Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2013, Article ID 516298, 11 pages
http://dx.doi.org/10.1155/2013/516298

Research Article

Accurate Counting Bloom Filters for Large-Scale Data Processing

Wei Li,[1] Kun Huang,[2] Dafang Zhang,[1] and Zheng Qin[1]

[1] *College of Information Science and Engineering, Hunan University, Changsha 410082, China*
[2] *Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China*

Correspondence should be addressed to Dafang Zhang; dfzhang@hnu.edu.cn

Received 3 May 2013; Accepted 5 July 2013

Academic Editor: Gelan Yang

Bloom filters are space-efficient randomized data structures for fast membership queries, allowing false positives. Counting Bloom Filters (CBFs) perform the same operations on dynamic sets that can be updated via insertions and deletions. CBFs have been extensively used in MapReduce to accelerate large-scale data processing on large clusters by reducing the volume of datasets. The false positive probability of CBF should be made as low as possible for filtering out more redundant datasets. In this paper, we propose a multilevel optimization approach to building an Accurate Counting Bloom Filter (ACBF) for reducing the false positive probability. ACBF is constructed by partitioning the counter vector into multiple levels. We propose an optimized ACBF by maximizing the first level size, in order to minimize the false positive probability while maintaining the same functionality as CBF. Simulation results show that the optimized ACBF reduces the false positive probability by up to 98.4% at the same memory consumption compared to CBF. We also implement ACBFs in MapReduce to speed up the reduce-side join. Experiments on realistic datasets show that ACBF reduces the false positive probability by 72.3% as well as the map outputs by 33.9% and improves the join execution times by 20% compared to CBF.

1. Introduction

A Bloom filter [1] is a simple space-efficient randomized data structure for representing a set to support fast membership queries. A Bloom filter uses m bits to represent n elements by using k independent hash functions. Bloom filters allow false positives when querying an element, that is, returning true when the element is not in the set, but not false negatives, that is, returning false when the element is not in the set. Nevertheless, the space savings of Bloom filters often outweigh this drawback when the probability of false positives is made sufficiently low. Due to their simplicity and efficiency, Bloom filters have found widespread applications in databases, network applications [2], and distributed systems [3].

As standard Bloom filters do not support deleting elements, there has been a huge surge in the popularity of Bloom filters and variants. One well-known variant is the Counting Bloom Filter (CBF) [4] which allows the set to change dynamically via insertions and deletions of elements. CBF extends Bloom filters by using a fixed-size counter instead of a single bit for each entry in the vector. When an element is inserted into CBF, the corresponding counters are incremented; deletions can now be safely done by decrementing the counters. To avoid counter overflow, we choose four bits per counter for most applications.

Large-scale data processing has been extensively used in the Cloud. MapReduce [5] is a popular program model for processing vast amounts of data on large clusters of commodity machines. The benefit of MapReduce is to provide *map* and *reduce* functions to users, hiding the details of parallelization, fault tolerance, and load balancing. Hadoop [6] is an open-source implementation of MapReduce. Hadoop is used on massive amounts of structured and unstructured datasets for data analytics applications. Due to its scalability and simplicity, Hadoop has recently been investigated and embraced by many big companies, such as Facebook, Yahoo!, IBM, JP Morgan Chase, New York Times, and China Mobile. So it is critical to study analytic techniques in the MapReduce framework.

However, large-scale data processing poses a significant challenge of performance to MapReduce. First, there are large amounts of data involved with data-intensive applications such as web search engines and log processing. For example, China Mobile has to process 5–8 TB of phone call records per day. Facebook gathers almost 6 TB new log data per day. For these applications, it is time consuming to distribute such data across hundreds or thousands of low-end machines for computations. Second, the join operation is very inefficient in the MapReduce framework. The join is one of fundamental query operations, which combines records from two different datasets based on a cross product. The main problem of the MapReduce join is that two entire datasets should be processed and distributed among a large amount of machines in the clusters. This causes high communication cost and even a performance bottleneck when a small fraction of data is relevant to the join operation. Third, there are not any auxiliary data structures such as indexes and filters in the MapReduce framework. This is due to the fact that MapReduce is initially designed to process only a single large dataset as its input. As all the records within a time period are analyzed together, dataset scans in the MapReduce framework therefore are preferable to index scans.

To address such challenge, CBFs have been widely used to accelerate large-scale data processing in MapReduce. In the reduce-side join [7, 8], CBF is used in the map phase to reduce dramatically the amount of redundant records shuffled across the network, improving the join performance. CBF is built on the smallest dataset in a distributed fashion and distributed via broadcast to all map tasks. To filter out more redundant traffic during the shuffle phase, the false positive probability of CBF should be made as low as possible to process ever-increasing datasets in the MapReduce join.

There are three performance metrics of CBF: processing overhead, memory consumption, and false positive probability. The processing overhead is the number of memory accesses for each primitive operation, which dominates the CBF throughput. The memory consumption is the counter vector size of CBF. Four bits per counter are typically used to support insertions and deletions. As the counters blow up the available memory space, several variants [9–13] have recently been proposed to minimize the memory consumption of CBF, fitting the whole filter in limited high-speed memory (i.e., SRAMs). The false positive probability is to claim an element to be a member in the set, though it is not. There is a tradeoff between the false positive probability and the memory consumption. Decreasing the false positive probability entails increasing the memory consumption. As datasets increasingly growing in size, it is vital to reduce the volume of datasets being processed and distributed across the network in MapReduce. Therefore, our goal is to minimize the false positive probability of CBF while maintaining its memory efficiency.

This paper presents a multilevel optimization approach to building an Accurate Counting Bloom Filter (ACBF). The goal of ACBF is to reduce the false positive probability. ACBF is constructed by partitioning the counter vector into multiple levels that are organized by offset indexing. In ACBF, the first level is used to perform set membership queries, while other levels are used to calculate the counters on insertions and deletions. In order to minimize the false positive probability, we propose an optimized ACBF by maximizing the first level size while maintaining the same functionality as the standard CBF. Simulation results show that ACBF outperforms CBF in false positive probability at the same memory consumption. We also implement ACBFs in MapReduce to improve the reduce-side join performance by filtering out more redundant records shuffled. Experiments in Hadoop show that ACBF reduces the false positive probability by 72.3% as well as the map outputs by 33.9% and improves the total execution times by 20% compared to CBF.

This paper makes the following main contributions.

(1) We propose a novel multilevel optimization approach to building a variant of CBF called ACBF for reducing the false positive probability. ACBF is built by partitioning the counter vector into multiple levels. We propose an optimized ACBF by maximizing the first level size, in order to minimize the false positive probability.

