Using Parallel Bloom Filters for Multiattribute Representation on Network Services

Bin Xiao, *Member*, *IEEE*, and Yu Hua, *Member*, *IEEE*

Abstract—One widely used mechanism for representing membership of a set of items is the simple space-efficient randomized data structure known as Bloom filters. Yet, Bloom filters are not entirely suitable for many new network applications that support network services like the representation and querying of items that have multiple attributes as opposed to a single attribute. In this paper, we present an approach to the accurate and efficient representation and querying of multiattribute items using Bloom filters. The approach proposes three variant structures of Bloom filters: Parallel Bloom Filter (referred as PBF) structure, PBF with a hash table (PBF-HT), and PBF with a Bloom filter (PBF-BF). PBF stores multiple attributes of an item in parallel Bloom filters. The auxiliary HT and BF provide functions to capture the inherent dependency of all attributes of an item. Compared to standard Bloom filters to represent items with multiple attributes, the proposed PBF facilitates much faster query service and both PBF-HT and PBF-BF structures achieve much lower false positive probability with a result to save storage space. Simulation and experimental results demonstrate that the new space-efficient Bloom filter structures can efficiently and accurately represent multiattribute items and quickly respond queries at the cost of a relatively small false positive probability.

Index Terms—Network services, parallel Bloom filters, false positives, data structure.

✦

---

1 INTRODUCTION

STANDARD Bloom filter is an important and widely used tool for supporting efficient query services in networking because of its ability to represent a set of items by using a bit array with several independent hash functions [1]. This allows queries to be made about whether an item is a member of the set and obviates the need to query every individual item in the set. Though this approach causes a small probability of a false positive replying to any membership query, but, to the degree that this is acceptable in particular circumstances, Bloom filters provide an effective tool for saving space when space is at a premium. Recent developments in network services, however, have led to a greater need for the ability to make queries based on the accurate representation of a set of items with multiple, rather than merely single, attributes. Yet, to date, not much work has been done in this area. A chief difficulty is that if, as a result of computing hash functions independently, we store different attributes of a multiattribute item in different places, we will loose information as to the dependency of multiple attributes of an item and this in turn greatly increases the probability of false positives. Currently, standard Bloom filters contain no functional units for recording the dependency of multiple attributes. Clearly, there is a need to develop a new structure for representing items with multiple attributes.

- B. Xiao is with the Department of Computing, Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong, China.
  E-mail: csbxiao@comp.polyu.edu.hk.
- Y. Hua is with the School of Computer, Huazhong University of Science and Technology, Wuhan, China 430074. E-mail: csyhua@mail.hust.edu.cn.

Bloom filters have an essential role in network services and consequently the growing importance of operations such as information retrieval, distributed databases, packet content inspection, and cooperative caching [2], [3], [4] results in the wide applications of Bloom filters that provide set-membership queries based on a relatively easy hardware implementation. However, succinct representation and queries for items that have multidimensional attributes are still becoming a critical challenge, although recent work [5], [6], [7], [8] begins some meaningful attempts based on the fact that many existing data structures are able to provide richer information for indexing. In [9], the authors proposed multidimension dynamic Bloom filters (MDDBFs) for items with multiple attributes. The basic idea was to represent a dynamic set A with a dynamic $s \times m$ bit matrix that consists of s standard Bloom filters with m-bit size. However, the MDDBF approach lacks a way to verify the dependency of multiple attributes of an item, which may increase the probability of false positives. In [10], only preliminary work has been reported, which supported the representation of items with multiple attributes by using parallel Bloom filters and a hash table.

An intuitive approach to representing multiattribute items can concatenate multiple attributes into a single-attribute array to be stored in a Standard Bloom Filter (SBF), which is a way to maintain the inherent dependency of multiple attributes of an item. However, such approach has the overhead for the concatenation operation that may delay query replies to users if multiple attributes have different formats (e.g., string, digit). In fact, it takes a long time to get the hashed result for a single but long attribute array. Nonetheless, standard form in an SBF is essentially a compressed representation, limiting its rich query services. There are no membership query services given a single or partial attributes because an SBF strips off explicit attribute values of an item in the hash function operation. For example, given an existing "Table" with "red" and "large" attributes in an SBF, the intuitive approach cannot provide any meaningful answer to query requests like "*Does the room have a "red" table?*" or "*Is a*

"large" table a member?". In real-world applications, many query requests cannot provide exact and complete descriptions of queried items, which limits the usage of SBFs for queries of partial attributes.

In this paper, we present an approach to the space-efficient representation of multiattribute items. The proposed approach utilizes data structures to carry out fast but accurate membership queries and achieve space savings. We describe data structures in three phases: first, we describe a Parallel-Bloom filter (PBF) structure, then, PBF with a Hash Table (PBF-HT), and finally, PBF with a Bloom Filter (PBF-BF). PBF is a counter-based matrix and consists of multiple submatrices. Each submatrix functions as a Bloom filter and can store one attribute of an item. PBF-HT uses a hash table to store the verification value of an item. This value reflects the inherent dependency of all attributes of an item. We generate this verification value by using an attenuation method to reduce false positives. PBF-BF uses a Bloom filter instead of a hash table to store verification values, achieving greater space savings at the expense of a very small increase to its false positive probability. Note that we only proceed with the verification in HT or BF only when the PBF part returns positive for a query in the PBF-HT and PBF-BF.

This paper has the following technical contributions:

- First, we propose PBF-based structures to represent items with multiple attributes with high query accuracy. To justify the presence of a queried item in PBF-HT and PBF-BF, that is, unrelated attributes are not falsely inferred to a nonexistent item, we utilize a double verification process. The first verification takes place in the PBF part and the second takes place in either HT or BF. Double verifications are necessary because the querying of multiattribute items can be complex and a single verification in PBF alone may not be able to tell multiple attributes from a single item. The verification within the auxiliary HT/BF improves the query accuracy.
- We illustrate data operations of adding, querying, and removing items in proposed novel data structures. We show that they are suitable for fast network query services because these operations maintain computational complexity of $O(1)$.
- We provide mathematical analysis of the query accuracy of proposed data structures by showing their low false positive probabilities. The PBF-based data structures can support algebraic operations to quickly respond queries among multiple (and perhaps distributed) data sets [10]. We show that SBF and PBF structures have the same minimum false positive probability when they use the same amount of space to store n items. PBF-HT and PBF-BF, using very limited storage space to implement HT and BF, can greatly improve query accuracy by minimizing the false positive probability for random queries.

Rather than storing multiattribute items in a standard Bloom filter concatenating multiple attributes into a single-dimensional long array, PBF using parallel Bloom filters can quickly respond to a query by simultaneously carrying out hash operations in each submatrix for a short attribute. There is no need for the concatenation operation of multiple attributes while partial attribute queries are supported by looking at corresponding submatrices. A reply to nonexistent
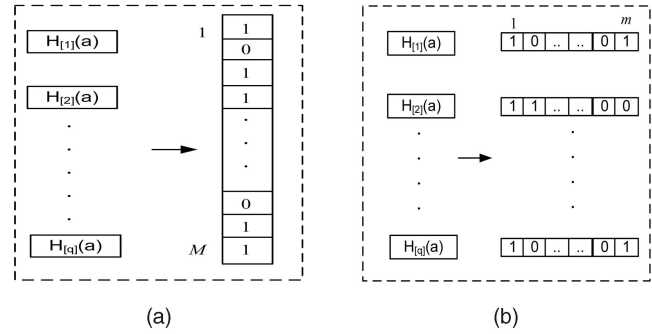

Fig. 1. Standard Bloom filter structures. (a) Flat form. (b) Segment form.

items can be obtained immediately if any submatrix in PBF returns negative. We also provide simulation and experimental results to show the space efficiency and query accuracy of newly proposed data structures. Simulation results illustrate that both PBF-HT and PBF-BF can remarkably reduce the query false positive probability. To maintain the same low false positive probability as PBF-HT and PBF-BF, SBF needs to use much larger amount of storage space. We conducted experiments using the real-world HP trace [11] and results show the space and query efficiency of PBF-HT and PBF-BF over SBF.

The rest of the paper is organized as follows: Section 2 shows basic Bloom filter structures, including standard Bloom filters and the basic PBF. Sections 3 and 4 present the PBF-HT and PBF-BF structures, respectively, and their data operations. In Section 5, we give mathematical analysis on the false positive probability of proposed PBF-based structures. Section 6 provides the performance evaluation. Section 7 describes some related work. Finally, Section 8 concludes our paper.

2 BLOOM FILTER-BASED STRUCTURES

In this section, we first describe the structure of an SBF, which can support operations on single-attribute items. We then describe the parallel Bloom filter (referred as PBF) structure, which is used to represent and query items with multiple attributes.

