A Multi-Partitioning Approach to Building Fast and Accurate Counting Bloom Filters

Kun Huang†, Jie Zhang‡, Dafang Zhang‡, Gaogang Xie†, Kave Salamatian+, Alex X. Liu⋆, and Wei Li‡

†Institute of Computing Technology, CAS, Beijing, China
‡Hunan University, Changsha, China
+Universite de Savoie, Chambery, France
⋆Michigan State University, East Lansing, U.S.A

{huangkun09, xie}@ict.ac.cn, {jiezhang, dfzhang}@hnu.edu.cn, kave.salamatian@univ-savoie.fr, alexliu@cse.msu.edu

***Abstract*—Bloom filters are space-efficient data structures for fast set membership queries. Counting Bloom Filters (CBFs) extend Bloom filters by allowing insertions and deletions to support dynamic sets. The performance of CBFs is critical for various applications and systems. This paper presents a novel approach to building a fast and accurate data structure called Multiple-Partitioned Counting Bloom Filter (MPCBF) that addresses large-scale data processing challenges. MPCBF is based on two ideas: reducing the number of memory accesses from *k* (for *k* hash functions) in the standard CBF to only one memory access in the basic MPCBF-1 case, and a hierarchical structure to improve the false positive rate. We also generalize MPCBF-1 to MPCBF-*g* to accommodate up to *g* memory accesses. Our simulation and implementation in MapReduce show that MPCBF outperforms the standard CBF in terms of speed and accuracy. Compared to CBF, at the same memory consumption, MPCBF significantly reduces the false positive rate by an order of magnitude, with a reduction of processing overhead by up to 85.9%.**

Keywords- Bloom filter; hashing; hierarchical structure; packet processing; mapreduce

I. INTRODUCTION

A Bloom filter [1] is a space-efficient probabilistic data structure used to test set membership. In Bloom filters false positives, i.e. returning that an element is in the set when it is not, are possible, but false negatives, i.e., answering that an element is not in the set when it is, are not possible. Elements can be added to the set, but not removed. With the number of elements being added in the set increasing, the probability of false positive also increases. Due to their simplicity and efficiency, Bloom filters have found widespread applications in all areas of computer science [2]. In particular, networking applications, such as web caching, peer-to-peer application, IP route lookup, etc., are major applications of Bloom filters.

One of the major drawbacks of Bloom filters is the impossibility of removing an element from the set, making difficult the application of Bloom filters to dynamic sets. A Counting Bloom Filter (CBF) [3] is a well-known variant of Bloom filters, which allows the set to change dynamically via insertions and deletions of elements. CBF achieves this by using *c*-bit (generally set to 4 bits for most applications) counter instead of a single bit for each location of the filter. This leads to multiplying by *c* the memory needed to store a CBF.

Due to their wide usage in applications, there are several driving factors that motivate the improvement of CBFs in terms of fast access and high accuracy. Data stream processing is a major application area of CBFs. In particular, modern high-end routers require the use of fast CBFs for different packet processing functions like packet forwarding and packet classification at line-speed.

While CBFs are amenable to easy hardware implementations over FPGA/ASIC, however, the available small fast memory (on-chip SRAMs) is not sufficient to build a fast CBF in hardware. Several parallel CBF-based solutions [4-10] that implement fast packet processing applications have recently been proposed. Nevertheless, they all require prohibitive amount of high-bandwidth on-chip SRAMs to run multiple CBFs in parallel, which impedes their usage in practice.

CBFs are characterized by three performance metrics: memory consumption, processing overhead, and false positive rate. The memory consumption is the membership counter vector size of CBF, where generally up to four bits per counter are used to support efficient insertions and deletions. The processing overhead usually determines the throughput of CBF. CBF often requires *k* memory accesses to the membership counter vector for each query or update operation, where *k* denotes the number of hash functions. The false positive rate is the probability that an element not belonging to the set described by CBF is claimed to belong to it. There is a balance between the false positive rate, memory consumption, and processing overhead. Decreasing the false positive rate of CBF entails increasing its memory consumption or processing overhead through adding more hashes to CBF. In this paper, our goal is to achieve a significant reduction in both the processing overhead and the false positive rate, at the same memory consumption.

Our work is inspired from one memory access Bloom filter (BF-1) described in [11]. BF-1 requires only one memory access to the membership vector for each query, instead of *k* memory accesses for the standard Bloom filter. The penalty is that BF-1 has a larger false positive rate than the standard Bloom filter. The penalty becomes larger when the idea of BF-1 is extended to CBF, due to larger memory requirements of CBF. We address this drawback of BF-1 by improving the false positive rate of the generalized CBF, while maintaining one or a few memory accesses and small memory usage.

This work is supported in part by the National Basic Research Program of China (973) under Grant No.2012CB315801, the National Science Foundation of China under Grant No.61100171, No.61173167, No. 61272546 and No. 61061130562, the National Science and Technology Major Project of China under Grant No. 2012ZX03002016, and NSF CNS-1017588.

This paper makes the following main contributions:

- We propose a multi-partitioning approach that combines the two ideas of one memory access and a hierarchical structure to build a fast and accurate data structure called Multiple-Partitioned Counting Bloom Filter with one memory access (MPCBF-1) that performs a membership query with a single memory access and a low false positive rate.
- We generalize MPCBF-1 to MPCBF-*g* that allows *g* memory accesses and decreases the false positive rate. MPCBF-*g* is to improve the false positive rate at the cost of higher processing overhead. Analytic results show that MPCBF-*g* achieves faster access and higher accuracy than the standard CBF.
- We conduct experiments on synthetic and realistic data sets to evaluate the performance of MPCBF. Experimental results show that compared to the standard CBF, MPCBF significantly reduces the false positive rate by an order of magnitude as well as the processing overhead by up to 85.9%, at the same memory consumption. We also implement MPCBF in MapReduce to accelerate reduce-side joins for large-scale data processing.

The rest of the paper is organized as follows. Section 2 introduces the background and related work on CBFs. In Section 3, we describe and analyze MPCBF-1 and MPCBF-*g*. Section 4 reports experimental results on the performance of MPCBF. Section 5 presents our MPCBF implementation in MapReduce. Finally, Section 6 concludes this paper.

II. BACKGROUND

A. Bloom Filters and CBFs

Bloom filters are simple space-efficient data structures for representing the elements belonging to a set $S=\{x_1,x_2,\cdots,x_n\}$ and enabling fast approximate membership queries. A Bloom filter consists of an *m*-bit vector, initially all set to 0, and *k* independent hash functions $h_1,h_2,\cdots,h_k$, each one mapping an element to a random index in the range $\{0,\cdots,m-1\}$. Inserting a new element *x* into a Bloom filter consists of obtaining *k* indices $h_1(x),\cdots,h_k(x)$. Thereafter the bits at position $h_i(x)$ ($1\le i\le k$) in the *m*-bit vector are set to 1. A CBF [3] extends the above-described algorithm by replacing the *m*-bit vector with a counter vector of length *m*, where each counter is a *c*-bit value; in a CBF the counters at position $h_i(x)$ in the counter vector are incremented. The counters are used to track the number of elements currently hashed to the corresponding locations. The counter enables deletion; if an element has to be removed from CBF, its *k* indices are calculated and the counters at position $h_i(x)$ are simply decremented. An issue can occur when a counter overflow happens. However four bits per counter have been shown to suffice for most applications. This means that the memory usage of CBF, *i.e.*, the length of the *m*-counter vector, is around 4 times the memory usage of the standard Bloom filter.

One can check membership in the Bloom filter and CBF by deriving *k* indices $h_i(x)$ and checking in the membership vector if they are all set to 1 (or larger than 0 for CBF). A false positive occurs when an element *y* not in the Bloom filter has indices $h_1(y),\cdots,h_k(y)$ that are all set in the vector. The false positive rate of a Bloom filter for an element is computed as follows:

$$f=(1-(1-\frac{1}{m})^{nk})^k\approx(1-e^{-kn/m})^k \qquad (1)$$

where *n* is the number of elements in the set *S*, *m* is the size of the vector, and *k* is the number of hash functions. Increasing the number *k* of hash functions reduces the false positive rate for a given ratio of *m/n*. The false positive rate is optimized as $f\approx(1/2)^k$ when $k=(m/n)\ln 2$. For example, when *m/n=10* and *k=7*, the false positive rate *f* is about 0.008.

