad = \quad & 1 - (1 - \alpha)^{1/s} \\ & s > 0 \end{aligned}$$

The optimized value of parameter s could be derived from the solution of this optimization problem. Once we have s, it can be used to calculate the false match probability δ and capacity c for these homogeneous SBFs, and then determine the parameters m and k according to the design method of an SBF. The set cardinality distribution has a direct impact on the result of this optimization problem. We compare the minimized memory size of a DBF under five different cardinality distributions through experiments: normal distribution, uniform distribution, random Zipf distribution, minimum Zipf distribution, and maximum Zipf distribution.

The first two distributions have been widely used for generating synthetic sets to emulate real sets [34]. In many networking applications, it is observed that the set cardinality at each node conforms to a Zipf distribution with a long tail. There is a bijective mapping from the cardinality values to the rank values. The Zipf distribution is called *random* if any rank value is mapped to a random integer over a range $\{1, ..., N\}$ uniformly. It is called *minimum* if the largest and second largest cardinality values are mapped to the last and second last ranks respectively, and so on. It is called *maximum* if the largest and second largest cardinality values are mapped to the first and second ranks respectively, and so on.

As shown in Figure 10, all five curves follow a similar trend as s increases under the constraints that N=1330 and α=0.0098. The expected memory size of each curve first decreases as s grows, and then increases after the s exceeds one or more keen points on the whole. It is clear that the expected memory size under the maximum Zipf distribution is always larger than that under other distributions of the same value of s. The reason is that the r_i under the maximum Zipf distribution is greater than that under other distributions for $2 \leq i \leq s$ and any given value of s. For each curve of DBF, the minimum memory size is achieved when the value of s is equal to a keen point on the curve, and is less than that of an SBF under the same constraints.

We then evaluate the impact of set cardinality on the minimum memory size of the SBF and the five different DBFs where α=0.0098. Figure 11 shows that a DBF with random Zipf distribution uses almost the same amount of memory as a DBF with uniform distribution, while DBFs with maximum and minimum Zipf distributions consume the most and least memory among the five DBFs, respectively. All DBFs, however, consistently outperform SBFs independent of set cardinality, and the performance difference seems to widen as set cardinality increases. In experiments, we also focus on the influence of set cardinality on the ratio of memory size of DBF to that of SBF. As shown in Figure 12, DBFs with maximum Zipf distribution, random Zipf distribution, normal distribution, and minimum Zipf distribution can save about 5%, 19%, 20%, and 35% of the memory used by SBF, respectively. The experimental results show that the set cardinality has a trivial impact on the ratio of the memory size of DBF to that of SBF, while the cardinality distributions have a major impact on that metric.

Moreover, we evaluate the impact of false match probability on the minimum memory size of SBF and the five different DBFs where N=13300. Figure 11 shows that DBFs with maximum and minimum Zipf distributions consume the most and least memory among the five DBFs, respectively. The five DBFs, however, consistently

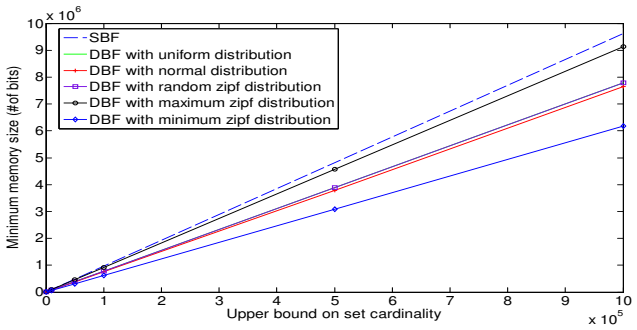


Fig. 11. Memory size under different values of set cardinality.

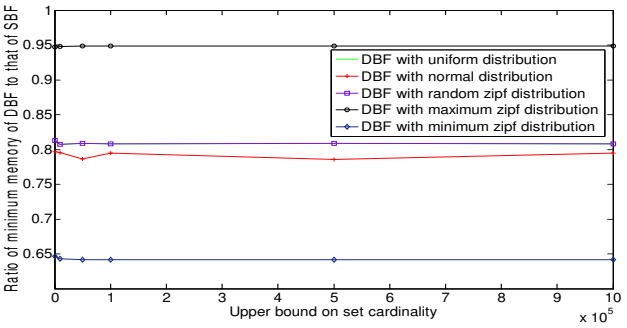


Fig. 12. Ratio of memory size of DBF to that of SBF under different set cardinality distributions.