2.1 Standard Bloom Filter

An SBF, as shown in Fig. 1a, is a bit array of M bits for representing a set $S = \{a_1, a_2, \ldots, a_n\}$ of n items. All bits in the array are initially set to 0. Then, a Bloom filter uses q independent hash functions $\{h_1, \ldots, h_q\}$ to map the set to the bit address space $[1, \ldots, M]$. For each item a, the bits of $h_i(a)$ are set to 1. To check whether an item a is a member of S, we need to check whether all $h_i(a)$ are set to 1. If not, a is not in the set S. If so, a is regarded as a member of S with a false positive probability, which suggests that set S contains an item a although it in fact does not. Generally, the false positive is acceptable if the false positive probability is sufficiently small. Note that a standard Bloom filter often displays either the flat form as shown in Fig. 1a or the segment form as shown in Fig. 1b. Two forms have the same filter size, i.e., M bits totally. The segment form, instead of using one single array of size M that is shared by all q hash functions, divides equally M bits for q hash functions, and thus, each hash function has an array with range $[1, \ldots, m]$
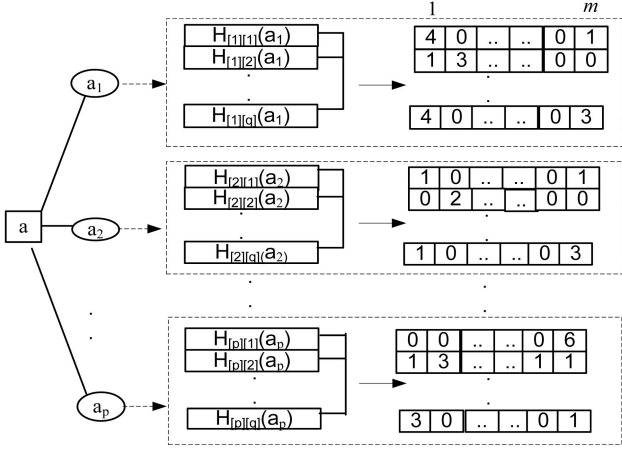
Fig. 2. The basic PBF structure.

and $mq = M$ [3]. The popular segment-based design facilitates the implementation by making easier parallel access to bit arrays of a Bloom filter. Therefore, in our design, we use the segment-based form, which, in essence, is the same as the flat form.

2.2 PBF Structure

An SBF can effectively represent items with a single attribute but it cannot support the representation and querying of items that have multiple attributes. To allow operations on multiattribute items, we propose a simple structure called PBF. To support the deletion operation, PBF utilizes the form of counting Bloom filters. In [12], counting Bloom filters replace bits of standard Bloom filters with counters and the counter size is set to 4 bits to minimize its overflow probability for most real-world applications.

2.2.1 Structural Representation

Fig. 2 shows the architecture of the basic PBF. PBF uses the counting Bloom filters [12] to support the deletion operation and can be viewed as a *matrix*, which consists of p parallel *submatrices* in order to represent p attributes. A submatrix is composed of q parallel *arrays* and can be used to represent one attribute. An array consists of m counters and is related to one hash function. Let $C_{[i][j][k]}$ be the kth ($1 \leq k \leq m$) counter, which is in the jth array of the ith submatrix. q arrays in parallel store results from q hash functions correspondingly. Assume that a_i is the ith attribute of item a. We use $H_{[i][j]}(a_i)(1 \leq i \leq p, 1 \leq j \leq q)$ to represent the hash value computed by the jth hash function for the ith attribute of item a. Thus, each submatrix has

---

Membership_Query_Item (Input: Item a)

1: **for** $(i = 1; i \leq p; i + +)$ **do**
2:     **for** $(j = 1; j \leq q; j + +)$ **do**
3:         $k = H_{[i][j]}(a_i)$
4:         **if** $C_{[i][j][k]} == 0$ **then**
5:             Return *False*
6:         **end if**
7:     **end for**
8: **end for**
9: Return *True*

Fig. 4. Query algorithm in the basic PBF.

$q \times m$ counters and PBF composed of p submatrices utilizes $p \times q \times m$ counters to store n items with p attributes.

2.2.2 Practical Algorithms

When an item with p attributes arrives, we allocate its p attributes separately into p submatrices. We compute the hash value of each attribute based on the given q hash functions and increase corresponding location counters by one. Fig. 3 shows the algorithm for inserting an item with p attributes. To represent a new item, the insertion algorithm must know the computed positions (i.e., k) from q hash functions. It can then increase counters by one in these positions.

Fig. 4 shows the querying algorithm in the basic PBF. q positions computed in q hash functions, as shown in Line 3, are then probed and if all of them are nonzero, the queried item is said to be present; otherwise, it is absent. Since each bucket in an array is a counter, the deletion operation is easy to implement, similar to the item insertion but with counter decrements. The deletion algorithm in the basic PBF is shown in Fig. 5. All operations on PBF can be done in $O(1)$, the same complexity as SBFs.

2.2.3 Limitations of the Basic PBF

The basic PBF uses parallel submatrices to store p attributes of an item, yet such an approach can lead to false positives because the basic PBF cannot distinguish one's attributes from the other's. Take, for example, a situation in which two tables with two attributes, *Table-1 (Large, Red)* and *Table-2 (Small, Green)*, are inserted into the basic PBF. The attributes in the same type should be hashed to the same submatrix as shown in Fig. 6. However, a future query, *"Is the table with attributes (Large, Green) in the Bloom filters?"*, would receive the answer *True* even though the filters do not contain such a table.

Insert_Item (Input: Item a)

1: **for** $(i = 1; i \leq p; i + +)$ **do**
2:     **for** $(j = 1; j \leq q; j + +)$ **do**
3:         $k = H_{[i][j]}(a_i)$
4:         $C_{[i][j][k]} + +$
5:     **end for**
6: **end for**

Fig. 3. Insertion algorithm in the basic PBF.

Delete_Item (Input: Item a)

1: **for** $(i = 1; i \leq p; i + +)$ **do**
2:     **for** $(j = 1; j \leq q; j + +)$ **do**
3:         $k = H_{[i][j]}(a_i)$
4:         $C_{[i][j][k]} - -$
5:     **end for**
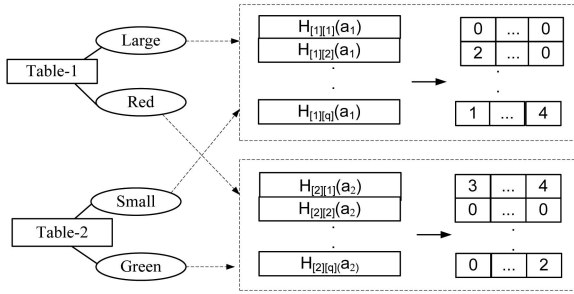6: **end for**

Fig. 5. Deletion algorithm in the basic PBF.

Fig. 6. Two items stored in the basic PBF.

Fig. 7. PBF-HT structure using counting Bloom filters.

This kind of false positives arises because the basic PBF *only* knows the existence of attributes and cannot determine whether the existing attributes belong to one item. The attribute integrity of an item is lost to separately store p attributes into p submatrices, leading to confused query conflicts. Thus, to accurately denote the presence of an item of p attributes, it requires an auxiliary structure that can record the attribute inherent dependency to preserve attribute integrity.

3   A NOVEL STRUCTURE: PBF-HT

In this section, we introduce a novel structure for representing items of p attributes. The proposed structure is composed of parallel Bloom filters and a hash table (HT) that stores the verification values of items. One novel feature of this hash table is that it uses an improved method for generating verification values. We also describe adding, querying, and deleting operations on the proposed PBF-HT Structure.

3.1   Proposed Structure

Fig. 7 shows the proposed structure, which is composed of two parts: PBF and a hash table. PBF and the hash table are used to store multiple attributes and verification values of items, respectively. A verification value in the hash table can verify p-attribute integrity from one item. Let $v_i = F(H_{[i][j]}(a_i))$, where F is a function, be the verification value of the ith attribute of item a. The verification value of item a with p attributes can be computed by $V_a = \sum_{i=1}^{p} v_i$, which will be inserted into the hash table for future dependency tests. In practical implementations, we only keep nonempty hash buckets to reduce required storage space. Any hash function can be used in the hash table. In this paper, we used MD5 [13].

3.2   Hash Table for Dependency Verification

One reason for false positives in multiattribute item queries is that the dependency among multiple attributes is lost after we insert p attributes into p independent submatrices. A hash table is required to confirm whether the queried multiple attributes belong to one item because the PBF structure can only verify the presence of multiple attributes but cannot conclude them from a single item. The presence of a multi-attribute item can be verified when both PBF and the hash table answer *Yes*. Traditionally, hash values computed by hash functions are only used to update location counters in the counting Bloom filters. In the proposed structure PBF-HT, we
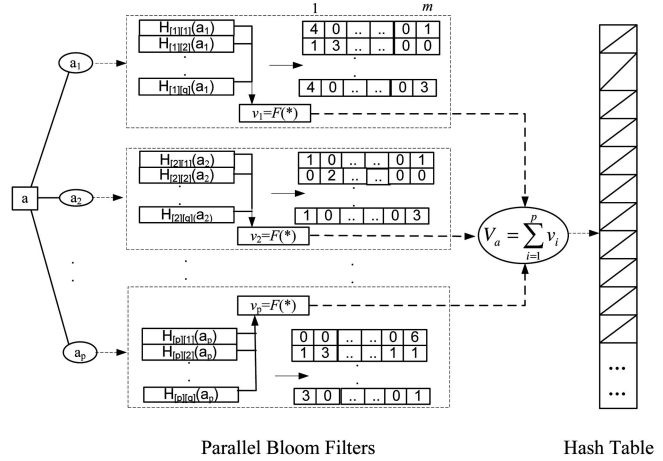
utilize the hash values to generate verification values to be stored in HT, which provides a double verification process, one in PBF and the other in HT, to reduce query false positives.