B. Related Work on CBFs

Recently, Bloom filters and CBFs have been widely used in a large variety of networking applications [4-16]. As the standard CBF has larger memory requirements, several variants have been proposed to minimize the memory usage of CBF and to optimize the use of scarce and expensive SRAMs space. *d*-left CBF (dlCBF) [17] is a simple hash-based alternative based on *d*-left hashing and fingerprints. dlCBF offers the same functionality as CBF, but need less than half the memory at the same false positive rate. Rank-indexed CBF (RCBF) [18] is a compact alternative based on rank-index hashing. In order to avoid the high overhead of pointer storage, RCBF uses a hierarchical index structure for chaining the fingerprints at each location. RCBF outperforms the standard CBF in memory by a factor of above three for a false positive rate of 1%, and also outperforms dlCBF in memory by 27% at the same false positive rate. Multilayer Compressed CBF (ML-CCBF) [19] is also a compact alternative that combines a hierarchical structure with Huffman codes to reduce the memory usage. In comparison to the standard CBF, ML-CCBF reduces both the memory by up to 50% and the lookup time.

The proposed approach in this paper also takes advantage of a hierarchical structure that is borrowed from RCBF and ML-CCBF. However, differently from previous work, we use the hierarchical structure to decrease the false positive rate, instead of targeting the memory consumption.

Some other variants have been proposed to improve the false positive rate and the processing overhead of Bloom filters and CBFs. The main idea of the power of two choices [20] is introduced to use two separate groups of hash functions for inserting each element and checking the membership. An improved variant [21] uses partitioned hashing to improve the accuracy while avoiding more hashing. This variant works by partitioning elements into multiple groups and selecting proper combination of hash functions for each group. In [22], the hash selection in [21] is simplified by using only two hash functions that are linearly combined. Variable-Increment CBF (VI-CBF) [23] is a generalization of CBF that uses variable increments to improve the efficiency of CBFs and their variants. Unlike

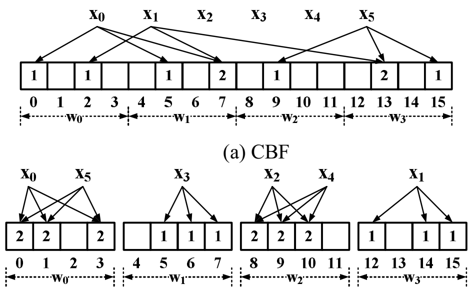


Figure 1. CBF and PCBF-1 with n=6, m=16, and k=3.

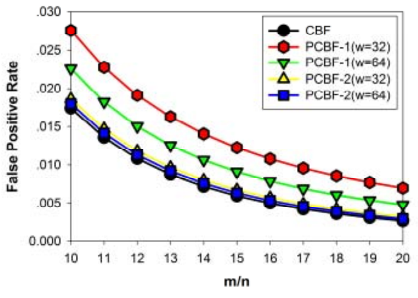


Figure 2. False positive rates of CBF, PCBF-1 and PCBF-2 with different word sizes.

CBFs, VI-CBF uses a hashed variable increment to update the hashed counters instead of a unit increment, achieving a lower false positive rate. However, all these variants still have a large processing overhead, as they require k memory accesses for a membership query.

The most closely related work to ours is one memory access Bloom filter (BF-1) [11], where the bit vector is partitioned into an array of continuous words, and each element is hashed to k bits in a randomly selected word for checking the membership. BF-1 requires only one memory access per query, improving the query overhead. Although BF-1 can be generalized to BF-g with g memory accesses, it has a larger false positive rate than the standard Bloom filter. The false positive rate increases more when the idea of BF-1 is naively extended to CBFs, due to larger memory usage of CBFs.

In this paper, we consider the combination of the two ideas of one memory access and a hierarchical structure to build a fast and accurate Multiple-Partitioned CBF (MPCBF), which outperforms the standard CBF and previous variants.

III. MULTIPLE-PARTITIONED CBF

For the purpose of clarity, we describe three variants of CBF incrementally in this section. First, we present a naïve variant called Partitioned CBF, which only uses the idea of one memory access to achieve fast access. Second, we present a basic MPCBF with one memory access, which uses the combination of two ideas of one memory access and a hierarchical structure to improve both the processing overhead and the false positive rate. Finally, we present a generalized MPCBF with g memory accesses for the accuracy-overhead trade-off.

A. Partitioned CBF

The standard CBF requires k memory accesses for a membership query, one for each hash function. As said before the optimal value of k is $(m/n)\cdot\ln 2$, which leads to too large an overhead to be used practically. We present a naïve variant of CBF called Partitioned CBF (PCBF), which uses the idea of one memory access to decrease the query overhead and build a fast CBF.

1) PCBF with One Memory Access

We now describe a PCBF with one memory access called PCBF-1. PCBF-1 is essentially an array of l words, each w bits long. The width w of a word is chosen so that it can be fetched from the memory to the processor in a single memory access. For example, today's general-purpose processors use a word of 32 bits or 64 bits. The words of PCBF-1 contain the counters that we are assuming to be 4 bits in the forthcoming. Therefore, the membership vector length of PCBF-1 is $m = l \times w/4$.

Any element to be inserted, deleted or queried in PCBF-1 is first hashed to a randomly selected word, and then hashed to k locations in the word. In other words we first use one hash function $\mathrm{H}_1(x)$ to hash an element x into one of the l words (requiring $\log_2 l$ hash bits). We then use k hash functions $h_1(x), h_2(x), \cdots, h_k(x)$ to hash x to k counters in the word, which requires $k\log_2(w/4)$ hash bits to locate these counters. Hence, PCBF-1 requires only one memory access per operation and an access bandwidth of $\log_2 l + k\log_2(w/4)$ bits, while the standard CBF requires k memory accesses and an access bandwidth of $k\log_2(l\times w/4) = k\log_2 l + k\log_2(w/4)$ bits.

Fig. 1 shows an example of CBF and PCBF-1 with n=6, m=16, and k=3. In CBF, each of elements $x_0, x_1, \cdots, x_5$ is hashed to three counters across four words w_0, w_1, w_2, w_3, which requires three memory accesses and an access bandwidth of $3\times\log_2 16 = 12$ bits. In PCBF-1, the vector of 16 counters is partitioned into four separate words, and each element is hashed to three counters in a word, which requires only one memory access and an access bandwidth of $\log_2 4 + 3\log_2 4 = 8$ bits.

We derive the false positive rate f_{P-1} of PCBF-1 as follows. Let F be the false positive event, E be a random variable for the number of elements that are hashed to the same word, and j be a constant in the range $\{0,\cdots,n\}$. When $E = j$, the conditional probability for F is computed as follows:

$$P(F \mid E=j) = (1-(1-\frac{1}{w/4})^{jk})^k$$

Because each element is hashed to one of $l = 4m/w$ words with equal probability, E follows the binomial distribution $B(n, 1/l)$:

$$P(E=j) = \binom{n}{j}(\frac{1}{l})^j(1-\frac{1}{l})^{n-j} = \binom{n}{j}(\frac{1}{4m/w})^j(1-\frac{1}{4m/w})^{n-j}$$

Hence, the false positive rate f_{P-1} is computed as follows:

$$f_{P-1} = P(F) = \sum_{j=0}^{n}\left(P(E=j)\cdot P(F\mid E=j)\right)$$
$$= \sum_{j=0}^{n}\left(\binom{n}{j}\left(\frac{1}{4m/w}\right)^{j}\left(1-\frac{1}{4m/w}\right)^{n-j}\cdot\left(1-\left(1-\frac{1}{w/4}\right)^{jk}\right)^{k}\right) \quad (2)$$

Even if increasing the word size w decreases the false positive rate f_{P-1}, PCBF-1 has a larger false positive rate than the standard CBF. This can be seen in Fig. 2 that depicts the false positive rates of CBF and PCBF-1 with different word sizes. This can be explained by observing that the standard CBF hashes an element to a random number in the long range $\{1,\cdots,m\}$, while PCBF-1 hashes it into the shorter range $\{1,\cdots,w\}$. Therefore, when w increases the false positive rate of PCBF-1 converges to that of CBF.

2) *PCBF with g Memory Accesses*

In order to attain a lower false positive rate, we can extend PCBF-1 to PCBF-*g* that enables up to *g* memory accesses. PCBF-*g* uses *g* hash functions that hash an element into *g* words (instead of one word for PCBG-1), and then divides *k* hash functions evenly among these words, *i.e.*, k/g hash functions to encode and query the element for each word. Hence, PCBF-*g* requires *g* memory accesses per operation and an access bandwidth of $g(\log_2 l + k/g\cdot\log_2(w/4))$ bits.

