Journal of Engineering Sciences

Vol 13, Issue 06, June/2022

ISSN NO:0377-9254

ACCURATE COUNTING BLOOM FILTERS FOR LARGE-SCALE DATA PROCESSING

N Mahema Reddy [1], Anvita Veeramallu [2], Nikhil Chagi [3]

#1,#2,#3, Student, Department of CSE, GITAM (deemed to be University), Rudraram, Hyderabad, Telangana, INDIA

Abstract: Bloom filters are space-efficient randomized data structures that allow for false positives in query searches. On the given dynamic sets that can be updated through insertions and deletions, Counting Bloom Filters (CBFs) conduct the same actions. CBFs have been widely employed in memory to reduce the size of datasets for large-scale data processing on massive clusters. For filtering out more redundant data, the false positive likelihood of the CBF should be generated low. This research suggests a multilevel optimization strategy for minimizing the false positive probability of an Accurate Counting Bloom Filter (ACBF). This paper presents an optimized ACBF that maximizes the first level size to reduce false positives while keeping the same functionality as CBF. This paper discovered counting bloom filters' functioning, efficiency, and error rate in the previous method. The results shows that the proposed methods performs better than the standard Bloom Filter.

Keywords: Counting Bloom Filters, Accurate Counting Bloom Filter, Data processing.

1. Introduction

A multilevel optimization strategy to develop an Accurate Counting Bloom Filter is presented in this study. The purpose of the Accurate Counting Bloom Filter is to limit the likelihood of a false positive. The counter vector is partitioned into numerous levels, and offsets indexing is used to manage them. The first level of ACBFs is used to perform set membership queries, while the other levels calculate insertion and deletion counters. We present an algorithm that is ACBF that maximizes the first level size while preserving the same methodology as the regular Counting Bloom Filter to reduce the false positive probability. Visualization studies show that Accurate CBF outperforms CBF in false-positive probability at identical memory usage. A Bloom filter is a basic randomized data structure which describes a set in order to facilitate rapid membership searches. Bloom filters employ separate hash algorithms to represent elements using bits. When querying an element, False positives, or true replies when an element isn't in the set, are allowed by Bloom filters. However, false negatives or giving false when the element is not in the set are not acceptable.

However, when the chance of false positives is low enough, the space savings of Bloom filters generally offset this disadvantage.

Because normal Bloom filters don't allow you to delete items, Bloom filters and variants have become extremely popular. The Counting Bloom Filter (CBF) is a well-known variation that permits the set to alter dynamically through element insertion and deletion. Bloom filters are extended by CBF, which uses a fixed-size counter instead of a single bit for each vector entry. The corresponding counters are incremented when an element is added into CBF; By decrementing the counts, deletions may now be done safely. For most applications, we use four bits per counter to avoid counter overflow. Large-scale data processing, on the other hand, is a huge performance barrier for MapReduce. For starters, online search engines and log processing are examples of data-intensive applications. deal with massive amounts of data. China Mobile, for example, must process 5–8 TB of call records each day. Every day, Facebook collects about six TB of fresh log data. Distributing data over tens of thousands of low-cost devices for division is time intensive for these applications. Memory consumption, processing overhead, and false positive probability are the performance indicators of CBF. The processing overhead, which dominates the CBF throughput, For each primitive operation, is the number of memory accesses. The counter vector size of CBF is the memory consumption. Insertions and deletions are normally supported by four bits per counter. Because the counters consume so much memory, many modifications have been proposed to reduce CBF's memory usage by fitting the entire filter into a small amount of high-capacity memory (i.e., SRAMs). The chance of a false positive is when an element is claimed to be a set member when it is not. A compromise exists between the chance of a false positive and the quantity of memory used.

The primary idea behind ACBF is, partition the counter vector into numerous levels using a multilayer technique for increased accuracy. We can now separate ACBF's query and insertion/deletion operations using this method. This distinction is used to reduce the likelihood of false positives while updating dynamic sets. This is owing to the fact that CBF is being monitored. We find that the CBF counter vector is ideal for rapidly modifying the counts at the bottom by incrementing or decrementing them at the expense of false positive probability. The Figure depicts a simple CBF example using and, where is the number of counters, is the number of hash functions, and is the maximum number of entries.

2. Literature survey

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 (*d*lCBF) [9, 10] is a simple hash-based alternative based on *d*-left hashing and fingerprints. *d*lCBF 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 *d*lCBF 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. Proposed Method

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 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 $b1, b2,...,bd$ and an idle space br. In this hierarchy, $b1$ is used to check the membership for each query, $b2,...,bd$ is used to calculate the counters hashed by each element inserted, and br is used to remain for elements to be inserted.

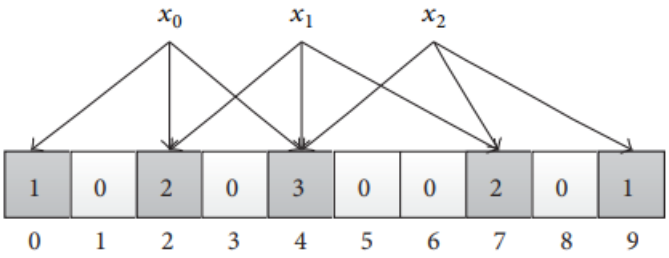


Figure 1: CBF with m = 10, n=5, and k=3.