(2) We show that ACBF outperforms CBF in false positive probability at the same memory consumption. Simulation results show that the optimized ACBF reduces the false positive probability by up to 98.4% compared to CBF and performs the same functionality as CBF.

(3) We implement ACBFs in MapReduce to improve the join performance. ACBF is constructed in a distributed fashion and distributed via broadcast to all map tasks for filtering out more redundant records transferred during the shuffle phase.

(4) Experiments on realistic datasets show that ACBF reduces the false positive probability by 72.3% as well as the map outputs by 33.9% and improves the join execution times by 20% compared to CBF.

The rest of this paper is organized as follows. Section 2 introduces the background and related work on Bloom filters and CBFs. We describe the construction and optimization of ACBF and present simulation results in Section 3. Section 4 presents the ACBF implementation in MapReduce and presents experimental results on realistic datasets. Finally, we conclude this paper in Section 5.

2. Background and Related Work

2.1. Bloom Filters and CBFs. Bloom filters are space-efficient randomized data structures to perform approximate membership queries. A Bloom filter represents a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements by using a bit vector of m bits, initially all set to 0. A Bloom filter uses k independent hash functions $h_1, h_2, \ldots, h_k$ with the range $[0, \ldots, m-1]$, each of which hashes an element to a random number uniform over the range. For an element x in S, the bits $h_i(x)$ are set to 1 for $1 \le i \le k$. To query whether an element y is in S, we check whether all bits $h_i(y)$ are set to 1. If not, then clearly y is not in S. If all bits $h_i(y)$ are set to 1, we assume that y is in S with a certain probability.

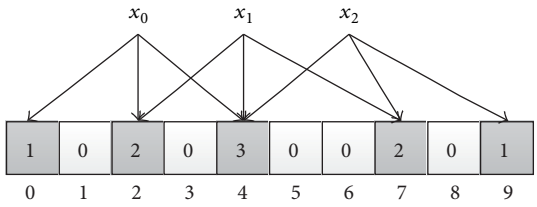


Figure 1: CBF with $m = 10$, $n = 5$, and $k = 3$.

A Bloom filter may yield false positives, but false negatives are not possible. The false positive probability is calculated as follows:

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k \approx \left(1 - e^{-kn/m}\right)^k, \quad (1)$$

where n is the number of elements in the set, m is the size of the bit vector, and k is the number of hash functions. Increasing the number k of hash functions can decrease the false positive probability f for a given ratio m/n. Thus, the false positive probability is minimized as $f \approx (1/2)^k$ when $k = (m/n) \ln 2$. For example, when $m/n = 10$ and $k = 7$, the false positive probability f is just over 0.008.

The standard Bloom filter allows insertion but not deletion. Deleting elements from the Bloom filter cannot be done simply by changing ones back to zeros. This is because a single bit in the vector may correspond to multiple elements inserted. The Counting Bloom Filter (CBF) [4] has been proposed to allow both insertions and deletions of elements. CBF extends the standard Bloom filter by using an array of m counters instead of bits. In CBF, counters are incremented on an insertion and decremented on a deletion. The counters are used to track the number of elements currently hashed to the same entries in the vector. The standard CBF can be derived from CBF by setting all nonzero counters to 1. To avoid counter overflow, four bits per counter have been shown to suffice for most applications. However, due to using counters of four bits, CBF blows up the memory space by a factor of four over the standard Bloom filter, even though most counters are zero.

2.2. Related Work on CBFs. CBFs have been widely used in a variety of applications such as networking [2] and distributed systems [3]. This is due to the fact that CBF is simple and efficient for performing fast membership queries and updates. In order to accommodate different applications, several variants [9–12] have been proposed to improve the performance of CBFs.

CBFs have one of key disadvantages of wasteful fourfold memory space. Several improvements on CBF have recently been proposed to minimize the memory consumption. The d-left CBF (dlCBF) [9, 10] is a simple hash-based alternative based on d-left hashing and fingerprints. dlCBF offers the same functionality as CBF but requires much less memory by a factor of above two at the same false positive probability. The Rank-indexed CBF (RCBF) [11] is a compact alternative based on rank-index hashing. RCBF uses a hierarchical structure for chaining-based fingerprints at each entry, avoiding the costly storage overhead of pointers. RCBF outperforms CBF in memory space by a factor of above three for a false positive probability of 1% and also outperforms dlCBF in memory space by 27% at the same false positive probability. The Multilayer Compressed CBF (ML-CCBF) [12] is also another compact alternative using the idea of a hierarchical structure as well as the Huffman code. ML-CCBF reduces memory space by up to 50% as well as the lookup time compared to CBF. Unlike previous work on the memory consumption, this paper targets the false positive probability. Our goal is to design an accurate variant of CBF, which dramatically reduces the false positive probability while maintaining the same functionality and the memory consumption as CBF.

Moreover, other variants of Bloom filters have recently been proposed to improve the false positive probability. The power of two choices [14] is introduced to reduce the false positive probability by using more hashing. The main idea of this variant is to use two groups of hash functions for inserting elements into the filter and for checking membership queries. An improved variant using partitioned hashing [15] is proposed to improve the false positive probability while avoiding more hashing. This variant works well by partitioning elements into multiple groups and choosing proper combination of hash functions for each group. To reduce hash computations, only two hash functions can be used to derive other hash functions by using a linear combination of the two hash functions [16]. In addition, one memory access Bloom filter [17] is proposed to improve the processing overhead. However, this variant has a larger false positive probability than the standard Bloom filter. Although these previous variants can be generalized to CBF, they suffer from a higher processing overhead due to more hash computations and a larger false positive probability due to wasteful CBF counters. In this paper, ACBF is designed to improve the false positive probability by using multilevel optimization, avoiding these previous limitations.

3. Accurate Counting Bloom Filters

In this section, we describe a multilevel optimization approach to building an Accurate Counting Bloom Filter called ACBF. We first present the construction of ACBF and then describe the query and insertion/deletion algorithms. Next, we describe optimized ACBFs and then analyze the false positive probability. Finally, we show simulation results to compare optimized ACBFs with the standard CBF.