There are two methods to implement function F to calculate v_i. A basic method of generating the verification value v_i is to add all hash values from q hash functions. For example, the value of v_i can be $v_i = F(H_{[i][j]}(a_i)) = \sum_{j=1}^{q} H_{[i][j]}(a_i)$ for the ith attribute of item a, where function F is the sum operation. The verification value of item a is $V_a = \sum_{i=1}^{p} \sum_{j=1}^{q} H_{[i][j]}(a_i)$, which will be inserted into the hash table. The problem with this method, however, is that the values computed by different hash functions can be the same, as might be their sums. As a result, different items can have a high probability with the same verification value in the hash table that leads to verification collisions and minimizes the verification effect on HT.

An improved attenuated method can distinguish verification values from different attributes in which we assign different weights to sequential hash functions. As for the ith attribute of item a, the value from the jth hash function in the ith submatrix is defined as $\frac{H_{[i][j]}(a_i)}{2^j}$, which is similar to the idea of the *Attenuate Bloom Filters* [14] adopting a lossy distributed index. The verification value of the ith attribute of item a becomes

$$v_i = F\big(H_{[i][j]}(a_i)\big) = \sum_{j=1}^{q} \frac{H_{[i][j]}(a_i)}{2^j}.$$

The verification value of item a in the improved attenuated method is

$$V_a = \sum_{i=1}^{p} \sum_{j=1}^{q} \frac{H_{[i][j]}(a_i)}{2^j}.$$

3.3   Operations on the PBF-HT Structure

Given a certain item a, it has p attributes and each attribute can be represented using q hash functions. We denote its verification value by V_a, which is initialized to zero. We implement the corresponding operations, such as the adding, querying, and deleting items, with a complexity of $O(1)$ in the PBF-HT structure.
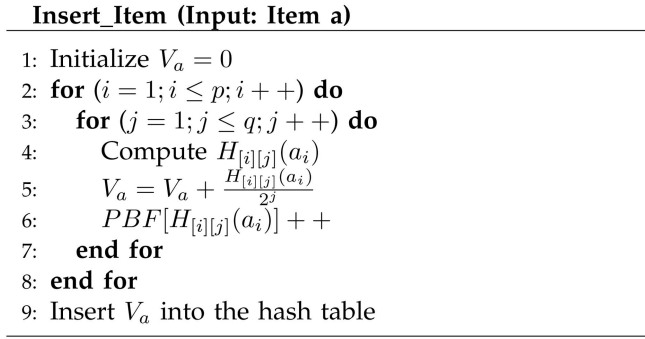

Insert_Item (Input: Item a)

1: Initialize $V_a = 0$
2: **for** $(i = 1; i \leq p; i++)$ **do**
3:     **for** $(j = 1; j \leq q; j++)$ **do**
4:         Compute $H_{[i][j]}(a_i)$
5:         $V_a = V_a + \frac{H_{[i][j]}(a_i)}{2^j}$
6:         $PBF[H_{[i][j]}(a_i)]++$
7:     **end for**
8: **end for**
9: Insert V_a into the hash table

Fig. 8. Insertion algorithm in PBF-HT.

Membership_Query_Item (Input: Item a)

1: Initialize $V_a = 0$
2: **for** $(i = 1; i \leq p; i++)$ **do**
3:     **for** $(j = 1; j \leq q; j++)$ **do**
4:         Compute $H_{[i][j]}(a_i)$
5:         **if** $PBF[H_{[i][j]}(a_i)]==0$ **then**
6:             Return *False*
7:         **end if**
8:         $V_a = V_a + \frac{H_{[i][j]}(a_i)}{2^j}$
9:     **end for**
10: **end for**
11: **if** V_a is in the hash table **then**
12:     Return *True*
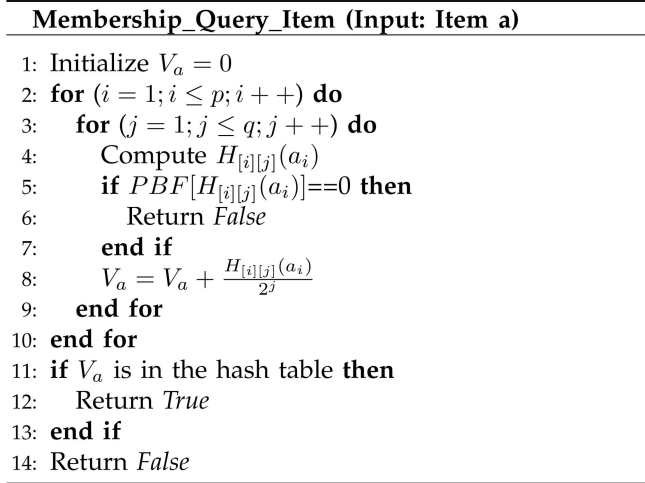13: **end if**
14: Return *False*

Fig. 9. Query algorithm in PBF-HT.

3.3.1 Adding Items

Fig. 8 presents the algorithm for inserting items in the PBF-HT structure. We need to compute the hash value of multiple attributes from q hash functions and then generate the verification value based on the improved method, as shown in Lines 4 and 5, respectively. Meanwhile, counters in the indexed positions in PBF are incremented by 1, denoted by $PBF[H_{[i][j]}(a_i)]++$. Finally, we insert the verification value of item a into the hash table.

3.3.2 Querying Items

Fig. 9 shows the multiattribute query algorithm, which realizes the two-step verification process. After computing hash values of multiple attributes, we need to check whether all p attributes exist in PBF as shown in Lines 2-10. If any $PBF[H_{[i][j]}(a_i)]$ is 0 for item a, the query returns the answer *False* showing that the queried item a does not exist. Otherwise, all hash values are added to generate the verification value V_a. If V_a is also in the hash table, we say that item a is a member.

3.3.3 Deleting Items

The deleting item operation must delete both the attributes in PBF and verification value in the hash table. Fig. 10 shows the algorithm for deleting an item a. On the contrary, to the item insertion process, the counter indexed by the position $H_{[i][j]}(a_i)$ is decremented by 1 in PBF. Afterward, the verification value of item a, V_a, is deleted from the hash table.

Delete_Item (Input: Item a)

1: Initialize $V_a = 0$
2: **for** $(i = 1; i \leq p; i++)$ **do**
3:     **for** $(j = 1; j \leq q; j++)$ **do**
4:         Compute $H_{[i][j]}(a_i)$
5:         $V_a = V_a + \frac{H_{[i][j]}(a_i)}{2^j}$
6:         $PBF[H_{[i][j]}(a_i)]--$
7:     **end for**
8: **end for**
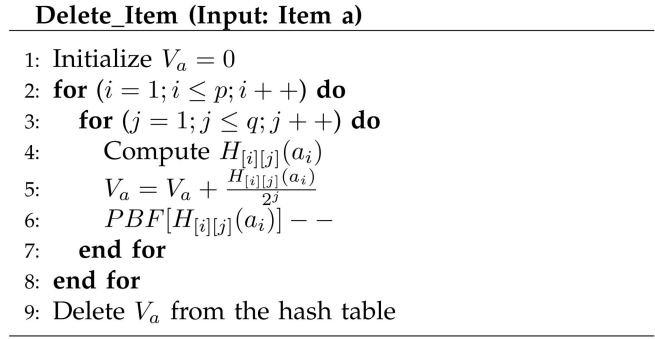9: Delete V_a from the hash table
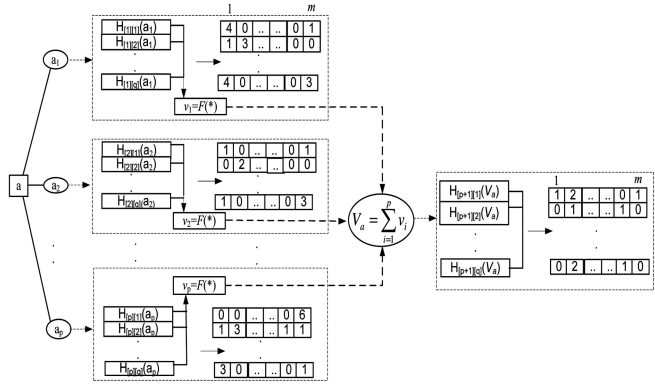

Fig. 10. Deletion algorithm in PBF-HT.

Fig. 11. The PBF-BF structure with multiattribute dependency verification.

4   AN IMPROVED STRUCTURE: PBF-BF

In this section, we extend PBF-HT structure into an improved structure PBF-BF. Although PBF-HT can provide exact match of verification values for multiattribute items in a hash table, it is not a space-efficient way to do so. The proposed PBF-BF structure makes use of a Bloom filter (BF) to store verification values to allow greater space savings than PBF-HT, but at the expense of a very small increased probability of false positives in the BF.

4.1 Proposed Structure

Fig. 11 presents PBF-BF structure, which replaces the hash table in PBF-HT with an additional Bloom filter. This additional Bloom filter, i.e., the $(p+1)$th submatrix, is used to store a verification value of an item to represent the inherent dependency of its p attributes. The PBF part, composed of p submatrices, functions the same as the PBF structure. When we add an item a with p attributes into PBF-BF, its p attributes are separately stored in p submatrices. Its verification value can be obtained using the attenuated method as in the PBF-HT structure and stored in the additional Bloom filter. Thus, similar to PBF-HT, the presence of an item in PBF-BF is confirmed when both PBF and BF answer *Yes* in the double verification process. The presence of an item is denied when any submatrix in PBF answers *No*. Compared to HT that can give exact match, the $(p+1)$th submatrix may cause false positives due to the Bloom filter property. PBF-BF is able to support operations like item insertion, query, and deletion. All operations are done on Bloom filters, having the same running time to be $O(1)$.