Figure 2 shows an example of ACBF with four levels $b1$, $b2$, $b3$, and $b4$ and an idle space br. We see that $b1$ has the same size of 9 bits as the counter vector of CBF, and there remain total 21 bits in br used for the idle space. Like the standard CBF, ACBF uses k hash functions $h1, h2,...,hk$ to hash an element x into k bits in the first level $b1$ instead of the counter vector of CBF. Therefore, we only need to check k bits in $b1$ on a membership query.

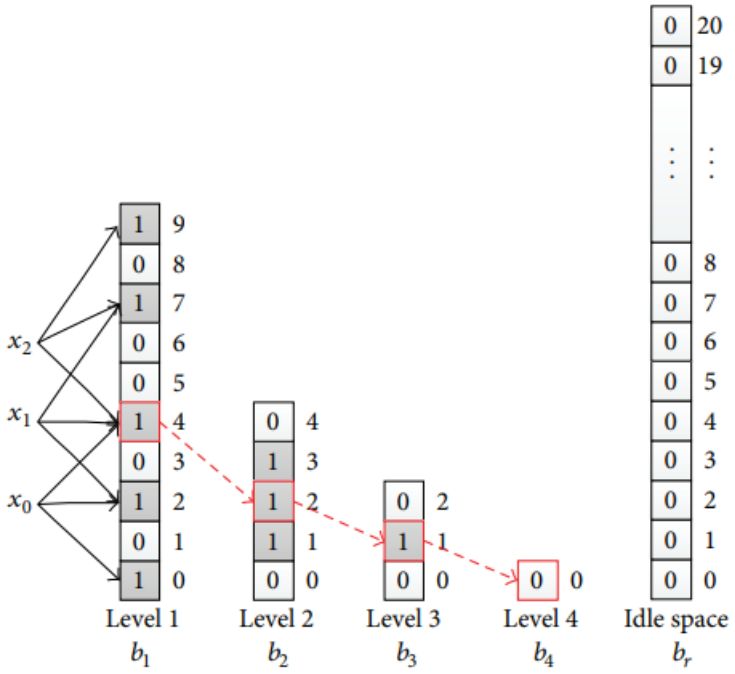


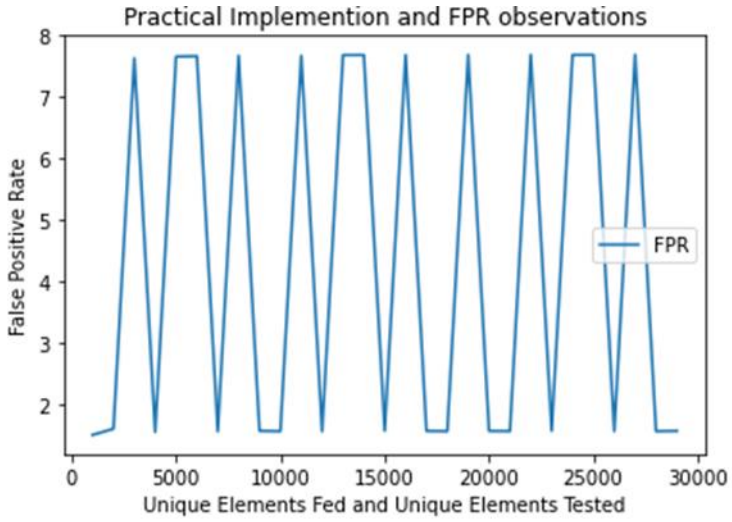
Figure 2: ACBF with four levels and an idle space.

Algorithm 1 shows the query operation in ACBF, where $l1$ is the bit size of $b1$. If all bits $hi(x)$ 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 bi has a size of $li = |bi|$ bits, where i is in the range $[1, \ldots, d]$ and br has $lr = |br|$ bits for the idle space. To span a counter, we recursively calculate an offset index in bi by using a function popcount (bi, j) which computes the number of ones before position j in bi. The offset value returned by popcount (bi, j) is used as an index to the bit in the next level bi+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 $b1$ is hashed by three elements $x0$, $x1$, and $x3$ inserted into ACBF, and its counter spans over four levels $b1$, $b2$, $b3$, and $b4$. We traverse these levels to calculate the counter value at position 4. First, as bit 4 in $b1$ is set to 1, we call popcount $(b1, 4)$ that returns 2 as an offset index in $b2$. Second, we check to see that bit 2 in $b2$ is set to 1 and then call popcount $(b2, 2)$ that returns 1 as an offset index in $b3$. Third, we continue to call popcount $(b3, 1)$ that returns 0 as an offset index in $b4$. Finally, we check to see that bit 0 in $b4$ 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 $b1,...,bj$ 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 $bi+1$ by adding a one bit in bj and shifting upward all bits of $bi+1$ 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 $b1,...,bj$ for each counter hashed by the element (see Lines from 3 to 10 in Algorithm 3) and then shrink the last level bj by shifting backward all the bits of bj, at the same time of removing a one bit from $bj-1$ (see Lines from 12 to 14 in Algorithm 3). For example, we assume that element $x3$ is deleted from ACBF as shown in Figure 2. As position 4 in $b1$ is hashed by $x3$, we traverse level $b1$ to level $b4$ for its counter value. In order to decrement the counter value, we shrink $b4$ by removing a zero bit at position 0 and shrink $b3$ 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: $O\ (k \times [E\ (\varphi) \times (E\ (\lambda) + 1) +1+1])$, (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.