Similarly to PCBF-1, we can derive the false positive rate f_{P-g} of PCBF-*g*. Let F' be the false positive event, E' be a random variable for the number of times that a word is selected to store an element, and j be constant in the range $\{0,\cdots,gn\}$. The conditional probability for F' when $E'=j$ is:

$$P(F'\mid E'=j) = \left(1-\left(1-\frac{1}{w/4}\right)^{jk/g}\right)^{k/g}$$

E' follows the binomial distribution $B(gn, 1/l)$:

$$P(E'=j) = \binom{gn}{j}\left(\frac{1}{l}\right)^{j}\left(1-\frac{1}{l}\right)^{gn-j} = \binom{gn}{j}\left(\frac{1}{4m/w}\right)^{j}\left(1-\frac{1}{4m/w}\right)^{gn-j}$$

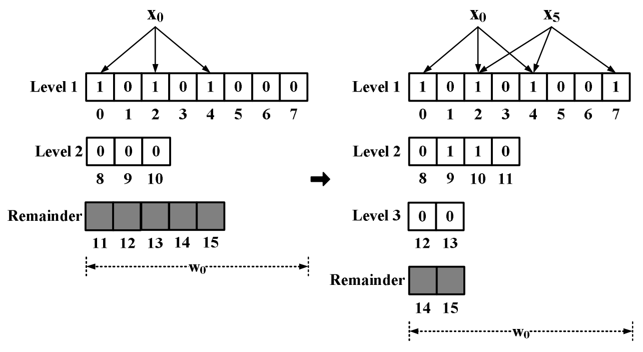
Hence, the false positive rate f_{P-g} is computed as follows:

$$f_{P-g} = \left(P(F')\right)^{g} = \left(\sum_{j=0}^{gn}\left(P(E'=j)\cdot P(F'\mid E'=j)\right)\right)^{g}$$
$$= \left(\sum_{j=0}^{gn}\left(\binom{gn}{j}\left(\frac{1}{4m/w}\right)^{j}\left(1-\frac{1}{4m/w}\right)^{gn-j}\cdot\left(1-\left(1-\frac{1}{w/4}\right)^{jk/g}\right)^{k/g}\right)\right)^{g} \quad (3)$$

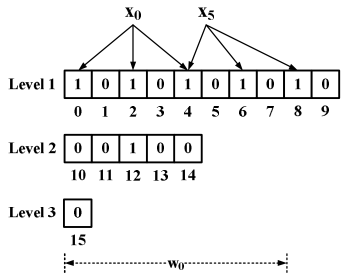
The false positive rate f_{P-g} of PCBF-*g* is much smaller than f_{P-1} of PCBF-1. Nevertheless, f_{P-g} is still larger than the false positive rate of the standard CBF. This can be seen in Fig. 2. Based on the observation, we develop a novel approach to achieving lower false positive rate, while maintaining smaller processing overhead.

B. *MPCBF with One Memory Access*

We propose a multi-partitioning approach to building a fast and accurate data structure called Multiple-Partitioned Counting Bloom Filter (MPCBF). MPCBF combines the two ideas: a single (or limited number of) memory access and a hierarchical structure to improve the processing overhead as well as the false positive rate, at the same memory consumption. In this section, we describe a basic MPCBF with one memory access called MPCBF-1.



(a) HCBF



(b) Improved HCBF

Figure 3. HCBF and improved HCBF in a word w_0.

1) *Hierarchical Counting Bloom Filter*

The key idea of MPCBF-1 is to partition the counter vector into an array of words that are further partitioned into Hierarchical Counting Bloom Filters (HCBF). A HCBF is composed of *d* levels containing *d* sub-vectors $v_1, v_2, \cdots, v_d$, each with a size $b_i = |v_i|$ bits ($1\le i\le d$). A HCBF is managed by using *k* hash functions $h_1,\cdots,h_k$ that hash an element *x* into *k* bits in the first-level sub-vector v_1 of HCBF.

To improve the false positive rate, HCBF uses a hierarchical structure, where each counter may span over different levels. HCBF is built by using a function $popcount(i)$ that computes the number of ones before position *i* at the hierarchy level that bit *i* belongs to. The value returned by $popcount(i)$ is used as an index to the bit in the next level of the hierarchy containing information relative to position i. Whenever a zero is flipped to one at any position (for example position e) and at any level of the hierarchy (for example j), the indices relative to the bits at the same hierarchy level and with the positions larger than e should be updated. This is done by inserting a zero at position $popcount(e)$ at the next level of the hierarchy and right shifting all bits at the positions larger than $popcount(e)$. This hierarchy updating approach can be used to insert and delete elements in HCBF.

- For an insert operation, we first check each value pointed by $h_i(x)$ in the first level. If the value is 0, we set it to 1 and update the hierarchy using the above-described approach; if the value is 1, we traverse the hierarchy by following the indices given by the *popcount* function, until we reach a value of 0 at position e of hierarchy level j. We flip the value to 1, and insert a 0 at position $popcount(e)$ of the next level (level $j+1$) of the hierarchy, and shift right the bits at the positions larger than $popcount(e)$. The counter value assigned to position $h_i(x)$ is simply j, the depth of the hierarchy attained by the traversal.
- For a delete operation, we do similarly to the insert operation. We first traverse the HCBF hierarchy beginning from position $h_i(x)$ in the first level till we reach the last value of 1 for example at position e in hierarchy level j. We then remove the value of 0 at position $popcount(e)$ at level $j+1$ and shift left the bits at the positions larger than $popcount(e)$ and we flip back the value of 1 at position e into 0.

We show Algorithm 1 that computes the counter values in HCBF as follows:

Algorithm1 Compute Counter Values in HCBF

```
1:  ComputeCounterValues (Element x)
2:  HCBF is composed of d levels $v_1, …, v_d$
3:  Each sub-vector $v_j$ has the size of $b_j$
4:  for (i = 1; i <= k; i++) do
5:      counter[i] = 0;
6:      index = $h_i$(x) mod $b_1$;
7:      for (j = 1; j <= d && HCBF[j][index] == 1; j++) do
8:          counter[i] ++;
9:          index = popcount(HCBF[j][index]);
10:     end for
11: end for
```

Fig. 3(a) depicts an example of HCBF with *k=3* in a word of the length *w=16*, where two elements x_0 and x_5 have been inserted. We assume that the first-level sub-vector v_1 has a size a priori set to 8 bits (such that level 2 of the hierarchy begins at bit 8), and that x_0 is first inserted in HCBF after x_5. When x_0 is inserted, it is hashed to three bits 0, 2, and 4 in v_1. Each of these bits is set to 1, and a shift of level 2 of the hierarchy is done for each new bit set to 1. Therefore, at the end of the insertion we have 2 levels and level 2 accounts for three bits between position 8 and 10. When x_5 is inserted, it is hashed to three bits 7, 4 and 2 in v_1. We flip bit 7 from 0 to 1, and insert a 0 at position 3 (position 8+3=11) of level 2, so that the size of v_2 becomes four bits. We need to traverse HCBF to handle bits $2, 4$. For bit 2, we compute the function $popcount(2) = 1$ and check the bit at position 8+1=9. We flip this bit from 0 to 1, and insert a 0 into level 3 of the hierarchy at position 0 (position 12+0=12). We do similarly for bit 4. Hence, at the end of this stage level 3 contains two bits from position 12 to 13.

The main advantage of HCBF compared to the standard CBF is that the available bits in a word are adaptively assigned to the counters relative to each bit of the first level. This means that a bit that has not been set will not use any bit in the hierarchy, where a larger counter will use more bits. In the standard CBF, all counters have the same number of bits assigned even if they are still 0. As a Bloom filter that is not close to its capacity storage has several zero membership bits, CBF is wasting the storage capacity and HCBF is doing a more sparing usage of this capacity. However the main issue of HCBF is that a word overflow can occur. It happens when there is not anymore space in the hierarchy to add a new element. Intuitively, the word overflow should happen more frequently.

2) *MPCBF-1*

We build a Multiple-Partitioned CBF with one memory access called MPCBF-1, where the membership counter vector is constructed by leveraging the above-described HCBF.