Insert_Item (Input: Item a)

1: Initialize $V_a = 0$
2: **for** $(i = 1; i \le p; i + +)$ **do**
3:    **for** $(j = 1; j \le q; j + +)$ **do**
4:       $k = H_{[i][j]}(a_i)$
5:       $C_{[i][j][k]} + +$
6:       $V_a = V_a + \frac{k}{2^j}$
7:    **end for**
8: **end for**
9: **for** $(j = 1; j \le q; j + +)$ **do**
10:    $k = H_{[p+1][j]}(V_a)$
11:    $C_{[p+1][j][k]} + +$
12: **end for**

Fig. 12. Insertion algorithm in PBF-BF.

Delete_Item (Input: Item a)

1: Initialize $V_a = 0$
2: **for** $(i = 1; i \le p; i + +)$ **do**
3:    **for** $(j = 1; j \le q; j + +)$ **do**
4:       $k = H_{[i][j]}(a_i)$
5:       $V_a = V_a + \frac{k}{2^j}$
6:       $C_{[i][j][k]} - -$
7:    **end for**
8: **end for**
9: **for** $(j = 1; j \le q; j + +)$ **do**
10:    $k = H_{[p+1][j]}(V_a)$
11:    $C_{[p+1][j][k]} - -$
12: **end for**

Fig. 14. Deletion algorithm in PBF-BF.

4.2 Practical Algorithms

Fig. 12 presents the insertion algorithm in the PBF-BF structure to represent multiattribute items. When adding item a, we first obtain the position of a counter to be increased by one, computed by the hash function as shown in Line 4. Line 5 increases the value of the counter and Line 6 generates the verification value V_a using the attenuated method, which will be stored in the $(p + 1)$th submatrix (the additional Bloom filter) as shown in Lines 9-12.

Fig. 13 gives the query algorithm. Given an item a with p attributes, we first compute q hash functions for each attribute and check corresponding counters. If any counter is zero, item a is not a member. Otherwise, PBF indicates that p attributes exist. We need to further verify whether item a is in the $(p + 1)$th submatrix by checking its verification value presence in this verification Bloom filter. Similarly, we compute q hash functions for the verification value (V_a). If all the counters are nonzero, we can determine that the verification value of the queried item exists in the $(p + 1)$th submatrix, which implies the existence of item a in

---

Membership_Query_Item (Input: Item a)

1: Initialize $V_a = 0$
2: **for** $(i = 1; i \le p; i + +)$ **do**
3:    **for** $(j = 1; j \le q; j + +)$ **do**
4:       $k = H_{[i][j]}(a_i)$
5:       $V_a = V_a + \frac{k}{2^j}$
6:       **if** $C_{[i][j][k]} == 0$ **then**
7:          Return *False*
8:       **end if**
9:    **end for**
10: **end for**
11: **for** $(j = 1; j \le q; j + +)$ **do**
12:    $k = H_{[p+1][j]}(V_a)$
13:    **if** $C_{[p+1][j][k]} == 0$ **then**
14:       Return *False*
15:    **end if**
16: **end for**
17: Return *True*

Fig. 13. Query algorithm in PBF-BF.

the PBF-BF structure. In this double verification process, one in PBF and one in BF, the queried item a is not a member in PBF-BF when either one returns *False*. In other words, the query service can be very fast to a nonexistent item in PBF-BF in most cases because any submatrix in PBF returning *False* can terminate the query process.

Fig. 14 shows the deletion algorithm of a multiattribute item a in PBF-BF. The deletion algorithm must delete both its p attributes and verification value in totally $(p + 1)$ submatrices. We first delete multiple attributes by decreasing counter values indexed by hash function results in p parallel Bloom filters, as shown in Lines 2-8. We then delete its verification value in the $(p + 1)$th Bloom filter as shown in Lines 9-12.

5 FALSE POSITIVE PROBABILITY

A False Positive (FP) occurs when a Bloom filter suggests that an item is in a set when in fact it is not [3]. There is some degree of probability of obtaining a false positive when using any Bloom filter. In this section, we describe and analyze the false positive probability for SBF, PBF, PBF-HT, and PBF-BF structures containing bit arrays. Although the FP analysis is given for bit arrays for Bloom filters in this section, it is applicable to Bloom filters with counters when we treat a nonzero counter as bit 1 in the counter array.

5.1 FP of SBF

A standard Bloom filter consisting of a bit array of M bits, which are initially set to 0, can store and represent a set $S = \{a_1, a_2, \ldots, a_n\}$ of n items based on the computation of q independent hash functions $\{h_1, \ldots, h_q\}$, which map the set to the bit address space $[1, \ldots, M]$. When an SBF displays the segment form, we have $M = qm$.

One can insert an item a into a Bloom filter by setting the bits of $h_i(a)$ to 1. Furthermore, one can determine the membership of item a in a set S by checking whether all $h_i(a)$ are set to 1. If not, item a is not in the set S. If they are, item a is a member of set S with an FP, which says that an item a is in set S although it is actually not. According to the conclusion, to get a low FP probability in [3], qn is set to be smaller than M ($qn < M$). Assume that a hash function selects each array position with equal probability. Theorem 5.1 gives

the FP probability approximation of an SBF, in either flat form or segment form as shown in Fig. 1.

Theorem 5.1. *The false positive probability of an SBF is*

$$f_{SBF} \approx \left(1 - e^{-\frac{qn}{M}}\right)^q = \left(1 - e^{-\frac{n}{m}}\right)^q$$

when the Bloom filter has totally M bits and q hash functions for storing n items. The probability can obtain the minimum $(1/2)^q$ or $(0.6185)^{M/n}$ when $q = (M/n)\ln 2$.

Proof. We prove the theorem for an SBF in the segment form since the proof for the flat form can be found in [3]. Assume that q hash functions are perfectly random and each hash function is associated with m ($m = M/q$ and $m > n$) bits. The probability that a particular bit in an array is set to 1 by a hash function is $\frac{q}{M}$. Thus, the probability that a bit is not set by the hash function is $\left(1 - \frac{q}{M}\right)$ (or $\left(1 - \frac{1}{m}\right)$). After inserting n items into the Bloom filter, the probability that a bit is still 0 is $(1 - \frac{q}{M})^n \approx e^{-\frac{qn}{M}}$. Thus, the false positive probability in the standard Bloom filter is $f_{SBF} = (1 - (1 - \frac{q}{M})^n)^q \approx (1 - e^{-\frac{qn}{M}})^q = (1 - e^{-\frac{n}{m}})^q$ because indexed bits in q arrays must be 1.

It is easy to check that the minimum of the false positive probability $min(f_{SBF}) = (1/2)^q \approx (0.6185)^{M/n}$ when $q = \ln 2(M/n)$. □

Obviously, the probability of false positives decreases as M (the number of bits used for storage) increases and increases as n (the number of inserted items) increases. In other words, an SBF can decrease its FP probability by increasing the storage space (M) for a fixed number of stored items. Although false positives are possible in an SBF, false negatives are not, which means that an SBF always returns *Yes* for a query of a stored item. Note that f_{SBF} also shows the probability when an SBF returns *Yes* for a randomly queried item, denoted by *hit rate* in this paper.

5.2 FP of PBF

A PBF structure is composed of p submatrices, i.e., p SBFs, to store p attributes of an item. Since each submatrix needs to store n distinct attributes, similar to n items in an SBF, we set each array with the bit address space $[1, \ldots, m]$. In the following, we describe the false positive probability of the basic PBF.

Theorem 5.2. *The false positive probability of PBF is*

$$f_{PBF} = (f_{SBF})^p \approx \left(1 - e^{-\frac{n}{m}}\right)^{pq}$$

when PBFs have totally $p \cdot M$ bits and q hash functions for storing n items with p attributes. The probability can obtain the minimum $(1/2)^{pq}$ or $(0.6185)^{pM/n}$ when $q = (M/n)\ln 2$.

Proof. The FP of PBF occurs when p submatrices return positive for a random query. Each submatrix in the PBF is in essence an SBF with M bits and q hash functions. According to Theorem 5.1, a submatrix returns positive with the probability of $f_{SBF} \approx (1 - e^{-\frac{n}{m}})^q$ to a random attribute request. Thus, we have $f_{PBF} = (f_{SBF})^p \approx (1 - e^{-\frac{n}{m}})^{qp}$. f_{PBF} can have the minimum $(1/2)^{qp}$ or $(0.6185)^{pM/n}$ for the derivative of 0 of f_{PBF} with respect to q when $q = (M/n)\ln 2$. □

The false positive probability, $f_{PBF} = (f_{SBF})^p \approx (1 - e^{-\frac{n}{m}})^{pq}$, also denotes the *hit rate* for a randomly queried item to be in the PBF. PBF using p times storage space (pM bits totally) as an SBF does can reduce the minimum FP probability to be $(0.6185)^{pM/n}$, which yields the minimum FP probability of an SBF to the power of p. Of course, an SBF can reduce its FP probability by increasing the storage space for n items. For example, when an SBF uses pM bits to store n items, its FP probability decreases to $(0.6185)^{pM/n}$. In summary, PBF does not add or lose any attribute information of items as an SBF does and when they use pM bits to store n items, they both can maintain a very low FP probability to be around $(0.6185)^{pM/n}$ when taking $q = (M/n)\ln 2$ hash functions.