3.1. ACBF Construction. The basic idea of ACBF is to use a multilevel approach to partitioning the counter vector into multiple levels for higher accuracy. Using this approach, we separate the query operation and the insertion/deletion operations of ACBF. This separation is used to achieve a lower false positive probability while attaining updates on dynamic sets. This is done due to the observation of CBF. We observe that the counter vector of CBF is suitable to support quick updates by incrementing or decrementing the counters at the cost of the false positive probability. Figure 1 shows a simple

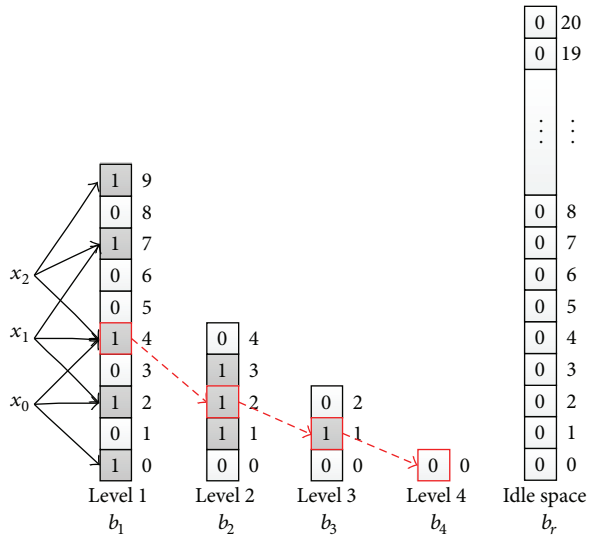


Figure 2: ACBF with four levels and an idle space.

example of CBF with $m = 10$, $n = 5$, and $k = 3$, where m is the number of counters, n is the maximal number of elements, and k is the number of hash functions.

We see that there are only three elements inserted into the filter, and the false positive probability is dominated by the number m of counters for given parameters n and k.

ACBF has a hierarchical structure which is composed of d levels $b_1, b_2, \ldots, b_d$ and an idle space b_r. In this hierarchy, b_1 is used to check the membership for each query, $b_2, \ldots, b_d$ is used to calculate the counters hashed by each element inserted, and b_r is used to remain for elements to be inserted. Figure 2 shows an example of ACBF with four levels b_1, b_2, b_3, and b_4 and an idle space b_r. We see that b_1 has the same size of 9 bits as the counter vector of CBF, and there remain total 21 bits in b_r used for the idle space. Like the standard CBF, ACBF uses k hash functions $h_1, h_2, \ldots, h_k$ to hash an element x into k bits in the first level b_1 instead of the counter vector of CBF. Therefore, we only need to check k bits in b_1 on a membership query. Algorithm 1 shows the query operation in ACBF, where l_1 is the bit size of b_1. If all bits $h_i(x)$ are set to 1, the algorithm claims that element x is in ACBF; otherwise, x is not in ACBF by returning false. Thus, ACBF has the same query complexity $O(k)$ as the standard CBF, denoting average k memory accesses per query.

ACBF is organized by using offset indexing for spanning the counters over different levels. We assume that each level b_i has a size of $l_i = |b_i|$ bits, where i is in the range $[1, \ldots, d]$ and b_r has $l_r = |b_r|$ bits for the idle space. To span a counter, we recursively calculate an offset index in b_i by using a function *popcount* (b_i, j) which computes the number of ones before position j in b_i. The offset value returned by *popcount* (b_i, j) is used as an index to the bit in the next level b_{i+1} contained by the spanned counter. Therefore, we traverse this hierarchy to calculate a counter value by adding up the bits indexed by offset values that the counter responds to. For example, as shown in Figure 2, position 4 in b_1 is hashed by three elements x_0, x_1, and x_3 inserted into ACBF, and its counter spans over four levels b_1, b_2, b_3, and b_4. We traverse these levels to calculate the counter value at position 4. First, as bit 4 in b_1 is set to 1, we call *popcount* $(b_1, 4)$ that returns 2 as an offset index in b_2. Second, we check to see that bit 2 in b_2 is set to 1 and then call *popcount* $(b_2, 2)$ that returns 1 as an offset index in b_3. Third, we continue to call *popcount* $(b_3, 1)$ that returns 0 as an offset index in b_4. Finally, we check to see that bit 0 in b_4 is set to 0 and then terminate the traversal, producing as output the counter value $1 + 1 + 1 + 0 = 3$ at position 4.

In order to insert or delete an element from ACBF, we must increment or decrement the counters hashed by the element. This is done by expanding or shrinking relative levels of the hierarchy. Algorithm 2 shows the insertion operation in ACBF. When an element is inserted into ACBF, we need to perform k lookups by traversing a series of levels $b_1, \ldots, b_j$ for each counter hashed by the element (see Lines from 3 to 9 in Algorithm 2). For incrementing a counter value, we expand the next level b_{i+1} by adding a one bit in b_j and shifting upward all bits of b_{i+1} by one position (see Lines from 11 to 16 in Algorithm 2). Like insertion, deletion also requires a lookup and a shift for each of k hashed counters. Algorithm 3 shows the deletion operation in ACBF. When an element is deleted from ACBF, we perform the same lookups by traversing a series of levels $b_1, \ldots, b_j$ for each counter hashed by the element (see Lines from 3 to 10 in Algorithm 3) and then shrink the last level b_j by shifting backward all the bits of b_j, at the same time of removing a one bit from b_{j-1} (see Lines from 12 to 14 in Algorithm 3). For example, we assume that element x_3 is deleted from ACBF as shown in Figure 2. As position 4 in b_1 is hashed by x_3, we traverse level b_1 to level b_4 for its counter value. In order to decrement the counter value, we shrink b_4 by removing a zero bit at position 0 and shrink b_3 by removing a one bit at position 1.

From Algorithms 2 and 3, we see that both insertion and deletion have almost the same time complexity. Let φ be a counter value and λ be a popcount cost. Thus, the insertion/deletion complexity of ACBF is calculated as follows:

$$O\left(k \times \left[E\left(\varphi\right) \times \left(E\left(\lambda\right) + 1\right) + 1 + 1\right]\right), \tag{2}$$

where $E(\varphi)$ is the average counter value and $E(\lambda)$ is the average *popcount* cost. $E(\lambda) + 1$ denotes a *popcount* function and a read operation, and $1 + 1$ denotes a write operation and a shift for updating each counter. Equation (2) shows that ACBF has more complexity for the insertion/deletion than CBF with $O(k)$. A recent study [12] has shown a tight approximation $E(\varphi) \simeq \ln 2$ when the false positive probability is minimized in ACBF. Thus, the insertion/deletion complexity depends on the average *popcount* cost. Fortunately, the *popcount* function is becoming increasingly common and very fast (e.g., one CPU cycle on a word) for most modern processors. Hence, $E(\lambda)$ is dominated by the word lengths of different levels in the ACBF hierarchy.