To insert or delete an element x from MPCBF-1, we first use one hash function $H_1(x)$ to hash x to a randomly selected word, which requires $\log_2 l$ hash bits to locate the word. After accessing the memory to locate this single word, we work on the word as a HCBF as described above.

For most modern processors, the *popcount* function is becoming increasingly common and very fast, so its computation time may be neglected. Hence, the overhead of both insertion and deletion in MPCBF-1 is the same, requiring one memory access (to access the word), and an access bandwidth of $\log_2 l + k(\log_2 b_1 + \cdots + \log_2 b_d)$ bits in worst-case[1]. We show that MPCBF-1 has a little larger access bandwidth per insertion or deletion than PCBF-1, due to the traversal of HCBF. Moreover, MPCBF-1 requires only a single memory access per membership query and an access bandwidth of $\log_2 l + k\log_2 b_1$ bits.

We can derive the false positive rate f_{MP-1} of MPCBF-1. Indeed, only the first-level sub-vector v_1 with the size of b_1 bits is used to verify the membership check. Hence, one can easily derive f_{MP-1} based on the false positive rate f_{P-1} of PCBF-1 as follows:

$$\begin{aligned} f_{MP-1} &= \sum_{j=0}^{n}\left(\binom{n}{j}(\frac{1}{l})^{j}(1-\frac{1}{l})^{n-j}\cdot(1-(1-\frac{1}{b_1})^{jk})^{k}) \right) \\ &= \sum_{j=0}^{n}\left(\binom{n}{j}(\frac{1}{4m/w})^{j}(1-\frac{1}{4m/w})^{n-j}\cdot(1-(1-\frac{1}{b_1})^{jk})^{k}) \right) \end{aligned} \quad (4)$$

As expected, f_{MP-1} decreases with the increase of b_1. As shown in Fig. 3(a), some spare bits in the word might remain. There is therefore an opportunity to reduce the false positive rate of MPCBF-1 by increasing the size of the first-level sub-vector in each HCBF by using these spare bits. This is what we propose to do with the improved HCBF.

3) *Improved HCBF*

[1] The size of the i-level sub-vector is $b_i = |v_i|$ bits for $1 \le i \le d$.

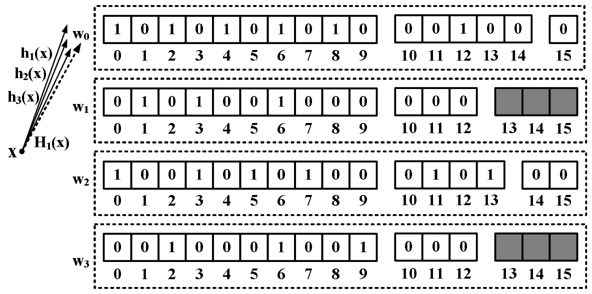


Figure 4. MPCBF-1 with four improved HCBFs.

To achieve a lower false positive rate, we improve the construction of HCBF by maximizing the size of the first-level sub-vector. Let's assume that a word w_i with the size of w bits is used to store at most $n_{\max}$ elements, and a HCBF is composed of d-level sub-vectors $v_1, v_2, \cdots, v_d$ each with the size of b_i bits. This word contains a HCBF and remaining bits, *i.e.*, there is $w = b_1 + \cdots + b_d + b_r$, where b_r denotes the size of remaining bits. As explained in the HCBF construction, each time a hashed value is added into HCBF, there is one bit containing 0 that is inserted in the levels of the hierarchy. This means that if we insert up to $n_{\max}$ elements, we need to have at least $kn_{\max}$ bits available in the levels of the hierarchy:

$$b_2 + \cdots + b_d = k \times n_{\max}$$

The bit size b_1 of the first-level sub-vector v_1 is maximized as:

$$b_{\max} = w - (b_2 + \cdots + b_d) = w - k \times n_{\max}$$

Therefore, the false positive rate f_{MP-1} is minimized as:

$$\begin{aligned} f_{MP-1} &= \sum_{j=0}^{n} \left(\binom{n}{j} \left(\frac{1}{4m/w}\right)^j \left(1-\frac{1}{4m/w}\right)^{n-j} \cdot \left(1-\left(1-\frac{1}{b_{\max}}\right)^{jk}\right)^k \right) \\ &= \sum_{j=0}^{n} \left(\binom{n}{j} \left(\frac{1}{4m/w}\right)^j \left(1-\frac{1}{4m/w}\right)^{n-j} \cdot \left(1-\left(1-\frac{1}{w-k\times n_{\max}}\right)^{jk}\right)^k \right) \end{aligned} \quad (5)$$

Fig. 3(b) depicts an example of the improved HCBF with k=3 and w=16 in a word w_0 and $n_{\max} = 2$. We compute the maximal bit size b_1 of the first-level sub-vector v_1 as $b_{\max} = 16 - 3 \times 2 = 10$. x_0 is hashed to three bits 0, 2 and 4 in v_1, while x_5 is hashed to three bits 4, 6, and 8 in v_1. With this choice five bits are assigned to level 2 and one bit to level 3. Fig. 3(b) shows that the improved HCBF can fill the whole word w_0, and there is no remainder.

Fig. 4 depicts a whole view of MPCBF-1 which is composed of four improved HCBFs. Each HCBF has the same size of the first-level sub-vector. However, the number of bits assigned to other levels of the hierarchy in each word depends on the particular values assigned to each of the membership bits. Fig. 4 shows that words w_0 and w_2 are full, while words w_1 and w_3 can still accept three more membership bits.

Assume that elements are distributed evenly among all the words of MPCBF-1. Each word contains $n_{avg} = n \times w / 4m$ elements, and has the same bit size $b_1 = w - k \times n_{avg}$ of the first-level sub-vector v_1. Therefore, the average false positive rate f_{MP-1}^{avg} of MPCBF-1 is computed as follows:

$$\begin{aligned} f_{MP-1}^{avg} &= \sum_{j=0}^{n} \left(\binom{n}{j} \left(\frac{1}{4m/w}\right)^j \left(1-\frac{1}{4m/w}\right)^{n-j} \cdot \left(1-\left(1-\frac{1}{w-k\times n_{avg}}\right)^{jk}\right)^k \right) \\ &= \sum_{j=0}^{n} \left(\binom{n}{j} \left(\frac{1}{4m/w}\right)^j \left(1-\frac{1}{4m/w}\right)^{n-j} \cdot \left(1-\left(1-\frac{1}{w-k\times n\times w/4m}\right)^{jk}\right)^k \right) \end{aligned}$$

Our analysis shows that MPCBF-1 significantly reduces the false positive rate. This can be seen in Fig. 5 that depicts the false positive rate f of CBF and the average false positive rate f_{MP-1}^{avg} of MPCBF-1 with k=3 and w=16 or 32. Fig. 5 shows that MPCBF-1 has an order of magnitude lower false positive rate than the standard CBF, and increasing the word size can decrease the average rate f_{MP-1}^{avg} of MPCBF-1.

4) *Overflow Analysis*

As explained before, HCBF may have an overflow when the number of elements introduced in a word goes beyond its maximal capacity. We can derive the probability of word overflow in MPCBF-1.

Let E be a random variable representing the number of elements that are hashed to the same word, and j be constant in the range $\{0, \cdots, n\}$. Each element is hashed to a word with equal probability $1/l$. E follows the binomial distribution $B(n, 1/l)$, where n is the number of elements, and l is the number of words in MPCBF-1:

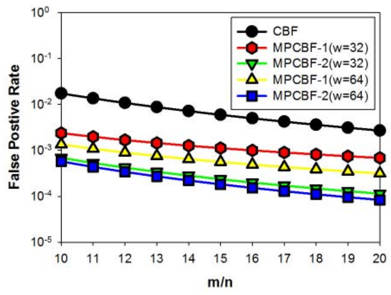


Figure 5. False positive rates of CBF, MPCBF-1 and MPCBF-2 with k=3 and different word sizes

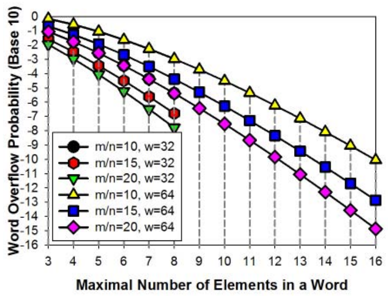


Figure 6. Word overflow probability of MPCBF-1 with n=100000 and k=3.

$$P(E=j)=\binom{n}{j}\left(\frac{1}{l}\right)^{j}\left(1-\frac{1}{l}\right)^{n-j}$$

The upper bound of the probability that any word contains at least $j=n_{\max}$ elements is computed as follows:

$$P(E\geq n_{\max})\leq\binom{n}{n_{\max}}\frac{1}{l^{n_{\max}}}\leq\left(\frac{en}{n_{\max}l}\right)^{n_{\max}} \quad (6)$$

Equation (5) shows that $n_{\max}$ has an important impact on the false positive rate f_{MP-1} of MPCBF-1. By decreasing $n_{\max}$, we can increase the size $b_{\max}$ of the first level of HCBF, which in turn translates to a lower false positive rate f_{MP-1}; however, this also results in a larger overflow probability. There is therefore a fundamental tradeoff between the false positive rate and the word overflow probability: decreasing the false positive rate means increasing the word overflow probability.