5.3 FP of PBF-HT

The PBF-HT structure utilizes an extra hash table to contain verification values of items whose attributes have been stored in p submatrices in the PBF part. Thus, an FP occurs when both the PBF and HT falsely return positive for a queried item that in fact does not exist. To know the FP probability in PBF-HT, we need to calculate the FP probability that a randomly queried item has its verification value stored in the hash table. Since there are two different methods, basic and attenuated, to implement the hash table as in Section 3.2, we first give the FP probability bound for these two methods and then show the FP probability in our proposed PBF-HT structure.

Theorem 5.3. *Given n items and their verification values stored in a hash table using the basic method, the probability $f_{HT_{basic}}$ that a randomly queried item has its verification value in the hash table is bounded by $2\Phi(\frac{\sqrt{3}n}{(m-1)\sqrt{pq}}) - 1$, where $\Phi(*)$ stands for the standard Normal Distribution.*

Proof. Given an item a with p attributes, $a_1, a_2, \ldots, a_p$, let V_a be the random variable representing its verification value in the basic method. Thus, $V_a = \sum_{i=1}^{p}\sum_{j=1}^{q} H_{[i][j]}(a_i)$. The range of each hash function is $[1, m]$, which means that V_a can take any value in the range $[pq, pqm]$. $H_{[i][j]}(a_i)$ is a random variable following the Uniform Distribution with the expected value of $\frac{1+m}{2}$ and variance of $\frac{(m-1)^2}{12}$. According to the central limit theorem[1] [15], the random variable V_a satisfies a Normal Distribution with the expected value of $\mu = pq\frac{1+m}{2}$ and variance of $\sigma^2 = pq\frac{(m-1)^2}{12}$.

Let $p(V_a)$ be the probability that the verification value of a randomly queried item is the same to the one of a, i.e., V_a. If V_a is stored in the hash table, it will return positive for the presence of the queried item. Thus, given n verification values $(V_1, V_2, \ldots, V_n)$ stored in a hash table, following the normal distribution, the FP probability $f_{HT_{basic}}$ in the HT for a randomly queried item becomes $\sum_{i=1}^{n} p(V_i)$. As illustrated in Fig. 15a, the FP probability is the sum of n shaded areas. For simplicity, we only provide an upper bound which is the sum of n consecutive areas in the center as shown in Fig. 15b:

---

1. The central limit theorem states that the probability of the sample sum of independent random variables converges to a normal distribution as the sample size grows.
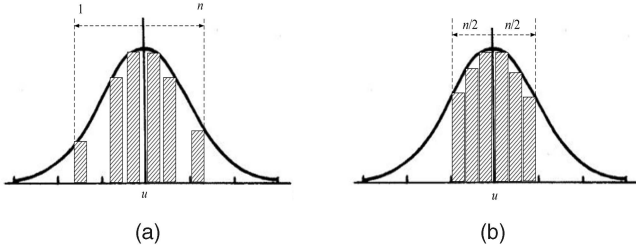
Fig. 15. Approximate upper bound representation of FP probability in a normal distribution. (a) n shaded regions in a normal distribution. (b) n shaded regions in the center as the upper bound.

$$f_{HT_{basic}} = \sum_{i=1}^{n} p(V_i)$$
$$\leq p\left(\mu - \frac{n}{2} \leq V_i \leq \mu + \frac{n}{2}\right)$$
$$= p\left(-\frac{n}{2\sigma} \leq \frac{V_i - \mu}{\sigma} \leq \frac{n}{2\sigma}\right)$$
$$= \Phi\left(\frac{n}{2\sigma}\right) - \Phi\left(-\frac{n}{2\sigma}\right) \qquad (1)$$
$$= \Phi\left(\frac{n}{2\sigma}\right) - \left[1 - \Phi\left(\frac{n}{2\sigma}\right)\right]$$
$$= 2\Phi\left(\frac{n}{2\sigma}\right) - 1$$
$$= 2\Phi\left(\frac{\sqrt{3}n}{(m-1)\sqrt{pq}}\right) - 1.$$

□

Theorem 5.4. *Given n items and their verification values stored in a hash table using the attenuated method, the probability f_{HT} that a randomly queried item has its verification value in the hash table is bounded by $2\Phi\left(\frac{\sqrt{3}2^q n}{p(m-1)(2^q-1)}\right) - 1$, where $\Phi(*)$ stands for the standard Normal Distribution.*

Proof. In the attenuated method, an item a has its verification value

$$V_a = \sum_{i=1}^{p} \sum_{j=1}^{q} \frac{H_{[i][j]}(a_i)}{2^j},$$

where

$$p\left(1 - \frac{1}{2^q}\right) \leq V_a \leq pm\left(1 - \frac{1}{2^q}\right).$$

Since the hash value $H_{[i][j]}(a_i)$ is a random variable following the Uniform Distribution, we obtain the expected value and variance of

$$\frac{H_{[i][j]}(a_i)}{2^j},$$

i.e.,

$$E\left(\frac{H_{[i][j]}(a_i)}{2^j}\right) = \frac{1+m}{2}\left(1 - \frac{1}{2^q}\right)$$

and

$$V\left(\frac{H_{[i][j]}(a_i)}{2^j}\right) = \frac{(m-1)^2}{12}\left(1 - \frac{1}{2^q}\right)^2,$$

which are constants. The verification value V_a is the sum of all $(p \times q)$ independent random variables, i.e., $\frac{H_{[i][j]}(a_i)}{2^j}$, and thus, follows the Normal Distribution, i.e.,

$$V_a \sim N\left\{\frac{(1+m)p}{2}\left(1 - \frac{1}{2^q}\right), \frac{(m-1)^2 p^2}{12}\left(1 - \frac{1}{2^q}\right)^2\right\}.$$

As shown in (1), we can obtain the upper bound of f_{HT} similarly. The only difference is the variance value of V_a in the attenuated method to affect the upper bound. Thus, considering the attenuated method that has the variance $\sigma'^2 = \frac{(m-1)^2 p^2}{12}\left(1 - \frac{1}{2^q}\right)^2$,

$$f_{HT} \leq 2\Phi\left(\frac{n}{2\sigma'}\right) - 1 = 2\Phi\left(\frac{\sqrt{3}2^q n}{p(m-1)(2^q-1)}\right) - 1,$$

where $\Phi(*)$ stands for the standard Normal Distribution. □

The PBF-HT structure involves a two-step query process, respectively, in PBF and the hash table, as shown in Section 3.3. Only when the basic PBF returns positive for the usual presence check for a queried item, does the second-step verification value check proceed with the HT. Using the attenuated method to generate verification values, different items may have the same result, causing FPs. The reason is that it is possible for a nonexistent item viewed as a member of the hash table because its verification value happens to be the one already in the hash table.

Theorem 5.5. *The false positive probability of PBF-HT structure is $f_{PBF-HT} = f_{PBF} f_{HT}$, which utilizes p submatrices to store p attributes and a hash table to store verification values through the attenuated method.*

Proof. The false positive in the PBF-HT structure occurs when false positives occur in both PBF and HT parts for a randomly queried item, which implies that $f_{PBF-HT} = f_{PBF} f_{HT}$. □

5.4 FP of PBF-BF

The PBF-BF structure utilizes a submatrix to store the verification values of items. This allows it to save space but at the cost of an increase of very small probability of false positives when compared with PBF-HT. The query operation in the PBF-BF structure also adopts a two-step verification approach. We first verify whether p attributes of a queried item exist in p submatrices and further, in the $(p+1)$th submatrix (i.e., an SBF), determine whether the verification value of those p attributes also exists. Thus, we give the FP probability of PBF-BF in Theorem 5.6, where a PBF-BF structure uses pM bits to implement p submatrices in the PBF part and M bits for the $(p+1)$th submatrix in the BF part to store n items.

Theorem 5.6. *The false positive probability of PBF-BF structure is $f_{PBF-BF} = f_{PBF} f_{SBF} \approx (1 - e^{-\frac{n}{m}})^{(p+1)q}$, which utilizes p submatrices to store p attributes and an SBF to store verification values.*

Proof. The false positive in the PBF-BF structure occurs when false positives occur in both PBF and SBF parts

for a randomly queried item, which implies that $f_{PBF-BF} = f_{PBF}f_{SBF}$.

In the PBF-BF structure, each submatrix encompasses q arrays with the range $[1, m]$. From Theorem 5.1, we have $f_{SBF} \approx (1 - e^{-\frac{n}{m}})^q$. From Theorem 5.2, we have $f_{PBF} = (f_{SBF})^p \approx (1 - e^{-\frac{n}{m}})^{pq}$. Thus, f_{PBF-BF} can be represented as $f_{PBF-BF} = (f_{SBF})^{(p+1)} \approx (1 - e^{-\frac{n}{m}})^{(p+1)q}$.   □

6   PERFORMANCE EVALUATION

We conducted both simulations and experiments to evaluate the space usage and query accuracy of SBF, PBF, PBF-HT, and PBF-BF. The SBF structure in this paper uses a concatenated array from multiple attributes as an input to hash functions and the approach is an extension of the Bloom filters in [3] for items with multiple attributes. In simulations, we generated items with multidimensional (from two to four) attributes that follow the uniform distribution. The tested queries were arbitrarily produced in the same range as stored items. Simulation results first show the impact of the generated verification value on the query accuracy in PBF-HT and PBF-BF structures. Then, we show the much lower false positive probability and smaller space requirement of PBF-HT and PBF-BF than SBF and PBF. Our experimental results are obtained by taking the average from 10 repeated experiments and the tested performance is consistent with our theoretical analysis.