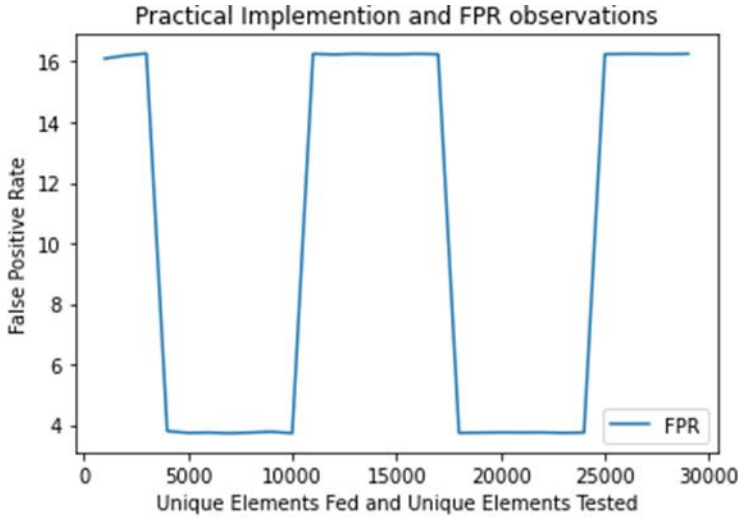
4. Results and Discussions

By constantly changing the False Positive Rate(initializing), we see the number of words that can be transmitted has increased. The words are continuously transmitted without any distortions in the process. The project's major purpose is to decrease the False Positive Rate(fpr). The POC demonstrates it perfectly.




```
FPR initialized as 5.0 %
Practical observation average: 4.302708654133927 %
```

Figure 3. Practical Implementation and FPR observations-1.




```
FPR initialized as 20.0 %
Practical observation average: 10.20874612312953 %
```

Figure 4. Practical Implementation and FPR observations-2

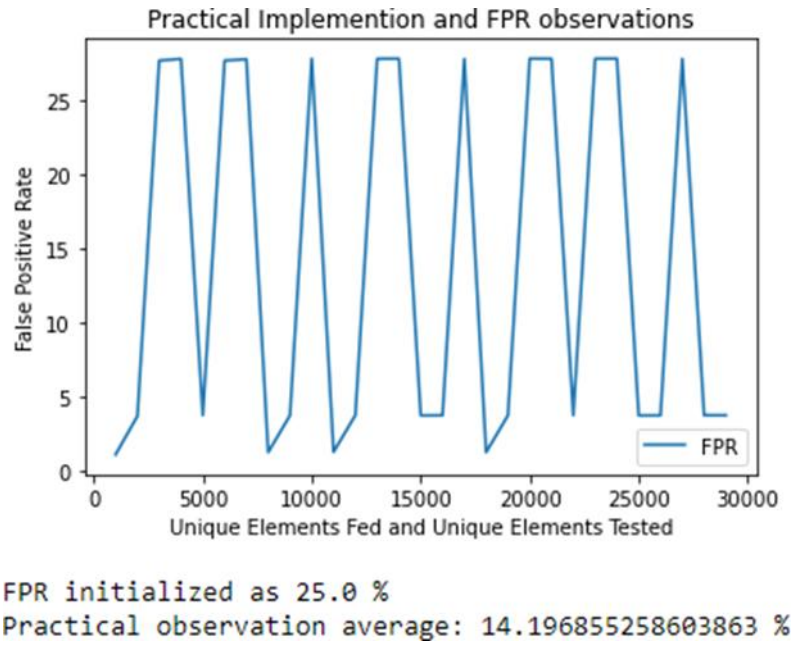


Figure 5. Practical Implementation and FPR observations-3

5. Conclusion

This article presented ACBF, a multilevel optimization strategy for creating an accurate CBF to minimize the false positive probability. The counter vector is partitioned over numerous layers to create ACBF. We present an optimized ACBF that maximizes the first level size while minimizing false positive likelihood and preserving the same functionality as CBF, where is the number of counters, is the number of items, and is the number of hash functions. To boost the reduce-side join performance, we use ACBFs in MapReduce. In the map phase, ACBF is utilized to filter out duplicate records that have been shuffled. ACBF is built in a distributed manner by combining all map jobs' local hash tables. We show that ACBF is a precise data format suited for large-scale data processing.

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 variableincrement 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.