To have a better view on this tradeoff, let's fix a memory size for storing a full CBF as M bits, or equivalently $l=M/w$ words. In an enhanced MPCBF-1 each word contains $kn_{\max}$ bits used for storing the HBCF hierarchy, and $w-kn_{\max}$ bits dedicated to the membership vector. Moreover, each word can contain at most $n_{\max}$ elements. Therefore, the efficiency ratio m/n of MPCBF-1 must be larger than:

$$m/n>(w-kn_{\max})/n_{\max}=w/n_{\max}-k \quad (7)$$

This means that knowing the number k of hash functions, the word size w and the parameter $n_{\max}$ and fitting the above lower bound on m/n in (7), one can derive an upper bound of the false positive probability. On other hand Equation (6) provides the overflow probability. By choosing correctly the parameters w, $n_{\max}$, and m/n, one can design MPCBF-1 so that it has a bounded false positive rate as well as an acceptable overflow probability. In practice, we propose a heuristic approach that will be described later in Section 4.B.

Fig. 6 depicts the word overflow probability of MPCBF-1 with n=100000 and k=3 for two choices w=32 and w=64. As can be seen in the figure, choosing w=64 gives more degree of freedom on the choice of $n_{\max}$ and lower word overflow probability that can be achieved.

We also compare the memory usage of MPCBF-1 with that of the standard CBF. Storing n elements in MPCBF-1 needs at least the memory equal to $M=m+kn$ bits, resulting in at least $m/n+k$ bits per element, while the memory is $4m/n$ bits for the standard CBF. However this higher efficiency comes with a cost of an overflow probability.

Equation (7) also gives insight into the possible choice of n as a function of the word size w and the number k of hash functions. Indeed, because of the integer values of w, k, and $n_{\max}$, all values of the efficiency ratio are not possible for MPCBF-1. For example, with w=32 and k=3 only the values of the efficiency ratio higher than $29/3$ are possible.

Even if MPCBF-1 achieves a major improvement over the standard CBF and other previous variants proposed in the literature, however we can further improve its performance by enabling g memory accesses in place of a single one.

C. *MPCBF with g Memory Accesses*

We can generalize MPCBF-1 to MPCBF-*g* that uses *g* hash functions to hash an element to *g* words instead of a single word in MPCBF-1. In MPCBF-*g*, we allocate $\lceil k/g\rceil$ hash functions among *k* hash functions we use for each one of *g* words. As *k* might be not divisible by *g*, we might assign less value to the last word. For example, in MPCBF-2 with *k*=3, we allocate two hash functions to the first word, and one to the second word.

The operation of MPCBF-*g* is similar to that of MPCBF-1: we partition the membership counter vector into an array of *l* words, each word containing a HCBF. We use *g* hash function $H_1,\cdots,H_g$ to derive the words that we have to work on to do insertion or deletion operation. However, we just use at most $\lceil k/g\rceil$ hash functions for each HCBF in the words. The operations on each HCBF are exactly the same as MPCBF-1. Hence, MPCBF-*g* has the same processing overhead for both insertion and deletion operations, but it requires *g* memory accesses with an access bandwidth of $g(\log_2 l+k/g\cdot(\log_2 b_1+\cdots+\log_2 b_d))$ bits in worst-case. A query of MPCBF-*g* also requires *g* memory accesses, and an access bandwidth of $g(\log_2 l+k/g\cdot\log_2 b_1)$ bits.

The false positive rate f_{MP-g} of MPCBF-*g* can be easily derived. Let F' be the false positive event, E' be a random variable for the number of times that a word is selected to store an element, and *j* be constant in the range $\{0,\cdots,gn\}$. Using (3), the false positive rate f_{MP-g} is computed as follows:

$$\begin{aligned}f_{MP-g}&=\left(P(F')\right)^{g}=\left(\sum_{j=0}^{gn}\left(P(E'=j)\cdot P(F'\,|\,E'=j)\right)\right)^{g}\\&=\left(\sum_{j=0}^{gn}\left(\binom{gn}{j}\left(\frac{1}{l}\right)^{j}\left(1-\frac{1}{l}\right)^{gn-j}\cdot\left(1-\left(1-\frac{1}{b_1}\right)^{jk/g}\right)^{k/g}\right)\right)^{g}\end{aligned} \quad (8)$$

where b_1 is the bit size of the first-level sub-vector in HCBF.

Similarly to MPCBF-1 we achieve a lower false positive rate f_{MP-g} of MPCBF-*g*, by maximizing the bit size b_1 and ensuring that no bits in HCBF are spared. For this we observe that the total size of the hierarchy in an improved HCBF must be equal to:

$$b_2+\cdots+b_d=k/g\times n'_{\max}$$

where $n'_{\max}$ is the maximal number of elements selected in each one of *g* words. The bit size b_1 can be maximized as:

$$b_{\max}=w-(b_2+\cdots+b_d)=w-k/g\times n'_{\max}$$

which results into a lower false positive rate f_{MP-g} that is minimized as:

$$\begin{aligned}f_{MP-g}&=\left(\sum_{j=0}^{gn}\left(\binom{gn}{j}\left(\frac{1}{l}\right)^{j}\left(1-\frac{1}{l}\right)^{gn-j}\cdot\left(1-\left(1-\frac{1}{b_{\max}}\right)^{jk/g}\right)^{k/g}\right)\right)^{g}\\&=\left(\sum_{j=0}^{gn}\left(\binom{gn}{j}\left(\frac{1}{l}\right)^{j}\left(1-\frac{1}{l}\right)^{gn-j}\cdot\left(1-\left(1-\frac{1}{w-k/g\times n'_{\max}}\right)^{jk/g}\right)^{k/g}\right)\right)^{g}\end{aligned} \quad (9)$$

When elements are distributed evenly among all the words of MPCBF-*g*, each word contains $n'_{avg} = gn \times w / 4m$ elements each with k / g hash functions, and has the same bit size $b_1 = w - (k / g) \times n'_{avg} = w - k \times n \times w / 4m$ of the first-level sub-vector v_1. Therefore, the average false positive rate f^{avg}_{MP-g} of MPCBF-*g* is computed as follows:

$$f^{avg}_{MP-g} = \left(\sum_{j=0}^{gn} \left(\binom{gn}{j} (\frac{1}{l})^j (1-\frac{1}{l})^{gn-j} \cdot (1-(1-\frac{1}{w-k\times n\times w/4m})^{jk/g})^{k/g} \right) \right)^g$$

We analytically show that MPCBF-*g* ($g \geq 2$) has a lower false positive rate than MPCBF-1. This can be seen in Fig. 5 that depicts the average false positive rates of MPCBF-1 and MPCBF-2 with *k*=3. Increasing the number *g* of memory accesses can decrease the false positive rate of MPCBF-*g*, at the cost of a large access overhead.

Similarly to MPCB-1 we can derive the word overflow probability of MPCBF-*g*. Let E' be a random variable for the number of bits that are hashed to a same word, and j be constant in the range $\{0,\cdots,gn\}$. The probability of $E' = j$ in g words is computed as:

$$\left(P(E'=j)\right)^g = \left(\binom{gn}{j} (\frac{1}{l})^j (1-\frac{1}{l})^{gn-j} \right)^g$$

When $j = n'_{\max}$, the word overflow probability of MPCBF-*g* is computed as follows:

$$\left(P(E' \geq n'_{\max})\right)^g \leq \left(\binom{gn}{n'_{\max}} \frac{1}{l^{n'_{\max}}} \right)^g \leq \left(\frac{en}{n'_{\max}\, l} \right)^{n'_{\max}\, g} \qquad (10)$$

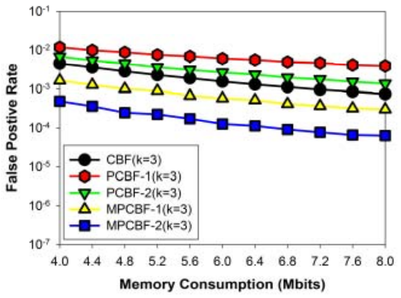
Equation (10) indicates that the word overflow probability of MPCBF-*g* is much smaller than that of MPCBF-1 for most applications.