6.1   Verification Values

Verification values can be used to check whether queried multiple attributes belong to one item. Inaccurate verification value generation method may lead to high false positives in which a nonexistent item is falsely considered as a member because the item has the same verification value as an existing item. We compare the false positive probability of two methods, i.e., *Basic Method (BM)* and *Attenuated Method (AM)*, based on the same hash functions and space ranges. In the simulation, each item has four attributes and each attribute is computed by six hash functions, i.e., $p = 4$ and $q = 6$, respectively.

Fig. 16 illustrates the simulation results that display the query accuracy, respectively, using BM and AM generation methods proposed in Section 3.2. The attenuated method is able to obtain much smaller false positive probability than the basic method when the address space m is set to 320 and 640. The attenuated method takes into account the sequential information of hash functions and this allows it to tremendously decrease false positives on verification values. The average false positive probability of AM can be bounded by 0.002, which is much smaller than BM.

6.2   Low False Positive Probability of PBF-HT and PBF-BF

This section describes the false positive probability of SBF, PBF, PBF-HT, and PBF-BF structures for multiattribute item queries. We select the MD5 [13] as the hash function for its well-known properties and relatively fast implementation. The hashed value for an attribute is 128 bits by calculating the MD5 signature. We set three attributes for each item and each attribute is computed using seven hash functions, i.e., $p = 3$ and $q = 7$. The storage space is set by m, where
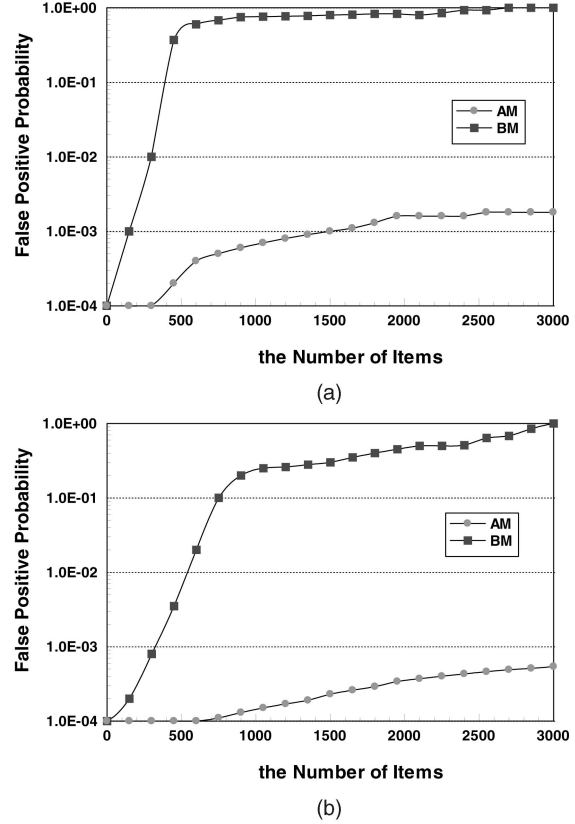

Fig. 16. False positive probability of Basic and Attenuated Methods. (a) Address space $m = 320$. (b) Address space $m = 640$.

$m = 320, m = 640, m = 1,280,$ and $m = 2,560$ counters are used in comparisons. In simulations, SBF and PBF always use the same storage space to store n items.

Fig. 17 shows the false positive probability of SBF, PBF, PBF-HT, and PBF-BF for different storage spaces. We observe that both SBF and PBF generate similar probability of false positives (curves almost overlapped) since they utilize the same space overhead to maintain multidimensional attributes of items, which is consistent with our theoretical analysis described in Section 5.2. The PBF structure has a slightly higher false positive probability than SBF. The reason is that PBF adopts the segment form to confine the hash results within a segment for each hash function. The actual false positive probability for PBF using q hash functions is $(1 - (1 - \frac{q}{M})^n)^q$, which is always at least as large as the false positive probability $(1 - (1 - \frac{1}{M})^{qn})^q$ for SBF in a flat form. Both $(1 - \frac{q}{M})^n$ and $(1 - \frac{1}{M})^{qn}$ can approximate to $e^{-\frac{qn}{M}}$ because their difference is very small.

Given a certain number of items, PBF-HT and PBF-BF structures always achieve much smaller false positive probability than SBF and PBF. The reason is that PBF-HT and PBF-BF structures check the multiattribute integrity of an item by using an extra hash table or submatrix to record verification values of items. Among them, PBF-HT structure has the smallest probability because PBF-HT uses a hash table to store the real verification values and can obtain exact matching. Instead, PBF-BF uses a Bloom filter to store verification values with false positives. However, the
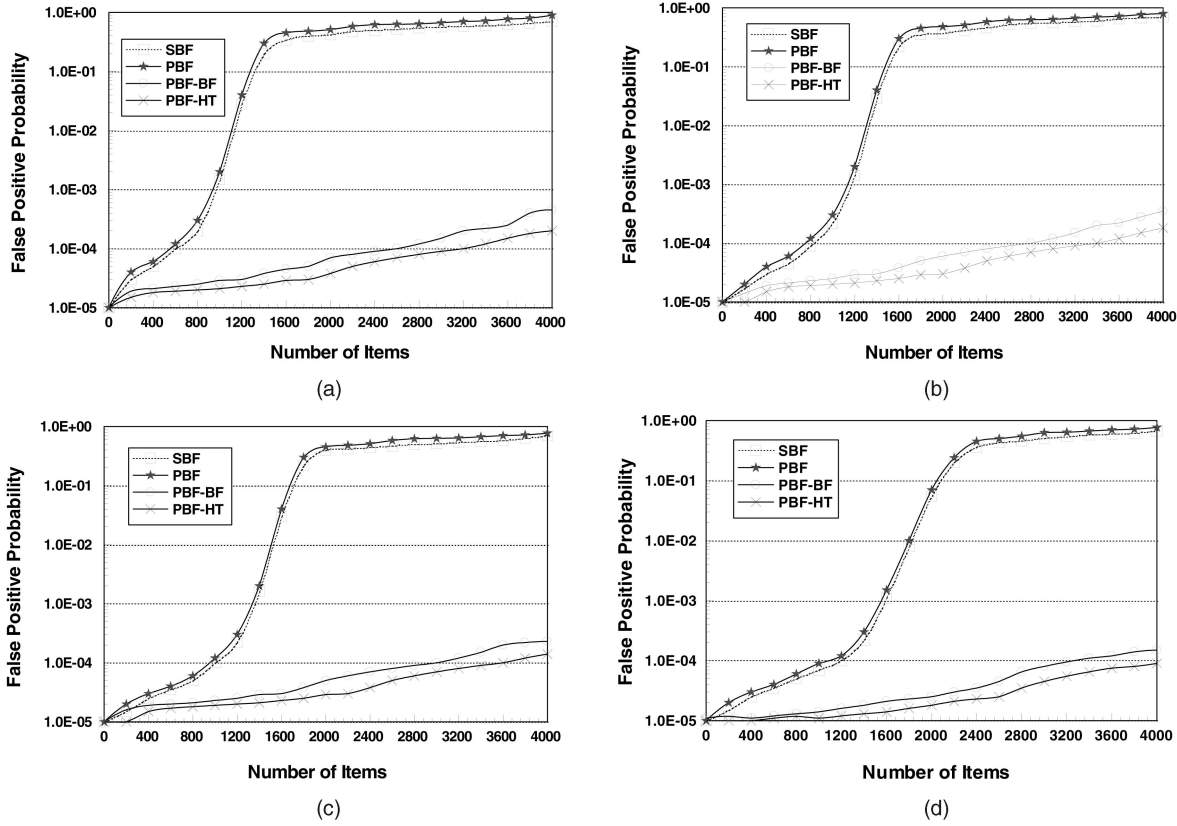
Fig. 17. False positive probability of SBF, PBF, PBF-HT, and PBF-BF. (a) Address space $m = 320$. (b) Address space $m = 640$. (c) Address space $m = 1,280$. (d) Address space $m = 2,560$.

difference of false positive probability between PBF-HT and PBF-BF is very obscure, maximally 0.01 percent, which in fact reflects that PBF-BF only produces a very small probability of false positives. Therefore, PBF-BF structure with space savings (i.e., no real value storage) could be a better choice than PBF-HT to support query services for multiattribute items. When the number of inserted items increases, the false positive probability curves of PBF-HT and PBF-BF arise smoothly because the proposed structures utilize a two-step verification process to reduce false positives, effectively tolerating increasing number of added items.

6.3 Storage Space

Fig. 18 shows the amount of space required by SBF, PBF, PBF-HT, and PBF-BF in order to maintain a given threshold of false positive probability. The amount of space required is compared when the attribute number of an item increases from two to four, while the predefined threshold of false positive probability is 0.0039.

To yield the minimum false positive probability, according to Theorem 5.2, the space used by PBF and SBF is linear to the number of items to attain a predefined threshold of false positive probability. We observe that PBF, without any attribute verification process, requires the approximate space as SBF does. However, PBF-HT exhibits the advantage to reduce storage space by using a double verification process in which the presence of a queried item is confirmed only when both PBF and HT return positive. Since PBF-HT uses a hash table to maintain real verification values of items, rather than

their hashed results in a Bloom filter, it requires a larger space than PBF-BF does as shown in Fig. 18.

Both PBF-HT and PBF-BF require smaller storage space with the same threshold of false positive probability than SBF and PBF. The space used in PBF-BF and PBF-HT increases slowly and we attribute such slow increment to the PBF-BF and PBF-HT structures that efficiently utilize the sequential information of hash functions to distinguish different attributes. Although it may require additional computation in BF and HT, PBF-BF and PBF-HT, executing a double verification process, can store more multiattribute items than other structures in the same condition of space usage while maintaining a very low false positive probability.