outperform SBFs independent of false match probability. In experiments, we also focus on the influence of the false match probability on the memory size of DBF, and the ratio of memory size of DBF to that of SBF. As shown in Figure 13, for each DBF, the memory size decreases as the false match probability increases. As shown in Figure 14, for each DBF, the ratio increases as the false match probability increases; however, it will always be less than 1 when the false match probability is not larger than 5%. Actually, the ratio for each DBF will reach, at most, 1 when the false match probability exceeds 5%. That is, the five DBFs never consume more memory than SBF, and save more memory as the false match probability decreases.

Figure 15 shows the impact of the Zipf parameter on memory size under different Zipf distributions[2] where N=1330 and α=0.0098. We observe that DBFs with a minimum Zipf distribution perform better as the parameter value increases, and DBFs with a random Zipf distribution outperform SBFs almost independent of the Zipf parameter value. If the Zipf parameter is less than 0.6, DBFs with a maximum Zipf distribution also outperform SBFs, otherwise, they perform worse than SBFs. Figure 16 shows the impact of standard deviation σ on the memory size of DBFs with normal distribution. We observe that DBFs with normal distribution use more memory as the value of σ increases, however, they

2. A large Zipf parameter means that the frequencies of some cardinality values are much higher than others. A small Zipf parameter means that the frequency of each cardinality value occurs just as often

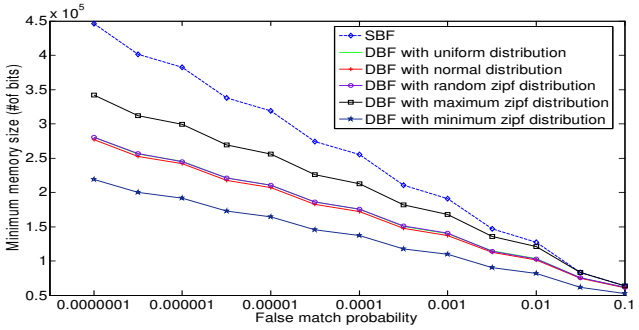


Fig. 13. Memory size under different false match probabilities.

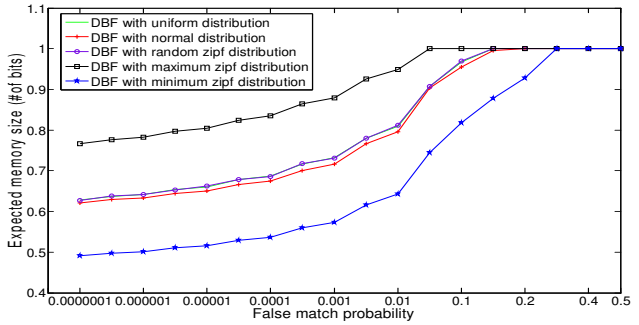


Fig. 14. Ratio of memory size of DBF to that of SBF under different false match probabilities.

always outperform SBFs.

4.3 Dynamic Set without an Upper Bound on Set Cardinality

In this section, we consider another scenario in which applications do not know the upper bound N in advance. Let β and γ denote the left and right upper bounds on the false match probability with $\beta < \gamma$. In this scenario, applications use a pair of β and γ instead of a tight upper bound α. In reality, applications expect the false match probability is less than β, and also tolerate an event that the false match probability is sometimes greater than β but less than γ. A common objective of DBF and SBF is to guarantee that the false match probability never exceeds γ as the set cardinality changes. In order to satisfy this objective, the DBF and the SBF may be reconstructed as the cardinality changes.

For a dynamic set, applications usually estimate a threshold n_0 of the set cardinality and use an initial SBF to represent the dynamic set. In practice, it is very difficult to estimate n_0 in an accurate manner. One possible approach is to trace the change of a dynamic data set, and then to investigate the statistic metric of set size before using DBF to represent that data set. The parameters m and k for an SBF are initialized with m=$\lceil n_0 \times \log(\beta)/\log(0.61285) \rceil$ and k=$\lceil (m/n_0)\ln 2 \rceil$. As shown in Figure 1, the false match probability exceeds β dramatically when the set cardinality exceeds n_0 gradually. Moreover, the false match probability exceeds γ once the set cardinality is greater than $n_0 \times \log(\beta - \gamma)$. To