3.2. ACBF Optimization. To attain a lower false positive probability, we propose two optimization methods to improve the ACBF construction by increasing the first level size. We assume that ACBF consists of d levels $b_1, b_2, \ldots, b_d$ and an idle

```
(1) Query (Element x)
# ACBF is composed of d levels b_1, ..., b_d and an idle space b_r
# Each bitmap b_j has the size of l_j
(2) for (i = 1; i <= k; i++) do
(3)     index = h_i(x) mod l_1;
(4)     if (b_1[index] == 0) then
(5)         return false;
(6)     end if
(7) end for
(8) return true;
```

Algorithm 1: Query operation in ACBF.

```
(1) Insert (Element x)
(2) for (i = 1; i <= k; i++) do
# Traverse d bitmaps b_1, ..., b_d
(3)    for (j = 1; j <= d; j++) do
(4)        if (j == 1) then
(5)            index = h_i(x) mod l_1;
(6)        end if
(7)        if (b_j[index] == 1) then
(8)            offset = popcount(b_j, index);
(9)            index = offset;
# Expand bitmap b_{j+1} by one position
(10)           else
(11)             b_j[index] = 1;
(12)             offset = popcount(b_j, index);
(13)             expand_bitmap(b_{j+1}, offset);
(14)             Index = offset;
(15)             b_{j+1}[index] = 0;
(16)             exit();
(17)           end if
(18)       end for
(19)   end for
```

Algorithm 2: Insertion operation in ACBF.

space b_r, each with a bit size $l_i = |b_i|$ for $1 \le i \le d$ or $i = r$. Thus, the false positive probability f is calculated as follows:

$$f = \left(1 - \left(1 - \frac{1}{l_1}\right)^{nk}\right)^k \approx \left(1 - e^{-kn/l_1}\right)^k, \tag{3}$$

where n is the maximal number of elements, l_1 is the bit size of the first level, and k is the number of hash functions. Equation (3) shows that the false positive probability f is dominated by the first level size l_1 for given parameters n and k.

The basic idea of the first optimization method is to simply increase the first level size by a multiplicative factor. Let c be an integer value, and let m be the number of CBF counters. Using this method, we set the first level size as $l_1 = cm$ for $1 \le c \le 4$ and improve the false positive probability f_c as follows:

$$f_c = \left(1 - \left(1 - \frac{1}{cm}\right)^{nk}\right)^k \approx \left(1 - e^{-kn/cm}\right)^k. \tag{4}$$

If $l_1 = m$, ACBF has the same false positive probability f_1 as CBF. If $l_1 = 4m$, ACBF has the lowest false positive probability f_4 but cannot allow deletions of elements. Hence, the key of optimizing ACBF is to set the optimal size of the first level for minimizing the false positive probability while maintaining the same functionality of insertions and deletions as CBF.

To achieve this goal, we propose the second optimization method for improving the ACBF construction. This method is deigned based on the following observation: if up to n elements are inserted in ACBF, we need to have at least kn bits available for $j \ge 2$ levels in the hierarchy:

$$l_2 + \cdots + l_d = kn. \tag{5}$$

The bit size l_1 of the first level b_1 is maximized as follows:

$$l_1 = 4m - (l_2 + \cdots + l_d) = 4m - kn. \tag{6}$$

```
(1) Delete (Element x)
(2) for (i = 1; i <= k; i++) do
# Traverse d bitmaps b_1, ..., b_d
(3)   for (j = 1; j <= d; j++) do
(4)     if (j == 1) then
(5)       index = h_i(x) mod l_1;
(6)     end if
(7)     if (b_j[index] == 1) then
(8)         offset = popcount(b_j, index);
(9)         pre = index;
(10)        index = offset;
(11)      else
# Shrink bitmap b_j by one position
(12)        shrink_bitmap(b_j, offset);
(13)      b_{j-1}[pre] = 0;
(14)      exit();
(15)    end if
(16)  end for
(17) end for
```

Algorithm 3: Deletion Operation in ACBF.

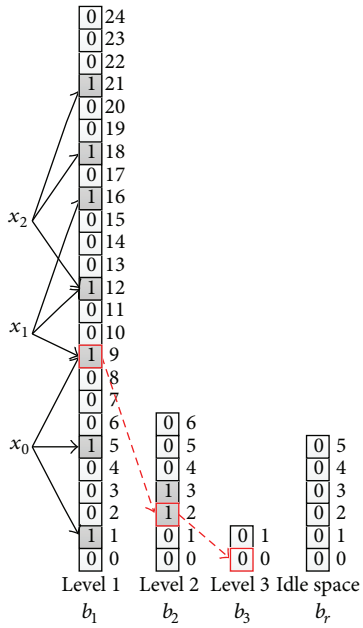


Figure 3: Optimized ACBF with $m = 10$, $n = 5$, and $k = 3$.

Thus, the optimal false positive probability f_o is calculated as follows:

$$f_o = \left(1 - \left(1 - \frac{1}{4m - kn}\right)^{nk}\right)^k \approx \left(1 - e^{-kn/(4m-kn)}\right)^k. \quad (7)$$

When $k = (m/n)\ln 2 < m/n$, the upper and lower bounds of the first level size l_1 are derived as follows:

$$3m < l_1 = 4m - kn < 4m. \quad (8)$$

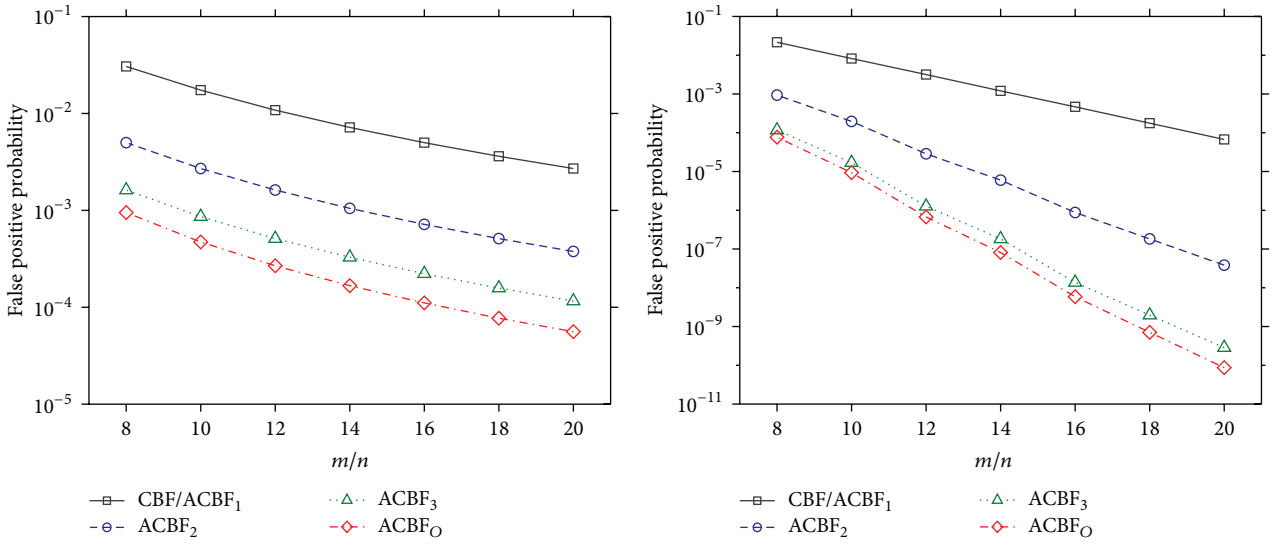
Hence, we can derive the upper and lower bounds of the optimal false positive probability f_o as

$$f_4 < f_o < f_3. \quad (9)$$

Equation (9) shows that the second method has a lower false positive probability than the first method with $l_1 = 3m$. Using the second method, ACBF remains kn bits for performing the same functionality of insertions and deletions as the standard CBF, avoiding space overflow.