Fig. 19 shows results when decreasing the predefined threshold of false positive probability into 0.00098. PBF-BF explicitly displays its advantage over other Bloom filter-based variants to efficiently achieve space savings.

6.4 Implementation

We have implemented the proposed structures, including SBF, PBF, PBF-HT, and PBF-BF, to verify their feasibility and efficiency in real-world applications. We evaluate them using a public file system trace, HP trace [11]. The HP trace is a 10-day file system trace collected on a time-sharing server. The trace records multiple operations, such as *READ, WRITE, LOOKUP, OPEN,* and *CLOSE,* on file systems. We select file name, device number, and last modified time (from last "*WRITE*" operation) as three attributes of a file, which were hashed into different data structures for performance
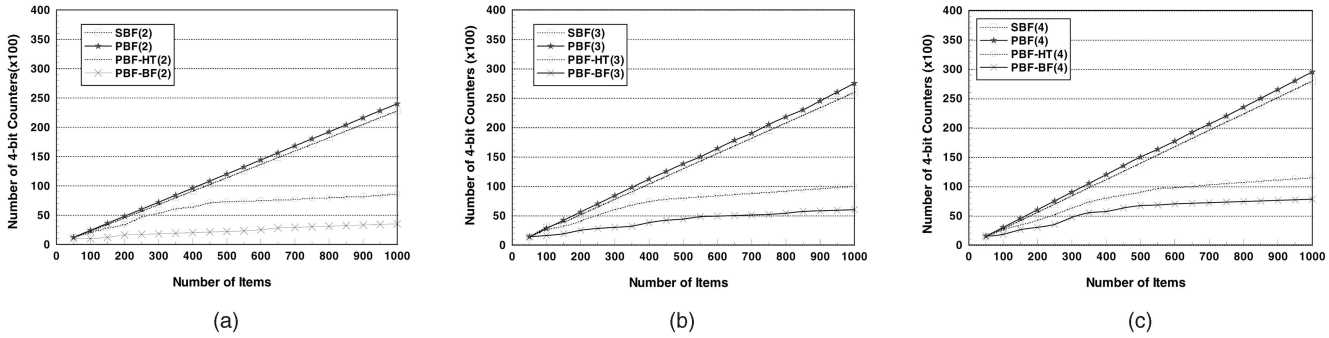
Fig. 18. Storage space required for SBF, PBF, PBF-HT, and PBF-BF to maintain a predefined threshold of false positive probability to be 0.0039. (a) Tested with two-dimensional attributes. (b) Tested with three-dimensional attributes. (c) Tested with four-dimensional attributes.
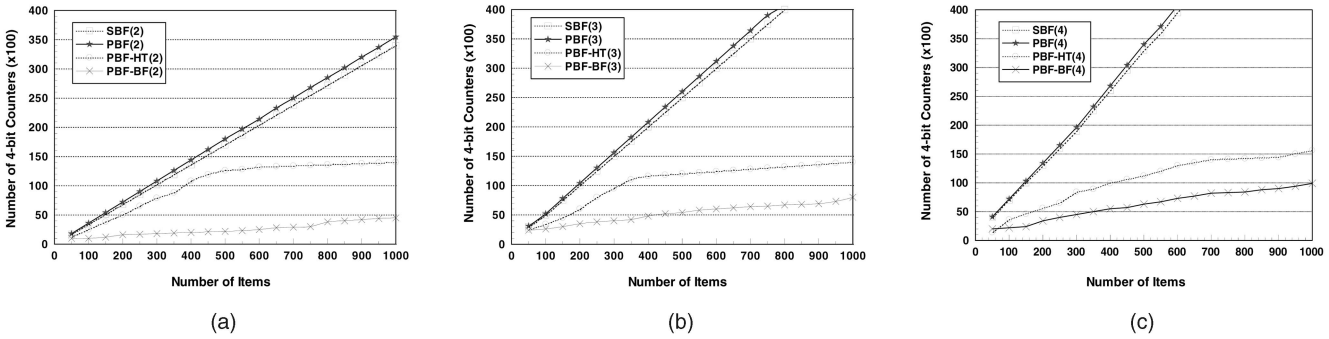

Fig. 19. Storage space required for SBF, PBF, PBF-HT, and PBF-BF to maintain a predefined threshold of false positive probability to be 0.00098. (a) Tested with two-dimensional attributes. (b) Tested with three-dimensional attributes. (c) Tested with four-dimensional attributes.

comparisons. Our experiments run on a computer with 3.2 GHz Dual Core processors and 2 GB RAM. Initially, each Bloom-filter-based structure needs to read collected trace segments into the main memory. Since our PBF-based structures are very space efficient, 2 GB main memory is large enough to contain the index structures.

Fig. 20 shows the comparison results by using a real HP trace file system. We compared PBF-HT and PBF-BF with SBF and PBF. The experimental results from real implementations illustrate that PBF and SBF present approximately the same probability of false positive, which is consistent with our theoretical analysis described in Section 5.2. In contrast to PBF and SBF, the two-step verification-based structures, i.e., PBF-HT and PBF-BF, obtain very high query accuracy by utilizing our proposed verification method. The false positive probability of PBF-HT is smaller than that of PBF-BF since the former uses a hash table to store actual verification values, which are hashed into Bloom filters in the latter, thus introducing some false positives.

Fig. 21 shows the query latency for discussed structures when using the HP trace. We observe that SBF has the largest query latency since it has the overhead to concatenate all attributes into a long vector. Nonetheless, it requires more time to get the hashed result for a long vector. In contrast, PBF produces the smallest query latency since it uses parallel execution on all independent attributes and can obtain query results very quickly. Meanwhile, PBF only needs to carry out one-step verification to check the presence of each attribute in a submatrix, thus showing smaller latency than both PBF-HT and PBF-BF. Compared with PBF-BF, PBF-HT has a larger latency since a hash table needs to attach short linked

lists to conflicted buckets and these lists must be checked to obtain final results for queries.

Table 1 presents the required physical storage space of all structures. The comparison results are normalized to SBF to keep a bounded false positive probability in each data structure by running the HP trace. PBF requires the same space as SBF. However, their physical storage space is much larger than the one used in PBF-HT and PBF-BF. Compared with SBF and PBF structures, PBF-HT and PBF-BF utilizing an additional verification structure can greatly reduce storage space. When the bounded false positive probability (FP Prob.) decreases from 10 to 2 percent, the space saving merit becomes more outstanding in PBF-HT and PBF-BF. Moreover, PBF-BF using a Bloom filter to store verification
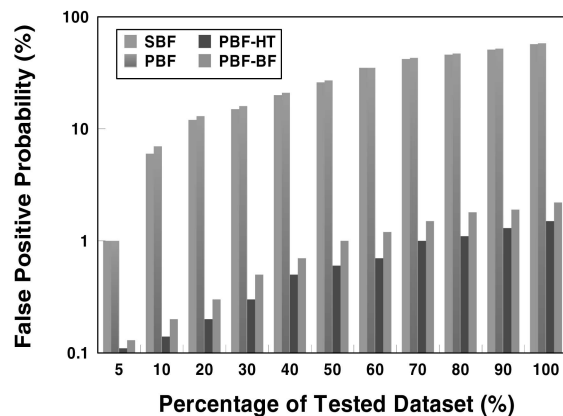

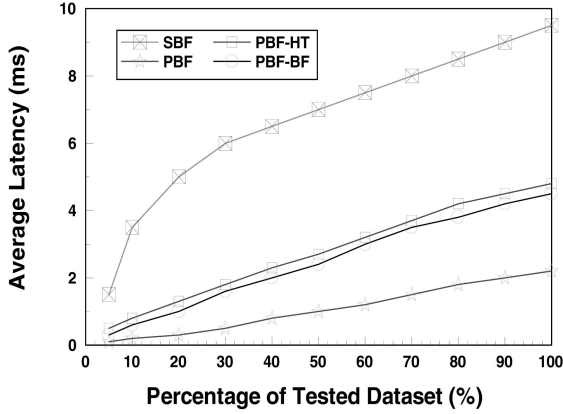Fig. 20. The false positive probability for SBF, PBF, PBF-HT, and PBF-BF.

Fig. 21. The query latency for SBF, PBF, PBF-HT, and PBF-BF.

TABLE 1
Required Storage Space Normalized to SBF in a
Bounded False Positive Probability

FP Prob.	SBF	PBF	PBF-HT	PBF-BF
10%	1.000	1.005	0.196	0.058
8%	1.000	1.006	0.178	0.054
6%	1.000	1.008	0.157	0.049
4%	1.000	1.011	0.132	0.042
2%	1.000	1.015	0.101	0.032

values further shows its space efficiency advantage over PBF-HT that uses a hash table to have the same function.

7 RELATED WORK

A Bloom filter can be used to support membership queries [3] because it uses a simple space-efficient data structure to represent a set. Bloom filters are broadly applied to network-related applications, e.g., they are used to find heavy flows for stochastic fair blue queue management scheme [16], summarize contents to help the global collaboration [17], and achieve optimal replacement [4] and the longest prefix matching [18]. Bloom filters provide a useful tool to assist the network routing, such as route lookup [19], packet classification [20], and active detection [21].