IV. SIMULATION EXPERIMENTS

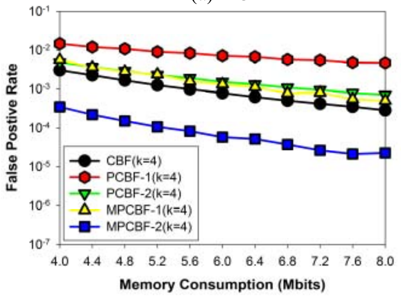
In order to evaluate the performance of MPCBF, we have conducted simulation experiments on synthetic and realistic datasets. The experiments mainly compare MPCBF-*g* with the standard CBF and PCBF-*g*, in terms of the false positive rate and the processing overhead, at the same memory consumption.

A. *Methodology*

For the synthetic experiments, we synthesize a test set and a query set, each containing five-byte strings; each string is randomly generated from the alphabet $\{'a'-'z','A'-'Z'\}$. The test set contains 100K unique strings that are inserted into the filters, while the query set contains 1M strings, of which 80% belongs to the test set, that are tested through the filters. During an update period, 20K strings are deleted from the filters, and another 20K strings are inserted into the filters, maintaining a constant number (100K) of strings in the filters. We generate ten different test sets and query sets, perform the experiments over each one of them, and average the results.



(a) k=3



(b) k=4

Figure 7. False positive rates with k=3 and k=4.

For the realistic experiments, we obtain real-world anonymized Internet traces from CAIDA [24], which have been gathered on the high-speed Internet backbone links of Equinix-Chicago monitor in 2011. The traces contain a total of 5,585,633 IPv4 flows and 292,363 unique IPv4 flows; a flow being defined by the 2-tuple of source IP address and destination IP address. We construct a test set of 200K unique flows randomly selected from the traces, which are inserted in the filters, and feed the total 5,585,633 flows as a query set to perform membership checks. During an update period, 40K flows are deleted from the filters, and then another 40K flows are inserted into the filters, maintaining a constant number (200K) of flows in the filters. This setting simulates a flow measurement system that measures the Internet traffic of 200 K flows in CBF.

B. *Comparisons with k=3 and k=4*

We first conduct the experiments on the synthetic testing set in order to examine the performance of the standard CBF and four variants including PCBF-1, PCBF-2, MPCBF-1, and MPCBF-2. However, we choose a priori the number of hash functions as *k*=3 and *k*=4, and fix the memory used, *i.e.*, above four variants have the same memory consumption as the standard CBF.

Experiments with optimal *k* are done in next sub-section. We also fix the memory usage, and the word size of *w*=64 bits compatible with 64-bit processors. It remains to choose the value of $n_{\max}$ to use for MPCBF-1 and MPCBF-2. For this purpose we use the following heuristic approach:

Let's assume that *g* hash functions $H_1(x),\cdots,H_g(x)$ used for the membership counter vector of MPCBF-*g* are uniform. In this case we expect to have n / l elements mapped to each one of l words. However, as stated before the distribution of the number of elements mapped to a single word follows a binomial distribution $B(n, 1/l)$. As l is generally large (from l=62500 to 250000 in our experiments), it is well known that the binomial distribution converges for large n to a Poisson distribution with the parameter $\lambda = n/l$. Let's define the function $PoissInv(p, \lambda)$ as the inverse CDF of a Poisson distribution, *i.e.*, $PoissInv(p, \lambda)$ returns the value such that the CDF of the Poisson distribution is larger than p. We may set:

$$n_{\max} = PoissInv(1/l, n/l) \qquad (11)$$

This heuristic method may result in choosing $n_{\max}$ from 10 to 7 in our experiments, or equivalently setting $b_1 = 34$ to 43 for k=3, or $b_1 = 24$ to 36 for k=4 in case of w=64. After applying this heuristic method, we never observe any word overflow in our experiments. This means that all reported results are without any word overflow.

Fig. 7 depicts the false positive rates of the standard CBF and four variants. As can be seen, with the increase of the memory consumption the false positive rate decreases almost exponentially. The speed of decrease is higher for MPCBF than PCBF, and also increases with the number g of memory accesses. In particular, it is noteworthy that at the same memory consumption the false positive rate of MPCBF-2 is around 23.2 times less than that of PCBF. Even compared to the standard CBF, both MPCBF-1 and MPCBF-2 have better performance: reduction of the false positive rate with a factor of 13, with less memory access (2 accesses for MPCBF-2 compared to 3 accesses for CBF with k=3) and the same memory usage.

These results are even amplified by using k=4 hash functions. As expected, the false positive rate decreases when k increases from 3 to 4. However, both PCBF and MPCBF do not increase the number of memory accesses. Interestingly, MPCBF-1 has a little larger false positive rate than CBF. This is due to the increase in memory needed for storing the hierarchy in MPCBF-1 with k=4 hash functions that are comparable with the memory space needed for 4-bit counters in CBF. But MPCBF-2 highly outperforms CBF as the false positive rate decreases from 2.9×10^{-4} for CBF with the memory usage of 8 Mb to 1.75×10^{-5} for MPCBF-2. Hence, Fig. 7 demonstrates that MPCBF outperforms both the standard CBF and PCBF in terms of the false positive rate, and its false positive rate decreases with both the decrease of number of hash functions in a word and the increase of number of memory accesses.

We also compare the execution speed of CBF and four variants on the synthesized datasets. Our programs are executed over an Intel Core Processor E6300 (2.8 GHz), 4 GB Main Memory and a single thread. We measure the execution time needed to query 1M elements in CBF and four variants. The results are provided in Fig. 8. The figure shows that the execution time is almost constant and does not depend on the memory consumption. Moreover, the execution time of PCBF-1 and MPCBF-1 is smaller than that of CBF, while the execution time of MPCBF-2 and PCBF-2 are larger than that of CBF. These results are somewhat unexpected as CBF achieves better than MPCBF. Our more complete analysis shows that the major part of software execution time is spent in the hash function calculation that needs to access intensively the memory. We also observe that CBF needs three hash function calculations while both MPCBF-2 and PCBF-2 need four hash function calculations. This explains why the performance of the last two variants is worse than CBF. However, MPCBF-1, PCBF-1 and CBF all have three hash function calculations. The fact that both MPCBF-1 and PCBF-1 take less time than CBF shows the improvement in speed of our approach. Indeed, in a realistic system we expect that the hash function calculations should be done through hardware via FPGA or specific processor instructions, so that their execution time can be much smaller than the time needed to access the memory. This means that in a realistic experiment with hardware support for hashing and no need to further memory access for the hash function calculations, we expect that the performance of MPCBF-2 and PCBF-2 would be higher than that of CBF. We are currently building such a hardware platform.

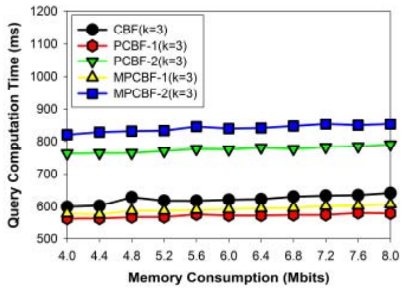


Figure 8. Execution time of 1M queries with k=3.

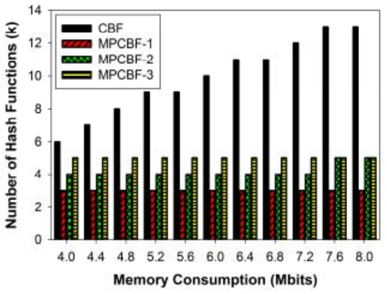


Figure 9. Optimal numbers of hash functions to minimize the false positive rate.