Figure 3 shows an example of an optimized ACBF with $m = 10$, $n = 5$, and $k = 3$, where m is the number of 4-bit counters. There are only three elements x_0, x_1, and x_2 inserted in the filter. Using (5), we need at most $kn = 15$ bits for $j \geq 2$ levels of the hierarchy. Thus, we set the optimal bit size $l_1 = 4m - kn = 25$ of the first level b_1 and remain the idle space b_r of 6 bits for allowing other two elements inserted in the filter. We also see that the bit size l_2 of the second level b_2 is set to the number of ones in the first level b_1, and the bit size l_3 of the third level b_3 is set to the number of ones in the second level b_2. For example, we assume that position 9 in b_1 is hashed by two elements x_0 and x_1 (see Figure 3). We span the counter value at position 9 over b_1 and b_2 by using offset indexing described in Section 3.1.

Next, we compare the theoretical false positive probability of ACBFs with CBF in two cases of $k = 3$ and $k = optimal$ as shown in Figure 4. Note that $k = optimal$ means the optimal number of hash functions $k = (m/n)\ln 2$ for minimizing the false positive probability. Figure 4 shows that ACBF_O with the optimal first level size $l_1 = optimal$ outperforms both CBF and ACBF_i with $l_1 = im$ for $1 \leq i \leq 3$ by using the first optimization method. For instance, Figure 4(a) shows that ACBF_O reduces the false positive probability by up to 97.6%, up to 83.7%, and up to 48.1% in case of $k = 3$ compared to CBF/ACBF_1, ACBF_2, and ACBF_3, respectively. We note that ACBF_1 has the same false positive probability as CBF due to



(a) $k = 3$ (b) k = optimal

Figure 4: Theoretical false positive probability of CBF and ACBFs.

$l_1 = m$. Figure 4(b) also shows that in case of $k = optimal$, $ACBF_O$ reduces the false positive probability by up to five orders of magnitude compared to CBF/$ACBF_1$ as well as the false positive probability by up to 99.8% and up to 69.6% compared to $ACBF_2$ and $ACBF_3$.

3.3. Simulation Results. We conduct simulation experiments to test the performance of ACBFs on synthetic datasets. As a standard CBF has the same memory consumption as its previous variants, we mainly compare ACBFs with the standard in the experiments. We compare $ACBF_O$ with CBF, $ACBF_1$, $ACBF_2$, and $ACBF_3$ in terms of the false positive probability, the query overhead, and the update overhead, at the same memory consumption. In the experiments, both CBF and ACBFs have the same parameters such as m, n, and k, where m/n varies from 8 to 20 and k is set to 3 or the optimal integer value $\lceil (m/n) \ln 2 \rceil$.

For each synthetic experiment, we synthesize a data set and a query set. The data set contains 100 K unique strings that we represent with CBF and ACBFs, while the query set contains 1000 K strings that are tested through the filters. During an update period, 20 K elements are deleted from the filters, and another 20 K elements are inserted into the filters, maintaining constant 100 K elements in the filters. We do ten experimental trials and average the results.

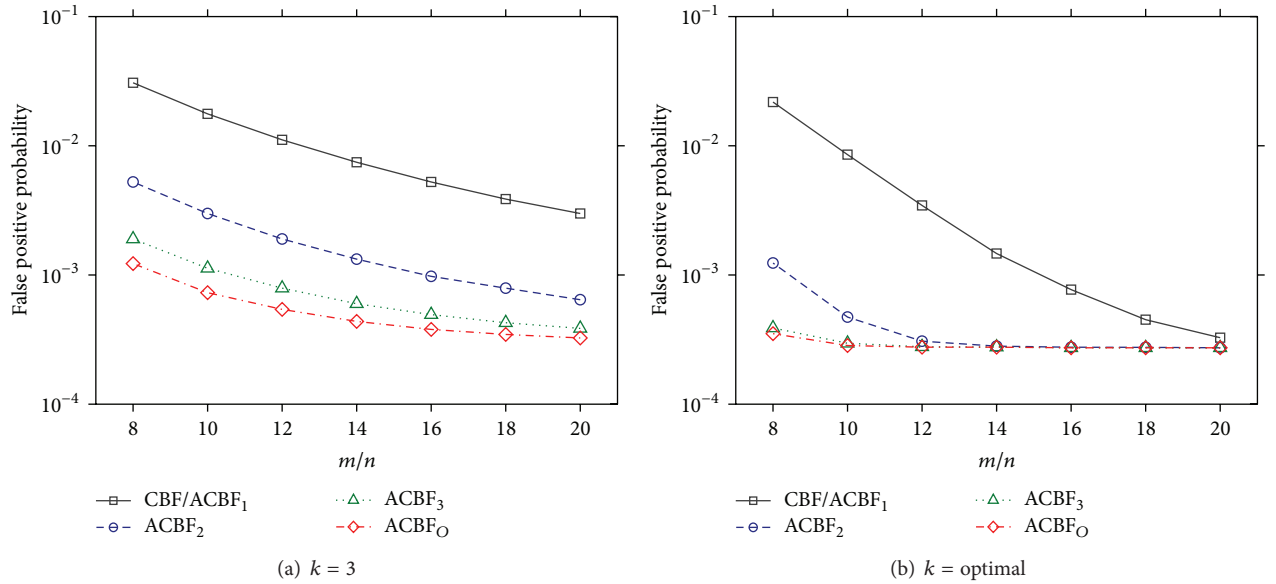
Figure 5 depicts the false positive probability of CBF, $ACBF_1$, $ACBF_2$, $ACBF_3$, and $ACBF_O$. From the figure, we demonstrate that $ACBF_O$ achieves significant reductions in the false positive probability compared to CBF, $ACBF_1$, $ACBF_2$, and $ACBF_3$. As shown in Figure 5(a), $ACBF_O$ with $k = 3$ reduces the false positive probability by up to 96.0%, up to 76.6%, and up to 35.5%, compared to CBF/$ACBF_1$, $ACBF_2$, and $ACBF_3$, respectively. Figure 5(b) shows that compared to CBF/$ACBF_1$, $ACBF_O$ with $k = optimal$ reduces the false positive probability by up to 98.4%. We also see that when m/n is from 16 to 20 in case of $k = optimal$, $ACBF_3$ has the same false positive probability as $ACBF_O$. This is because the first level size l_1 of $ACBF_O$ is very close to $3m$.