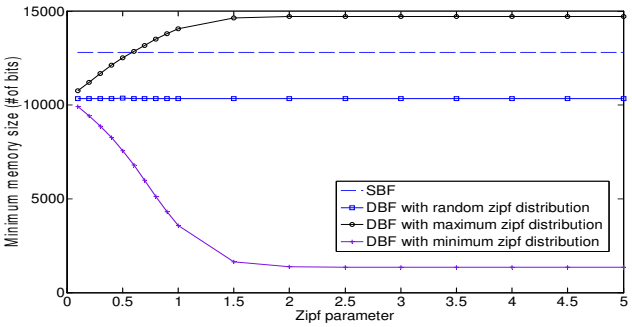


Fig. 15. Impact of Zipf parameter on minimum memory size

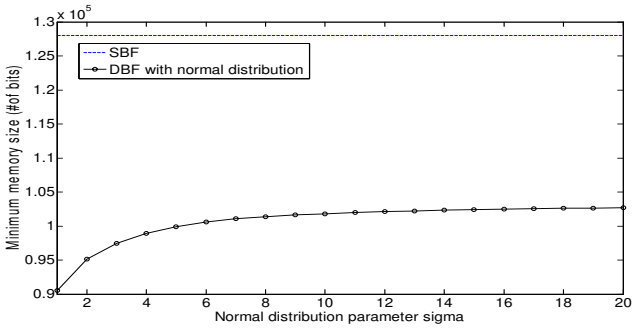


Fig. 16. Impact of standard deviation σ on minimum memory size

handle this issue, n_0 is assigned a new value which is at least greater than $n_0 \times \log(\beta - \gamma)$, and then the initial SBF is reconstructed using new parameters to represent the dynamic set again. There exist many policies to enlarge the value of n_0. This paper does not discuss this in detail since different policies have less impact on the final results. If the false match probability exceeds γ again, the SBF is adjusted in the same way.

DBF first adopts an SBF to represent the dynamic set, and may expand its capacity by allocating more SBFs as the set cardinality increases. As shown in Figure 1, the false match probability of DBF increases slower than SBF when the set cardinality gradually exceeds c. The false match probability exceeds γ once the set cardinality is greater than $n_0 \times \lceil \log(1-\gamma)/\log(1-\beta) \rceil$. To deal with this issue, c is reassigned a new value which is at least greater than $n_0 \times \lceil \log(1-\gamma)/\log(1-\beta) \rceil$, and the DBF is reconstructed to represent the dynamic set again. This paper does not discuss policies to enlarge n_0 in detail for similar reasons. If the false match probability of a new DBF exceeds γ again, the DBF must be adjusted in the same way.

In reality, it is unavoidable to reconstruct DBFs and SBFs under specific conditions if the upper bound on set cardinality is not known a priori. Fortunately, the adjustment frequency of DBF is lower than that of SBF, especially when the difference between β and γ is large; that is, DBF causes less overhead and is more stable than SBF due to infrequent reconstructions as the increase of set cardinality. Note that if the difference between β and γ is very low, the benefit of our approach using left and right bounds on the false match probability becomes trivial. To address this rare case, we use the approaches proposed in Section 4.2 after estimating the cardinality distribution and initializing N=n_0 and α=γ.

4.4 Distributed Application Scenarios

In the above discussions, we considered SBF and DBF as objects residing in memory in stand-alone applications. In distributed applications, however, they are not just objects that reside in memory, but objects that must be transferred between nodes. In this case, all nodes are required to adopt the same configuration of m, k, and hash functions in order to guarantee compatibility and interoperability of SBF or DBF between any pair of nodes.

We first consider the case in which the upper bounds N and α on the set size and false match probability over nodes, depending on the applications, are known a priori. Nodes can construct a local but homogeneous SBF with m=$\lceil N \times \log(\alpha)/\log(0.61285) \rceil$ and k=$\lceil (m/N)\ln 2 \rceil$ even if those sets are different in size. This approach requires the nodes with small sets to sacrifice more space to be in accordance with those nodes with the large sets, hence hurting the space-efficiency and causing large transmission overhead. As discussed in Section 4.2, DBF can address this drawback of SBF if the distribution of set sizes over nodes is known by the relevant application. The reason is that each node allocates just enough memory to a DBF according to its set size, and can satisfy the requirement of compatibility and inter-operability of DBF with other nodes. Although the approaches proposed in Section 4.2 focus on stand-alone applications, they are also suitable to distributed applications. The only difference is that the expected number of bits used by a DBF is minimized in stand-alone applications, while the total number of bits used by DBFs at all nodes is minimized in distributed applications. For more information, we refer the reader to Section 4.2.