TABLE I. QUERY OVERHEAD WITH K=3 AND K=4

Data Structure	k=3		k=4	
	Num of Memory Accesses	Access Bandwidth (Bits)	Num of Memory Accesses	Access Bandwidth (Bits)
CBF	2.6	55	3.5	72
PCBF-1	1.0	28	1.0	31
PCBF-2	1.8	41	1.8	45
MPCBF-1	1.0	33	1.0	35
MPCBF-2	1.8	46	1.8	51

TABLE II. UPDATE OVERHEAD WITH K=3 AND K=4

Data Structure	k=3		k=4	
	Num of Memory Accesses	Access Bandwidth (Bits)	Num of Memory Accesses	Access Bandwidth (Bits)
CBF	3.0	63	4.0	84
PCBF-1	1.0	29	1.0	33
PCBF-2	2.0	46	2.0	50
MPCBF-1	1.0	36	1.0	40
MPCBF-2	2.0	52	2.0	59

We show in Table 1 and Table 2 a comparison of the query and update overheads for the standard CBF and four variants. The query or update overhead includes the number of memory accesses and the access bandwidth. MPCBF and PCBF have the same number (1.0 for *g*=1 and 1.8 for *g*=2) of memory accesses of query with *k*=3 and *k*=4. However, compared to CBF, MPCBF-1 and MPCBF-2 reduce highly the query and update overheads.

C. Comparisons with Optimal k

In the experiments, we first choose the number of hash functions to optimize the false positive rate. The choice of optimal *k* results in different memory usage for the standard CBF. The previous sub-section shows that the performance of PCBF is always lower than that of CBF. For simplicity of the description, we do not give the results on PCBF and just compare the standard CBF with MPCBF.

We use a word size of *w*=64 bits as in the previous sub-section. CBF uses the optimal number $k=(m/n)\ln 2$ of hash functions. Finding the optimal value of *k* for MPCBF is trickier, as optimizing (8) is difficult. As the values of *k* are discrete, we develop a brute force exhaustive search method for finding the optimal number of hash function in (8).

The results for the optimal values of *k* as a function of the memory consumption are shown in Fig. 9 for CBF and MPCBF in our experiments. As expected, when the memory consumption (proportional to *m*) goes from 4.0 Mb to 8.0 Mb, the optimal number *k* of hash functions increases from 6 to 12; for MPCBF, the optimal value of *k* is almost constant (*k*=3 for MPCBF-1, *k*=4 or *k*=5 for MPCBF-2, and *k*=5 for MPCBF-3).

Fig. 10 shows the false positive rates achieved by CBF and MPCBF when using optimal *k*. The optimal *k* is more favorable to CBF when *k*=3 and *k*=4. As can be seen, for the memory consumption of 8 Mb, the optimal performance of CBF reaches that of MPCBF-2. However, CBF needs almost 12 memory accesses and 12 hash function calculations to

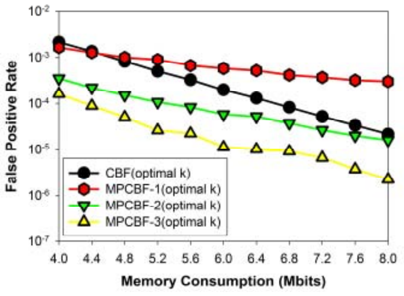
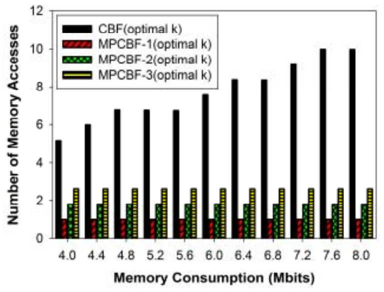
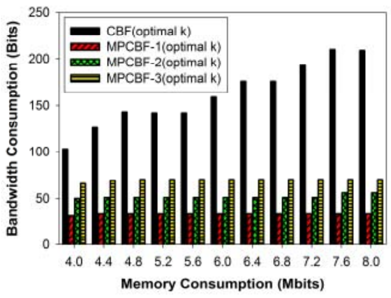


Figure 10. False positive rates with optimal k.



(a) Number of memory accesses



(b) Access bandwidth

Figure 11. Query overhead with optimal k.

achieve this performance, while MPCBF-2 needs only 2 memory accesses and 5 hash function calculations for this performance. Moreover, to show that MPCBF can reach a largely better performance than CBF, we plot the false positive rate (2.2×10^{-6}) of MPCBF-3 with only 3 memory access and *k*=5 hash functions is one order of magnitude less than that (2×10^{-5}) reached by CBF.

Fig. 11 depicts the query overhead of CBF and MCBF when the optimal values of *k* are used. As expected, with increasing the values of *k*, MPCBF-2, and MPCBF-3 have constant 1.0, 1.8, and 2.6 memory accesses per query, respectively, while CBF has the number of memory accesses per query varied from 5.2 to 10. Hence, compared to CBF, MPCBF-1, MPCBF-2, and MPCBF-3 reduce the number of memory accesses per query to 1.0, 1.8 and 2.6, respectively. In the experiments, a similar behavior can be observed on the access bandwidth used, and the update overhead.

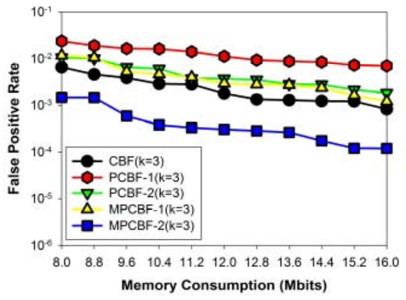


Figure 12. False positive rates with k=3 on IP traces.

TABLE III. PROCESSING OVERHEAD WITH K=3 ON IP TRACES

Data Structure	Query		Update	
	Num of Memory Accesses	Access Bandwidth (Bits)	Num of Memory Accesses	Access Bandwidth (Bits)
CBF	2.1	46	3.0	66
PCBF-1	1.0	26	1.0	30
PCBF-2	1.5	36	2.0	48
MPCBF-1	1.0	28	1.0	36
MPCBF-2	1.5	39	2.0	56

D. Comparisons on Real-World IP Traces

In this sub-section, we present the experimental results on real-world IP traces. We compare the standard CBF and above four variants. Using CBF with optimal *k* incurs a too large processing overhead to be used in practice for a realistic packet processor. We therefore focus on CBF and its variants with *k*=3 in this experiment.

Fig. 12 depicts the false positive rates observed when using CBF, PCBF, and MPCBF for *k*=3. Compared to the synthetic case, the memory used is larger in this experiment. When the memory consumption increases from 8.0 Mb to 16.0 Mb, CBF decreases the false positive rate from 0.66% to 0.083%, while MPCBF-2 decreases the rate from 0.15% to 0.012% (a reduction by a factor of 6.9). MPCBF-1 has a little larger false positive rate than CBF but is very close to it. Hence, MPCBF is demonstrated to achieve higher accuracy than the standard CBF on real-world IP traces.

Table 3 shows the processing overhead obtained over real-world IP traces. As can be seen, MPCBF-1 and MPCBF-2 have separately the same 1.0 and 1.5 memory accesses per query as PCBF-1 and PCBF-2, while CBF has 2.1 memory accesses; the access bandwidth per query of MPCBF-1 and MPCBF-2 is a slightly larger than that of PCBF-1 and PCBF-2, The update overhead of PCBF-1 and MPCBF-2 is separately 1.0 and 2.0 memory accesses, while CBF has 3.0 memory accesses. We therefore demonstrate that MPCBFP achieves faster access than the standard CBF on real-world IP traces.

V. IMPLEMENTATION IN MAPREDUCE

We implement a MPCBF prototype in MapReduce to accelerate the performance of reduce-side join. MapReduce [25] is a programming model for large-scale data processing. The MapReduce model provides *map* and *reduce* functions to allow users to focus only on data processing strategies, hiding the details of parallel execution. Hadoop [26] is an open-source implementation of MapReduce. In the Hadoop, input datasets are split into even-sized records, and each record is then scheduled to a map task node. During the shuffle phase, a map output is fetched by a reduce task node.

TABLE IV. JOIN PERFORMANCE COMPARISON IN MAPREDUCE

Data Structure	m/n=10, k=3, w=64			
	False Positive Rate	Map Output Records	Map Output MegaBytes	Total Execution Time (s)
CBF	0.357	51,309,895	48.93	105
MPCBF-1	0.097	37,631,199	35.89	90
MPCBF-2	0.044	35,777,357	34.12	89

Reduce-side join [27] is the most general join approach implemented in the MapReduce framework, where the join is conducted during the reduce phase. As can be seen in Fig. 13, the *map* function tags a key-value pair $<k, v>$ and produces $<k'+tag, v'+tag>$ as the output; the *reduce* function first separates a list of all values v' associated with each join key k' into two sets according to the *tag*, and then performs a cross-product between values in these sets as the output.