Figure 6 depicts the query overhead of CBF, $ACBF_1$, $ACBF_2$, $ACBF_3$, and $ACBF_O$ in two cases of $k = 3$ and $k = optimal$. We see that $ACBF_O$ has almost the same query times as CBF, $ACBF_1$, $ACBF_2$, and $ACBF_3$. This reason is that all the filters have the same number k of hash functions. We do examine the query overhead of CBF and ACBFs on a server with 2.4 GHz Intel Core Duo CPU P8600 and 3 GB main memory. Figure 6 shows that the query overhead of CBF and ACBFs is dominated by the number of hash functions. For instance, all the filters have about 2.4 us per query in case of $k = 3$ (see Figure 6(a)), while they require 2.7 3.6 us per query in case of $k = optimal$ (see Figure 6(b)).

Figure 7 depicts the update overhead of CBF, $ACBF_1$, $ACBF_2$, $ACBF_3$, and $ACBF_O$ in two cases of $k = 3$ and $k = optimal$. We see that ACBFs require more update overhead than CBF. This is because ACBFs use the hierarchical structure to span the counters, which incurs more computations and I/O overhead. Figure 7 shows that ACBFs have more three orders of magnitude update times compared to CBF. For instance, CBF with $k = 3$ in Figure 7(a) requires 2.6 3.2 us per update, while ACBFs requires 1.4 4.2 ms per update. From the figure, we demonstrate that ACBFs dramatically reduce the false positive probability at the cost of the update overhead as analyzed in (2) of Section 3.1.

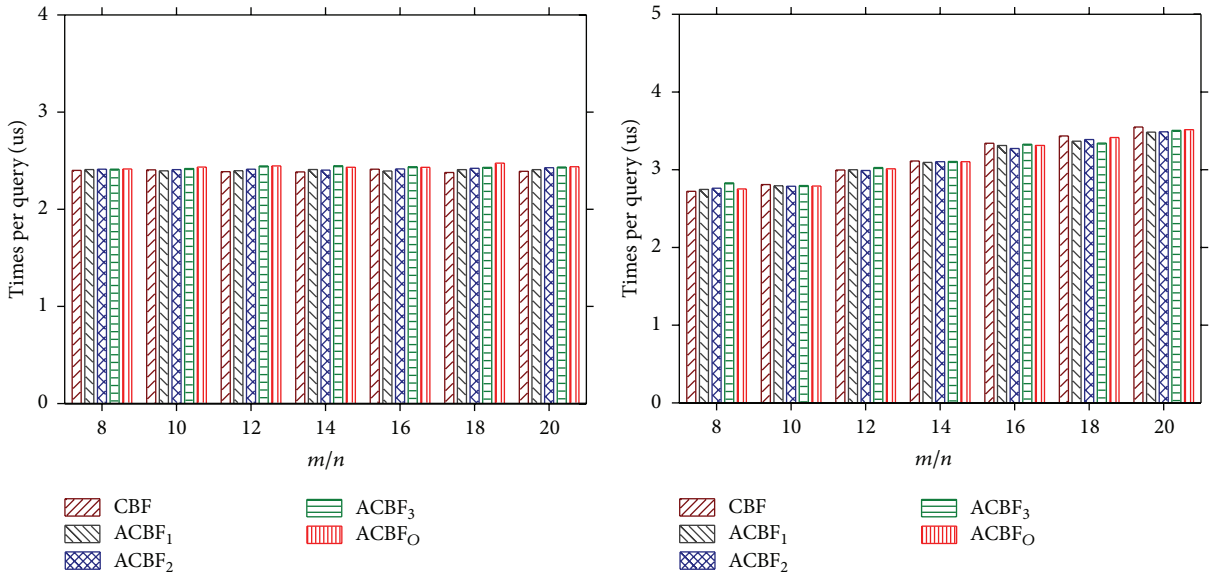
4. Implementation in MapReduce

In this section, we implement ACBFs in MapReduce to accelerate reduce-side joins for large-scale data processing. We first present the MapReduce overview and then describe the optimized reduce-side join with ACBF in MapReduce. Finally, we report experimental results on realistic datasets.



(a) $k = 3$ (b) k = optimal

Figure 5: False positive probability of CBF, ACBF$_1$, ACBF$_2$, ACBF$_3$, and ACBF$_O$.



(a) $k = 3$ (b) k = optimal

Figure 6: Query overhead of CBF, ACBF$_1$, ACBF$_2$, ACBF$_3$, and ACBF$_O$.

4.1. MapReduce Overview. MapReduce [5] is a programming model for large-scale data processing on large clusters of commodity machines. The MapReduce model offers automatic parallel execution to allow users to only focus on their data processing strategies, hiding the details of parallel and distributed executions.

A MapReduce program provides *map* and *reduce* functions to users. The *map* function takes a set of key-value pairs (K, V) from input files as the input and produces intermediate key-value pairs (K', V') as the output. The *reduce* function takes pairs $(K', LIST\ (V'))$ as the input, where $LIST\ (V')$ is a list of all values V' that are grouped on a given key K' by sorting and merging. Yet, the *reduce* function produces the final output key-value pairs. Typically, both the input and the output of a MapReduce job are files in a distributed file system (DFS), that is, Google File System (GFS). DFS is a block-based file system that supports fault tolerance by data partitioning and replication.