We then consider the case in which the upper bound on set sizes over nodes is not known in advance. In this scenario, as discussed in Section 4.3, applications impose β and γ ($\beta < \gamma$) as a pair of upper bounds on the false match probability over nodes, and estimate a threshold n_0 on the upper bound on set sizes over nodes. If applications use SBFs to represent sets over nodes, an event where the size of the set at any node exceeds $n_0 \times \log(\beta - \gamma)$, will trigger a reconfiguration of its SBF, thereby propagating a new configuration to other nodes, and reconstructing an SBF at each node. It is clear that frequent reconfigurations lead to huge overhead and destroy the stability of applications. One possible solution to this problem is to overestimate n_0 and allocate a larger SBF at each node. This solution, however, hurts the space-efficiency of SBF and causes large transmission overhead. If applications use DBFs instead of SBFs, all nodes reconfigure their DBFs only if the set size at any node exceeds $n_0 \times \lceil \log(1-\gamma)/\log(1-$

$\beta)\rceil$. Note that the node just expands its DBF without performing the consistency operation over all nodes if its set size is greater than $n_0 \times \log(\beta - \gamma)$ but less than $n_0 \times \lceil \log(1-\gamma)/\log(1-\beta)\rceil$. It is clear that the adjustment frequency of DBF is lower than that of SBF especially when the difference between β and γ is large. DBFs are more stable than SBFs with the increase of set cardinality in this case. For more information, we refer the reader to Section 4.3.

After discussing the use of approaches of DBF, we consider a necessary procedure to update a DBF in distributed applications. For each DBF, we adopt an incremental update procedure by only sending those SBFs which are changed. For each varied SBF, the procedure to send updates first inspects if an old version of the SBF exists in the previous DBF. If not, this must be an added SBF and the update is simply the SBF itself. Otherwise, an update is sent by computing the xor of the current version with the previous version. All updates can be compressed using arithmetic coding before being sent reliably. At the other end, the procedure to receive each updated SBF first inspects if a previous SBF exists. If not, this must be an added new SBF and the update is simply stored in the corresponding DBF. Otherwise, the updated SBF is treated as an incremental one and its previous SBF is modified suitably by computing its bitwise xor with the new update.

5 Conclusion

A Bloom filter is an excellent data structure for succinctly representing static sets with fixed cardinality in order to support membership queries. However, it does not take dynamic sets into account. In reality, most applications often encounter dynamic data sets as well as static sets. We present dynamic Bloom filters to deal with dynamic sets as well as static sets. Dynamic Bloom filters not only inherit the advantage of Bloom filters, but also have better features than Bloom filters when dealing with dynamic sets. The false match probability of Bloom filters increases exponentially with the increase of the cardinality of a dynamic set, while that of dynamic Bloom filters increases slowly because it expands capacity in an incremental manner according to the set cardinality.

Through comprehensive mathematical analysis, we show that dynamic Bloom filters use less expected memory than Bloom filters when dealing with dynamic sets with upper bounds on set cardinality, and that dynamic Bloom filters are more stable than Bloom filters due to infrequent reconstruction when addressing dynamic sets without upper bounds on set cardinality. Moreover, the analytical results hold in stand-alone applications as well as distributed applications. The only disadvantage is that dynamic Bloom filters do not outperform Bloom filters in terms of false match probability when dealing with a static set with the same size memory.

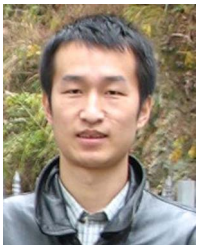
Acknowledgments

The authors would like to thank Maria Basta and Christian Esteve for their constructive comments and careful proofreading. The work of Deke Guo was supported in part by the Ph.D. Candidates Research Foundation of National University of Defense Technology under Grant No. 0615. The work of Jie Wu was supported in part by NSF grants CNS 0422762, CNS 0434533, CNS 0531410, and CNS 0626240.