Standard Bloom filters have been the basis of many other types of Bloom filters such as compressed Bloom filters [22], hierarchical Bloom filter arrays [23], space-code Bloom filters [24], spectral Bloom filters [25], and counting Bloom filters [12]. Compressed Bloom filters can achieve a smaller false positive probability using a function of compressed size than a standard Bloom filter without compression. Hierarchical Bloom filters maintain two levels of Bloom filter arrays to represent the metadata location and distribution information. Space-code Bloom filters and spectral Bloom filters are used to represent a multiset and allow querying as to how many occurrences of an item there are in a given set. Both of these Bloom filters are also suitable for representing a static set whose size can be well estimated in advance. Counting Bloom filters replace an array of bits with counters, allowing them to count the number of items hashed to a location and to deal with item update and deletion operations. They can be very useful for handling a set that is changing over time, where items can be added, updated, or removed. Using a counter rather than a bit allows a record to be kept of how many times a counter has been indexed. When an item is deleted, the corresponding counters are decremented. If an entry in a counting Bloom filter becomes 0, the filter does not contain its corresponding item. It has been shown that 4 bits per counter should suffice to avoid counter overflows in most applications [12].

There is a great deal of room to develop variants or extensions of Bloom filters for specific applications. When space is an issue, a Bloom filter can be an excellent alternative to keeping an explicit list. Group-based Hierarchical Bloom filter Array (G-HBA) is designed to manage metadata in large-scale file systems and improve metadata operation performance [26]. In [24], the authors designed a data structure called an exponentially decaying Bloom filter (EDBF) that encoded probabilistic routing tables in a highly compressed manner and allowed for efficient aggregation and propagation. Based on research into stale streaming data information, stable Bloom filters [27] were proposed to detect the duplicates of streaming data that had a tight upper bound of false positive rates. Attenuated Bloom filters are used to perform context discovery in ad hoc networks [28], where Bloom filters can represent context information to facilitate fast queries. Beyond Bloom filters [29] further considered the question of how to compactly represent concurrent state machines and took into account approximate finite state machines by allowing a "don't know" state in the Bloom filter to represent set membership.

In addition, network applications emphasize a strong need to engineer hash-based data structures as these can achieve faster lookup speeds with better worst-case performance in practice. From the engineering perspective, the authors of [30] extended the multiple-hashing Bloom filter by using a small amount of multiport on-chip memory, which can support better throughput for router applications based on hash tables. Not much previous work has addressed the efficient representation of items with multiple attributes, which may be common in many network applications. Compared to the work in [10], this paper presents a new data structure PBF-BF and gives more detailed false positive probability analysis on PBF-based Bloom filter structures.

8 CONCLUSION

In this paper, we presented a comprehensive solution to support the representation and membership queries of multiattribute items with accurate query results. The solution consists of three Bloom filter-based structures, PBF, PBF-HT and PBF-BF, in series to store and represent multiattribute items, with each structure supporting queries at different levels of accuracy. For random queries, we showed that the hit rate is very low for proposed PBF-based structures. The false positive probability analysis in the paper reveals that SBF and PBF yield the same minimum false positive probability using the same storage space. However, PBF-HT and PBF-BF can significantly reduce the false positive probability because of their verification value storage at a HT

or BF, respectively. The explored PBF-based structures also support practical algebraic operations [10], facilitating queries among distributed data sets. Through simulations and real experiments, we demonstrated that the novel structures, PBF-HT and PBF-BF, can efficiently be applied in network services for their small space requirement, short delay to queries, and very low false positive probability.

ACKNOWLEDGMENTS

REFERENCES

[1] H. Burton, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Comm. ACM,* vol. 13, no. 7, pp. 422-426, 1970.
[2] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci, "MultiLayer Compressed Counting Bloom Filters," *Proc. IEEE INFOCOM,* pp. 311-315, 2008.
[3] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Math.,* vol. 1, pp. 485-509, 2005.
[4] A. Pagh, R. Pagh, and S. Rao, "An Optimal Bloom Filter Replacement," *Proc. 16th Ann. ACM-SIAM Symp. Discrete Algorithms,* pp. 823-829, 2005.
[5] M. Zhong, P. Lu, K. Shen, and J. Seiferas, "Optimizing Data Popularity Conscious Bloom Filters," *Proc. ACM Symp. Principles of Distributed Computing (PODC),* 2008.
[6] F. Hao, M. Kodialam, and T. Lakshman, "Building High Accuracy Bloom Filters Using Partitioned Hashing," *Proc. ACM SIGMETRICS,* pp. 277-288, 2007.
[7] F. Hao, M. Kodialam, and T.V. Lakshman, "Incremental Bloom Filters," *Proc. IEEE INFOCOM,* pp. 1741-1749, 2008.
[8] B. Donnet, B. Baynat, and T. Friedman, "Retouched Bloom Filters: Allowing Networked Applications to Trade Off Selected False Positives Against False Negatives," *Proc. Int'l Conf. Emerging Networking Experiments and Technologies (CoNEXT),* 2006.
[9] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and Network Application of Dynamic Bloom Filters," *Proc. IEEE INFOCOM,* 2006.
[10] Y. Hua and B. Xiao, "A Multi-Attribute Data Structure with Parallel Bloom Filters for Network Services," *Proc. IEEE Int'l Conf. High Performance Computing (HiPC),* pp. 277-288, Dec. 2006.
[11] E. Riedel, M. Kallahalla, and R. Swaminathan, "A Framework for Evaluating Storage System Security," *Proc. Conf. File and Storage Technologies (FAST),* pp. 15-30, 2002.
[12] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: A Scalable Wide Area Web Cache Sharing Protocol," *IEEE/ACM Trans. Networking,* vol. 8, no. 3, pp. 281-293, June 2000.
[13] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography.* CRC Press, 1997.
[14] S.C. Rhea and J. Kubiatowicz, "Probabilistic Location and Routing," *Proc. IEEE INFOCOM,* 2002.
[15] A. Barron, "Entropy and the Central Limit Theorem," *The Annals of Probability,* vol. 14, no. 1, pp. 336-342, 1986.
[16] W. chang Feng, D.D. Kandlur, D. Saha, and K.G. Shin, "Stochastic Fair Blue: A Queue Management Algorithm for Enforcing Fairness," *Proc. IEEE INFOCOM,* 2001.
[17] F.M. Cuenca-Acuna, C. Peery, R.P. Martin, and T.D. Nguyen, "PlantP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities," *Proc. Conf. High Performance Distributed Computing (HPDC),* 2003.
[18] S. Dharmapurikar, P. Krishnamurthy, and D.E. Taylor, "Longest Prefix Matching Using Bloom Filters," *Proc. ACM SIGCOMM,* 2003.
[19] A. Broder and M. Mitzenmacher, "Using Multiple Hash Functions to Improve IP Lookups," *Proc. IEEE INFOCOM,* 2001.
[20] F. Baboescu and G. Varghese, "Scalable Packet Classification," *Proc. ACM SIGCOMM,* 2001.
[21] B. Xiao, W. Chen, and Y. He, "A Novel Technique for Detecting DDoS Attacks at the Early Stage," *J. Supercomputing,* vol. 36, pp. 235-248, 2006.
[22] M. Mitzenmacher, "Compressed Bloom Filters," *IEEE/ACM Trans. Networking,* vol. 10, no. 5, pp. 604-612, Oct. 2002.
[23] Y. Zhu, H. Jiang, J. Wang, and F. Xian, "HBA: Distributed Metadata Management for Large Cluster-Based Storage Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 19, no. 6, pp. 750-763, June 2008.
[24] A. Kumar, J. Xu, and E.W. Zegura, "Efficient and Scalable Query Routing for Unstructured Peer-to-Peer Networks," *Proc. IEEE INFOCOM,* 2005.
[25] C. Saar and M. Yossi, "Spectral Bloom Filters," *Proc. ACM SIGMOD,* 2003.
[26] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, "Scalable and Adaptive Metadata Management in Ultra Large-Scale File Systems," *Proc. Int'l Conf. Distributed Computing Systems (ICDCS),* pp. 403-410, 2008.
[27] F. Deng and D. Rafiei, "Approximately Detecting Duplicates for Streaming Data Using Stable Bloom Filters," *Proc. ACM SIGMOD,* 2006.
[28] F. Liu and G. Heijenk, "Context Discovery Using Attenuated Bloom Filters in Ad-Hoc Networks," *J. Internet Eng.,* vol. 1, no. 1, pp. 49-58, 2007.
[29] F. Bonomi, M. Mitzenmacher, R. Panigrah, S. Singh, and G. Varghese, "Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines," *Proc. ACM SIGCOMM,* 2006.
[30] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing," *Proc. ACM SIGCOMM,* 2005.

Bin Xiao received the BSc and MSc degrees in electronics engineering from Fudan University, China, in 1997 and 2000, respectively, and the PhD degree in computer science from the University of Texas at Dallas in 2003. Currently, he is an assistant professor in the Department of Computing at Hong Kong Polytechnic University, Hong Kong. His research interests include communication and security in computer networks, peer-to-peer networks, and wireless mobile ad hoc and sensor networks. He is a member of the IEEE.

Yu Hua received the bachelor and PhD degrees in computer science from Wuhan University, China, in 2001 and 2005, respectively. Currently, he is an assistant professor in the School of Computer at Huazhong University of Science and Technology, China. He was a research assistant in the Department of Computing at Hong Kong Polytechnic University in 2006. His research interests include distributed computing and network storage. He is a member of the IEEE and the IEEE Computer Society.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.