When a MapReduce job is launched, a job tracker creates a total of m map tasks and r reduce tasks. Each map task works on a nonoverlapping data block called a split of the input file. Each map task $Map_i\ (i = 1, \ldots, m)$ reads the file split, converts the records into a sequence of key-value pairs, and then executes the user-defined *map* function with each

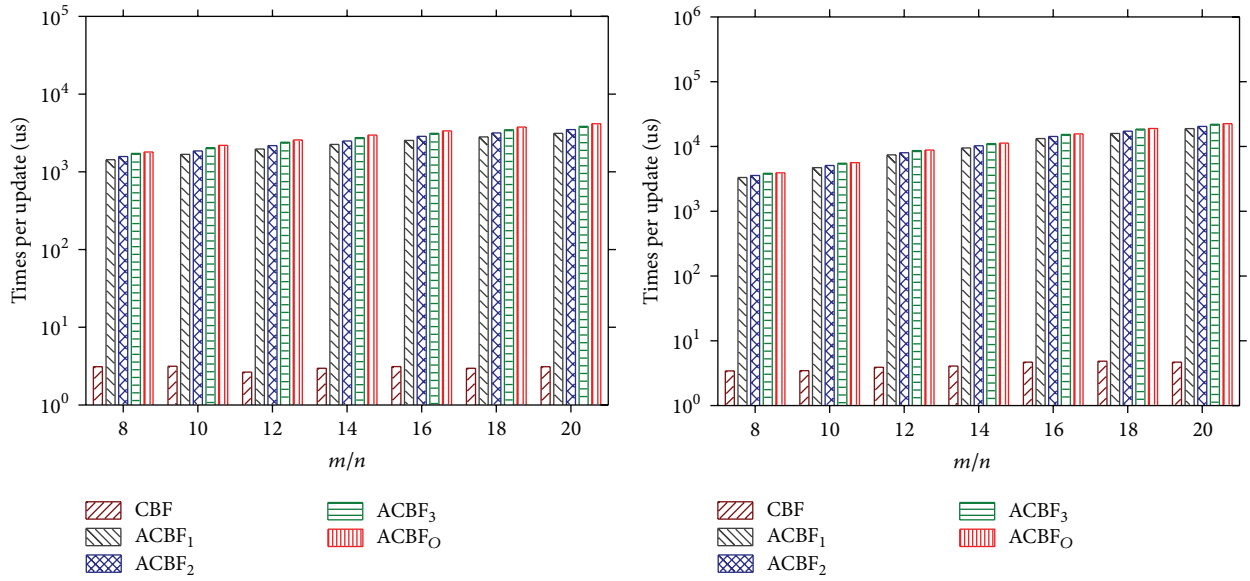


Figure 7: Update overhead of CBF, $ACBF_1$, $ACBF_2$, $ACBF_3$, and $ACBF_O$.

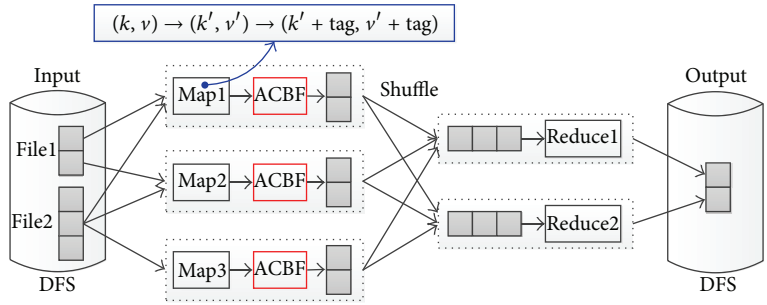


Figure 8: Reduce-side join with ACBF in MapReduce.

pair (K, V). The output pairs (K', V') are partitioned into r chunks, one for each reduce task. This partitioning is basically done by a hash function to guarantee that pairs with the same key are allocated to the same chunk. The pairs in each chunk are sorted by K' and written to local storage. Each reduce task $Reduce_j$ $(j = 1, \ldots, r)$ fetches the jth chunk outputted by each map task remotely. This is referred to the shuffle phase in MapReduce, where each chunk is assigned to only a single reduce task by one-to-one shuffling strategy. $Reduce_j$ merges these chunks, produces the intermediate pairs $(K', LIST\ (V'))$ sorted by K', and then invokes the user-defined *reduce* function for each pair $(K', LIST\ (V'))$. The final output is stored and tripled in DFS before the MapReduce job terminates.

4.2. Reduce-Side Join with ACBF. The join operation is one of fundamental query operations. It combines records from two different datasets based on a cross product [18]. We consider a join between two datasets R and S on attribute A of R and attribute B of S. The join result contains the pairs of records from R and S when $R \cdot A = S \cdot B$.

There are two main join implementations in MapReduce: the map-side join and the reduce-side join. As their own names imply, the map-side join implements the join during the map phase, while the reduce-side join implements the join during the reduce phase [19]. The map-side join is more efficient than the reduce-side join because it produces the final results of the join in the map phase. However, the map-side join is used only in particular circumstances, lacking the generality of the reduce-side join. This is because its efficiency requires the two input datasets to be partitioned and sorted on the join keys in advance.

The reduce-side join is the most general join approach implemented in MapReduce. The basic idea behind the reduce-side join is that a map task tags each key-value pair with its source and uses the join keys as the map output keys, so that the pairs with the same key are grouped for a reduce task. Figure 8 shows the tagging in the map phase. The map task reads a key-value pair (K, V) and produces a tagged pair $(K' + tag, V' + tag)$ as the output. Next, the map outputs $(K' + tag, V' + tag)$ with the same K' are sent to the same reduce task. The pairs $(K' + tag, V' + tag)$ are sorted primarily

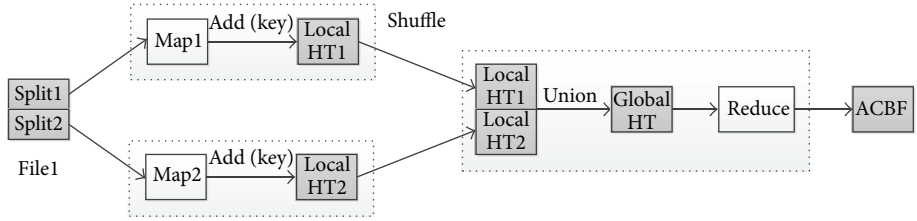


Figure 9: ACBF construction in MapReduce.

Table 1: Reduce-side join performance comparisons in Hadoop.

Filter parameters $m/n = 10$, $k = 3$	False positive probability	Map inputs (MB)	Map outputs (MB)	Filter construction times (s)	Total execution times (s)
Join + CBF	0.01772	252.5	45.1	62	115
Join + $ACBF_O$	0.00491	252.5	29.8	68	92

on K' and secondarily on *tag* and grouped only on K'. Finally, the *reduce* function separates a list of all values V' associated with the same join key K' into two sets according to *tag* and performs a cross product between values in the two sets as the final output.

To mitigate I/O cost of the reduce-side join, CBF is widely used in the map phase to filter the map outputs shuffled across the network. We use ACBF to replace CBF in the reduce-side join for minimizing the amount of traffic during the shuffle phase. ACBF has so lower false positive probability that more redundant map outputs can be reduced. Figure 8 shows an example of the reduce-side join with ACBF in MapReduce. The smallest input file, that is, $File_1$, is often used to construct an ACBF in a distributed way. The job tracker then broadcasts the ACBF to all the map tasks, that is, Map_1, Map_2, and Map_3, by an efficient facility called *DistributedCache* which is provided by the MapReduce framework to cache data needed by the applications. Each map task uses the ACBF to filter out the redundant records. After that all the reduce tasks, that is, $Reduce_1$ and $Reduce_2$, perform the joins and produces the final results.