References

[1] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
[2] J. K. Mullin. Optimal semijoins for distributed database systems. *IEEE Trans. Software Eng.*, 16(5):558–560, 1990.
[3] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proc. the 12th International Conference on Very Large Data Bases (VLDB)*, pages 149–159, Kyoto, Jpn, August 1986.
[4] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2005.
[5] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, and D. Geels. Oceanstore: An architecture for global-scale persistent storage. *ACM SIGPLAN Notices*.
[6] J. Li, J. Taylor, L. Serban, and M.Seltzer. Self-organization in peer-to-peer system. In *Proc. the 10th ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.
[7] F. M. Cuena-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. Plantp: Using gossiping to build content addressable peer-to-peer information sharing communities. In *Proc. the 12th IEEE International Symposium on High Performance Distributed Computing*, pages 236–249, Seattle, WA, USA, June 2003.
[8] S. C. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *Proc. IEEE INFOCOM*, pages 1248–1257, New York, NY, United States, June 2004.
[9] T. D. Hodes, S. E. Czerwinski, and B. Y. Zhao. An architecture for secure wide-area service discovery. *Wireless Networks*, 8(2-3):213–230, 2002.
[10] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Proc. ACM International Middleware Conference*, pages 21–40, Rio de Janeiro, Brazil, June 2003.
[11] D. Bauer, P. Hurley, R. Pletka, and M. Waldvogel. Bringing efficient advanced queries to distributed hash tables. In *Proc. IEEE Conference on Local Computer Networks*, pages 6–14, Tampa, FL, United States, November 2004.
[12] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Networking*, 8(3):281–293, 2000.
[13] C. D. Peter and M. Panagiotis. Bloom filters in probabilistic verification. In *Proc. the 5th International Conference on Formal Methods in Computer-Aided Design*, pages 367–381, Austin, Texas, USA, November 2004.
[14] C. Jin, W. Qian, and A. Zhou. Analysis and management of streaming data: A survey. *Journal of Software*, 15(8):1172–1181, 2004.
[15] F. Deng and D. Rafiei. Approximately detecting duplicates for streaming data using stable bloom filters. In *Proc. 25th ACM International Conference on Management of Data (SIGMOD)*, pages 25–36, Chicago, Illinois, USA, June 2006.
[16] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. Beyond bloom filters: From approximate membership checks to approximate state machines. In *Proc. ACM SIGCOMM*, pages 315–326, Pisa, Italy, September 2006.
[17] K. Li and Z. Zhong. Fast statistical spam filter by approximate classifications. In *Proc. the Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/Performance*, pages 347–358, Saint Malo, France, June 2006.
[18] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Trans. Networking*, 10(5):604–612, 2002.
[19] A. Kirsch and M. Mitzenmacher. Distance-sensitive bloom filters. In *Proc. the 8th Workshop on Algorithm Engineering and Experiments (ALENEX06)*, Miami, Florida,USA, January 2006.

[20] A. Kirsch and M. Mitzenmacher. Building a better bloom filter. tr-02-05.pdf, January 2006.
[21] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li. Space-code bloom filter for efficient per-flow traffic measurement. In *Proc. 23rd IEEE INFOCOM*, pages 1762–1773, Hongkong, China, March 2004.
[22] S. Cohen and Y. Matias. Spectral bloom filters. In *Proc. 22nd ACM International Conference on Management of Data (SIGMOD)*, pages 241–252, San Diego, CA, United States, June 2003.
[23] R. P. Laufer, P. B. Velloso, and O. C. M. B. Duarte. Generalized bloom filters. Technical Report Research Report GTA-05-43, University of California, Los Angeles (UCLA), September 2005.
[24] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 30–39, New Orleans, Louisiana, USA, January 2004.
[25] F. Hao, M. Kodialam, and T. V. Lakshman. Building high accuracy bloom filters using partitioned hashing. In *Proc. SIGMETRICS/Performance*, pages 277–287, San Diego, CA, USA, June 2007.
[26] D. Guo, J. Wu, H. Chen, and X. Luo. Theory and network applications of dynamic bloom filters. In *Proc. 25th IEEE INFOCOM*, Barcelona, Spain, April 2006.
[27] M. Xiao, Y. Dai, and X. Li. Split bloom filters. *Chinese Journal of Electronic*, 32(2):241–245, 2004.
[28] P. S. Almeida, C. Baquero, N. M. Preguiça, and D. Hutchison. Scalable bloom filters. *Inf. Process. Lett.*, 101(6):255–261, 2007.
[29] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better bloom filter. In *Proc. the 14th Annual European Symposium on Algorithms (ESA 2006)*, pages 456–467, Zurich, Switzerland, September 2006.
[30] J. Wang, M. Xiao, J. Jiang, and B. Min. I-dbf: an improved bloom filter representation method on dynamic set. In *Fifth International Conference on Grid and Cooperative Computing Workshops*, pages 156–162, Changsha, China, September 2006.
[31] A. Kent and R. S. Davis. A signature file scheme based on multiple organizations for indexing very large text databases. *Journal of the American Society for Information Science*, 41(7):508–534, 1990.
[32] M. Faloutsos C. Faloutsos and P. Faloutsos. On power-law relationships of the internet topology. In *Proc. ACM SIGCOMM*, pages 251 – 262, Cambridge, Massachusetts, United States, August 1999.
[33] F. Hao, M. Kodialam, and T. V. Lakshman. Incremental bloom filters. In *Proc. IEEE INFOCOM*, Phoenix, Arizona, United States, 2008.
[34] S. Melnik and H. C. Molina. Adaptive algorithms for set containment joins. *ACM Transactions on Database Systems*, 28:56–99, 2003.