To reduce I/O cost of reduce-side join, a CBF is used in each map task node to filter out irrelevant output records being fetched during shuffling. The smallest of input datasets, i.e. $File1$ in Fig. 13, is often used to construct a CBF that is broadcasted to all map task nodes via *DistributedCache* in Hadoop, avoiding the network overhead for moving the file. We can use MPCBF to replace CBF in this framework for reducing more map outputs, which in turn reduces the join overhead during the reduce phase, improving the total execution time of reduce-side join.

For the reduce-side join evaluation, we use the NBER U.S patent citations data file *cite75_99.txt* [28] as the input dataset (with 16,522,438 records), and a patent dataset as the primary join keys (with 71,661 records on average) from the NBER U.S patents data file *pat63_99.txt* [28]. Our prototype runs on three servers, each with a single 2.8GHz Intel Core 2 Duo CPU and 4GB main memory. We run Hadoop version 0.20.203 on Red Hat Enterprise Linux 6 to perform 10 times experiments, and average the results. Table 4 shows the join performance comparison in MapReduce when using CBF, MPCBF-1, and MPCBF-2. As can be seen, MPCBF reduces the false positive rate from 35.7% (for CBF) to 9.7% (for MPCBF-1) and 4.4% (MPCBF-2). Compared to CBF, MPCBF-1 and MPCBF-2 separately reduce 26.7% and 30.3% of the map outputs, as well as 14.3% and 15.2% of the total execution time due to local data access and no data shuffling. Our implementation and evaluation therefore demonstrate that MPCBF outperforms CBF, accelerating reduce-side joins in MapReduce..

VI. CONCLUSION

We propose a multi-partitioning approach to building a fast and accurate CBF called MPCBF. It combines two ideas of one memory access and a hierarchical structure to improve both the processing overhead and the false positive

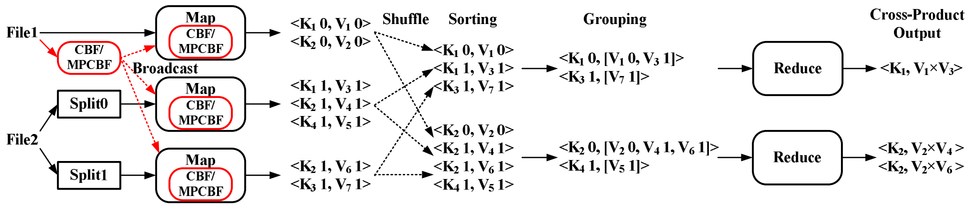


Figure 13. Reside-side join in MapReduce

rate, at the same memory consumption. MPCBF partitions the membership counter vector into an array of continuous words, each of which is further partitioned into multiple levels called HCBF. We present the basic MPCBF with one memory access called MPCBF-1, where each element is hashed to a randomly selected word, and then hashed by *k* hash functions to the first level of HCBF in the work for the membership check. To improve the false positive rate, we propose an improved HCBF to maximize the bit size of the first level. We also generalize MPCBF-1 to MPCBF-*g* that allows *g* memory accesses, improving the false positive rate at the cost of modestly higher processing overhead.

We conduct simulation experiments on synthetic and realistic datasets to evaluate the performance of MPCBF. Results on the synthetic datasets show that for k=3 and k=4, compared to CBF, MPCBF significantly reduces the false positive rate by up to 16.6 times, requires constant one (or a few) memory accesses, and reduces the access bandwidth by up to 52.1%. Results on the real-world IP traces demonstrate that MPCBF significantly outperforms CBF in terms of the false positive rate as well as the access bandwidth. In addition, we implement a MPCBF prototype in MapReduce to accelerate the performance of reduce-side join, showing that MPCBF is a fast and accurate data structure for large-scale data processing.

REFERENCES

[1] B. Bloom, “Space/time tradeoffs in in hash coding with allowable errors,” Communications of the ACM, vol.13, no.7, pp.422-426, 1970.

[2] A. Broder and M. Mitzenmacher, “Network applications of Bloom filters: A survey,” Internet Mathematics, vol.1, no.4, pp.485-509, 2004.

[3] L. Fan, P. Cao, J. Almeida, and A. Broder, “Summary cache: A scalable wide-area web cache sharing protocol,” IEEE/ACM Transactions on Networking, vol.8, no.3, pp.281-293, 2000.

[4] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor, “Longest prefix matching using Bloom filters,” in ACM SIGCOMM, 2003, pp.201-212.

[5] H. Song, F. Hao, M. Kodialam, and T. V. Lakshman, “IPv6 lookups using distributed and load balanced Bloom filters for 100Gbps core router line cards,” in IEEE INFOCOM, 2009, pp.2518-2526.

[6] H. Yu, R. Mahapatra, and L. Bhuyan, “A hash-based scalable IP lookup using Bloom and fingerprint filters,” in IEEE ICNP, 2009, pp.264-273.

[7] A. Kirsch and M. Mitzenmacher, “Simple summaries for hashing with choices,” IEEE/ACM Transactions on Networking, vol.16, no.1, pp.218-231, 2008.

[8] S. Kumar, J. Turner, and P. Crowley, “Peacock hashing: Deterministic and updatable hashing for high performance networking,” in IEEE INFOCOM, 2008, pp.101-105.

[9] H. Yu and R. Mahapatra, “A memory-efficient hashing by multi-predicate Bloom filters for packet classification,” in IEEE INFOCOM, 2008, pp.1795-1803.

[10] K. Huang and D. Zhang, “An index-split Bloom filter for deep packet inspection,” SCIENCE CHINA Information Sciences, vol.54, no.1, pp. 23-37, 2011.

[11] Y. Qiao, T. Li, and S. Chen, “One memory access Bloom filters and their generalization,” in IEEE INFOCOM, 2011, pp.1745-1753.

[12] S. Cohen and Y. Matias, “Spectral Bloom filter,” in ACM SIGMOD, 2003, pp.241-252.

[13] F. Bonomi, M. Mitzenmacher, R. Panigrapy, S. Singh, and G. Varghese, “Beyond Bloom filters: From approximate membership checks to approximate state machines,” in ACM SIGCOMM, 2006, pp.315-326.

[14] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, “The Bloomier filter: An efficient data structure for static support lookup tables,” in SODA, 2004, 30-39.

[15] M.Yu, A. Fabrikant, and J. Rexford, "BUFFALO: Bloom filter for forwarding architecture for large organizations", in ACM CoNEXT, 2009, pp.313-324.

[16] D. Li, H. Cui, Y. Hu, Y. Xia, and X.Wang, "Scalable data center multicast using multi-class Bloom filter," in IEEE ICNP, 2011.

[17] F. Bonomi, M. Mitzenmacher, R. Panigrapy, S. Singh, and G. Varghese, “An improved construction for counting Bloom filters,” in ESA, 2006, pp. 684-695.

[18] N. Hua, H. Zhao, B. Lin, and J. Xu, “Rank-indexed hashing: A compact construction of Bloom filters and variants,” in IEEE ICNP, 2008, pp.73-82.

[19] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci, “Multilayer compresses counting Bloom filters,” in IEEE INFOCOM, 2008, pp.311-315.

[20] S. Lumetta and M. Mitzenmacher, “Using the power of two choices to improve Bloom filters,” Internet Mathematics, vol.4, no.1, pp.17-33, 2009.

[21] F. Hao, M. Kodialm, and T. V. Lakshman, “Building high accuracy Bloom filters using partitioned hashing,” in ACM SIGMETRICS, 2007, pp.277-287.

[22] A. Kirsch and M. Mitzenmacher, “Less hashing same performance: Building a better Bloom filter,” Random Structures and Algorithms, Vol.33, No.1, pp. 187-218, 2008.

[23] O. Rottenstreich, Y. Kanizo, and I. Keslassy, “The variable-increment counting Bloom filter,” in IEEE INFOCOM, 2012.

[24] “CAIDA: The cooperative association for Internet data analysis,” http://www.caida.org.

[25] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in OSDI, 2004.

[26] “Hadoop,” http://hadoop.apache.org.

[27] S. Blanas, J. M. Patel, V. Ercegovac, and J. Rao, “A comparison of join algorithms for log processing in mapreduce,” in ACM SIGMOD, 2010.

[28] “NBER U.S. patent citation data file,” http://data.nber.org/patents/