Figure 9 shows the ACBF construction in MapReduce. First, a job tracker splits the input file, that is, $File_1$, and then each map task (i.e., Map_1 or Map_2) reads an input split, that is, $Split_1$ or $Split_2$. A local hash table is created in each map task by adding the unique keys of each file split. Note that each map task consists of the *map* function only, without additional I/O cost. Each local filter (i.e., HT_1 or HT_2) is a chained hash table which has the same number of buckets and uses the same hash function called *MurmurHash* implemented in MapReduce. When each map task is completed, all local hash tables are assigned to one reduce task (i.e., $Reduce_1$), which creates a global hash table. All the local hash tables are merged by a union function that eliminates duplicated keys. Next, the reduce task creates an ACBF by adding the keys of the global hash table in the filter. Finally, the ACBF is written to local storage in DFS and submitted via broadcast to all other map tasks for performing the reduce-side join (see Figure 8).

4.3. Experimental Results. To evaluate the optimized reduce-side join, we implement ACBFs in Hadoop that is an open-source Java implementation of MapReduce. We obtain the NBER US patent citations data files [20] for evaluation. We use the patent citations file named *cite75_99.txt* of 16,522,438 records as the input dataset. We extract 65,771 records from the patent data file named *pat63_99.txt* as the join keys. Our Hadoop prototype runs on three servers each with 2.8 GHz Intel Core 2 Duo CPU and 4 GB main memory. We run Hadoop version 0.20.203 on Red Hat Enterprise Linux 6, perform ten experimental trials, and average the results.

Table 1 shows reduce-side join performance comparisons in Hadoop. We compare the join with CBF to that with the optimized ACBF named $ACBF_O$. From the table, we see that $ACBF_O$ dramatically reduces the false positive probability by 72.3% as well as the map outputs by 33.9% compared to CBF. We also see that the reduce-side join with $ACBF_O$ requires less 20% total execution times than that with CBF. This is because $ACBF_O$ filters out more redundant records in the map phase. In addition, $ACBF_O$ requires only more 9.7% construction times than CBF. This is due to the hierarchical structure of $ACBF_O$.

5. Conclusions

We propose a multilevel optimization approach to building an accurate CBF called ACBF for reducing the false positive probability. ACBF is constructed by partitioning the counter vector into multiple levels. We propose an optimized ACBF named $ACBF_O$ which maximize the first level size as $4m\text{-}kn$, minimizing the false positive probability while maintaining the same functionality as CBF, where m is the number of counters, n is the number of elements, and k is the number of hash functions. Simulation results show that $ACBF_O$ reduces the false positive probability by up to 96.0% in case of $k = 3$ and by up to 98.4% in case of $k = optimal$ compared to CBF.

We implement ACBFs in MapReduce to improve the reduce-side join performance. ACBF is used in the map phase to filter out redundant records shuffled. ACBF is constructed in a distributed way by merging local hash tables of all map

tasks. Experiments on realistic patent citations data files show that $ACBF_O$ reduces the false positive probability by 72.3% as well as the map outputs by 33.9% and improves the join execution times by 20% compared to CBF. We show that ACBF is an accurate data structure suitable for large-scale data processing.

Acknowledgments

This work was supported in part by the National Basic Research Program (973 Program) of China under Grant no. 2012CB315805, the National Natural Science Foundation of China under Grant no. 61173167, no. 61100171, and no. 61272546, and the Scientific and Technological Project of Hunan Province, China, under Grant no. 2013SK3149.

References

[1] B. B. Bloom, "Space/time tradeoffs 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] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *IEEE Communications Surveys and Tutorials*, vol. 14, no. 1, pp. 131–155, 2012.

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

[5] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI '04)*, pp. 137–149, San Francisco, Calif, USA, 2004.

[6] Hadoop [EB/OL], 2012, http://hadoop.apache.org .

[7] C. Lam, *Hadoop in Action*, Manning Publications Press, Shelter Island, NY, USA, 2010.

[8] T. Lee, K. Kim, and H. Kim, "Join processing using Bloom filter in MapReduce," in *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, pp. 100–105, San Antonio, Tex, USA, 2012.

[9] F. Bonomi, M. Mitzenmacher, R. Panigrapy, S. Singh, and G. Varghese, "Beyond Bloom filters: from approximate membership checks to approximate state machines," in *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '06)*, pp. 315–326, Pisa, Italy, October 2006.

[10] F. Bonomi, M. Mitzenmacher, R. Panigrapy, S. Singh, and G. Varghese, "An improved construction for counting Bloom filters," in *Proceedings of the 14th Annual European Symposium on Algorithms (ESA '06)*, pp. 684–695, Zurich, Switzerland, September 2006.

[11] N. Hua, H. Zhao, B. Lin, and J. Xu, "Rank-indexed hashing: a compact construction of bloom filters and variants," in *Proceedings of the 16th IEEE International Conference on Network Protocols (ICNP '08)*, pp. 73–82, Orlando, Fla, USA, October 2008.

[12] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci, "MultiLayer compressed counting bloom filters," in *Proceedings of the 27th IEEE Communications Society Conference on Computer Communications (INFOCOM '08)*, pp. 311–315, Phoenix, Ariz, USA, April 2008.

[13] O. Rottenstreich, Y. Kanizo, and I. Keslassy, "The variable-increment counting Bloom filter," in *Proceedings of the 31th IEEE International Conference on Computer Communications (INFOCOM '12)*, pp. 1880–1888, Orlando, Fla, USA, 2012.

[14] 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.

[15] F. Hao, M. Kodialm, and T. V. Lakshman, "Building high accuracy Bloom filters using partitioned hashing," in *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, pp. 277–287, San Diego, Calif, USA, 2007.

[16] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: building a better bloom filter," *Random Structures and Algorithms*, vol. 33, no. 2, pp. 187–218, 2008.

[17] Y. Qiao, T. Li, and S. Chen, "One memory access bloom filters and their generalization," in *Proceedings of the 30th IEEE International Conference on Computer Communications (INFOCOM '11)*, pp. 1745–1753, Shanghai, China, April 2011.

[18] F. N. Afrati and J. D. Ullman, "Optimizing multiway joins in a map-reduce environment," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 9, pp. 1282–1298, 2011.

[19] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in MaPreduce," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD '10)*, pp. 975–986, Indianapolis, Ind, USA, June 2010.

[20] NBER U.S. patent citation data file [EB/OL], 2012, http://data.nber.org/patents.