Deke Guo received the B.E. degree in industry engineering from Beijing University of Aeronautic and Astronautic, Beijing, China, in 2001, and the Ph.D. degree in management science and engineering from National University of Defense Technology, Changsha, China, in 2008. He was a visiting scholar in the Department of Computer Science and Engineering, Hong Kong University of Science and Technology from Jan. 2007 to Jan. 2009. Currently, he is an assistant professor of Information System and Management, National University of Defense Technology, Changsha, China. His current research focus on peer-to-peer networks, Bloom filters, MIMO, data center networking, wireless sensor networks, and wireless mesh networks. He is a member of the ACM and the IEEE.

Jie Wu is currently a distinguished professor in the Department of Computer Science and Engineering, Florida Atlantic University. He has been on the editorial board of the IEEE Transactions on Parallel and Distributed Systems and was a guest editor of Computer and the Journal of Parallel and Distributed Computing. He is currently on the editorial board of the IEEE Transactions on Mobile Computing. He was a program cochair of the First IEEE International Conference on Mobile Ad Hoc and Sensory Systems (MASS 2004), the executive program vice chair of the 28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), and the program vice chair of the 29th International Conference on Parallel Processing (ICPP 2000). He was also the general chair of MASS 2006 and is the general chair of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008). He has served as a distinguished visitor of the IEEE Computer Society and is the chairman of the IEEE Technical Committee on Distributed Processing (TCDP). His research interests include wireless networks and mobile computing, routing protocols, fault-tolerant computing, and interconnection networks. He has published more than 450 papers in various journals and conference proceedings. He is the author of Distributed System Design (CRC Press). He is the recipient of the 1996-1997, 2001-2002, and 2006-2007 Researcher of the Year Awards from Florida Atlantic University. He is a fellow of the IEEE.

Honghui Chen received the M.S. degree in operational research and the Ph.D. degree in management science and engineering from National University of Defense Technology, Changsha, China, in 1994 and 2007, respectively. Currently, he is a professor of Information System and Management, National University of Defense Technology, Changsha, China. His research interests are requirement engineering, web services, information grid, and peer-to-peer networks. He has co-authored three books in the past five years.

Ye Yuan Ye Yuan received the B.S. and M.S. degrees in computer science from Northeastern University, China, in 2004 and 2007, respectively. Currently, he is a Ph.D. candidate in the Department of Computer Science, Northeastern University. He is also a visiting scholar in the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. His current research focus on peer-to-peer computing, graph databases, uncertain database, probabilistic database, and wireless sensor networks.

Xueshan Luo received the B.E. degree in information engineering from Huazhong Institute of Technology, Wuhan, China, in 1985, the M.S. and Ph.D. degrees in system engineering from National University of Defense Technology, Changsha, China, in 1988 and 1992, respectively. He was a faculty member and associate professor at National University of Defense Technology from 1992 to 1994 and from 1995 to 1998, respectively. Currently, he is a professor of Information System and Management, National University of Defense Technology. His research interests are in the general areas of information system and operation research. His current research focuses on architecture of information system.