countBF: A General-purpose High Accuracy and Space Efficient Counting Bloom Filter

1st Sabuzima Nayak
Dept. Computer Science & Engineering
National Institute of Technology Silchar
Cachar-788010, Assam, India
sabuzima_rs@cse.nits.ac.in

2nd Ripon Patgiri, *Senior Member, IEEE*
Dept. Computer Science & Engineering
National Institute of Technology Silchar
Cachar-788010, Assam, India
ripon@cse.nits.ac.in

***Abstract*—Bloom Filter is a probabilistic data structure for the membership query, and it has been intensely experimented in various fields to reduce memory consumption and enhance a system's performance. Bloom Filter is classified into two key categories: counting Bloom Filter (CBF), and non-counting Bloom Filter. CBF has a higher false positive probability than standard Bloom Filter (SBF), i.e., CBF uses a higher memory footprint than SBF. But CBF can address the issue of the false negative probability. Notably, SBF is also false negative free, but it cannot support delete operations like CBF. To address these issues, we present a novel counting Bloom Filter based on SBF and 2D Bloom Filter, called countBF. countBF uses a modified murmur hash function to enhance its various requirements, which is experimentally evaluated. Our experimental results show that countBF uses $1.96\times$ and $7.85\times$ less memory than SBF and CBF respectively, while preserving lower false positive probability and execution time than both SBF and CBF. The overall accuracy of countBF is 99.999921, and it proves the superiority of countBF over SBF and CBF. Also, we compare with other state-of-the-art counting Bloom Filters.**

***Index Terms*—Bloom Filter, Counting Bloom Filter, Membership Filter, Frequency Count, Count-Min Sketch, Data Structures.**

I. Introduction

Bloom Filter [1] is an extensively experimented data structure. It is categorized into two key categories, namely, counting and non-counting Bloom Filter. Non-counting Bloom Filter (conventional) is faster than counting Bloom Filter. However, the non-counting Bloom Filter does not support delete operation due to a false negative issue. Delete operation introduces a false positive issue. On the contrary, counting Bloom Filter supports delete operation and can solve false negative issue [2]. However, the false positive probability counting Bloom Filter is higher than the conventional Bloom Filter [3], [4]. Alternatively, Bloom Filter occupies more memory to achieve the desired false positive probability than conventional Bloom Filter. Delete operation is crucial for Bloom Filter. Suppose a database management system integrates Bloom Filter to avoid unnecessary disk accesses and enhance its performance significantly with a tiny amount of memory [5]. Thus, the database management system requires to insert, query, and delete operation for which conventional Bloom Filter is not suitable. Therefore, counting Bloom Filter is used in such kind of requirements.

As we know that counting Bloom Filter has a high false positive probability for which it requires a higher memory footprint than conventional Bloom Filter. CBF can eradicate the false negative issues, but it has a high false positive probability, and high memory footprint. To lower the false positive probability, counting Bloom Filter sacrifices memory footprint. Therefore, we propose a novel counting Bloom Filter to address the above-raised issues, called countBF. The countBF can reduce the false positive probability significantly while preserving a low memory footprint. Our experimental results show that countBF outperforms standard Bloom Filter (SBF) [3] and counting Bloom Filter [2] in every aspect. Key objectives of our proposed system is to reduce memory footprint, to lower false positive probability and to increase its accuracy without compromising the insertion/query performance. Our proposed counting Bloom Filter is similar to the conventional Bloom Filters. countBF has counters, while SBF does not have counters. Also, countBF is implemented in the platform of a 2D Bloom Filter (2DBF) [6]. 2DBF uses a 2D integer array instead of relying on the bitmap array. This 2D integer array is used as a bitmap where each integer represents a block of bits. countBF enhances its performance by tuning the murmur hash function [7]. Murmur hash function is the best non-cryptographic string hash functions [8]. There are also cryptographic string hash functions, however, it does not enhances the performance and the false positive probability [9]. Therefore, we compare countBF with SBF and CBF to evaluate the characteristics using various test cases. In our experimental work, we have compared countBF with SBF because counting Bloom Filter is unable outperform SBF, and thus, it is justified to compare with SBF and CBF. Also, we compare with the other filters with countBF.

II. countBF: The Proposed System

We present a novel counting Bloom Filter, called countBF, by deploying 2-Dimensional Bloom Filter [6]. countBF uses a few arithmetic operations to increase its performance. Let, $\mathbb{B}_{x,y}$ be the two-dimensional integer array to implement counting Bloom Filter where x and y are the dimensions of the filter. The $x \neq y$ and these are prime numbers. A cell of the $\mathbb{B}_{x,y}$ is constituted by η counters, as shown in Figure 1. We have demonstrated 8 bits counters for an example in

978-3-903176-36-2 ©2021 IFIP

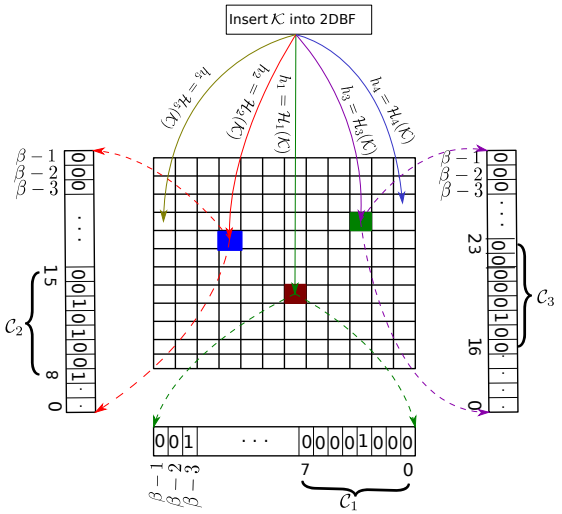

Fig. 1. Architecture of countBF with 8 bit counters

Figure 1. Each counter contains α bits, which is defined by the user, and it can be 1 to 64 bits depending on the user's requirement. The data type of a cell is **unsigned long int** or **unsigned int**. Each cell of $\mathbb{B}_{x,y}$ occupies β $bits$. Therefore, the total number of counters is calculated as $\eta = \frac{\beta}{\alpha}$. Particularly, $\eta = 8$ counters, if each cell occupies $\beta = 64$ bits and each counter contains $\alpha = 8$ bits. Our key objective is to reduce memory footprint, lower the false positive probability, increase the accuracy and enhance the query performance of the counting Bloom Filter. countBF uses two masks, namely, extract mask and reset mask. An extract mask is used to extract the bit information of a counter from a cell. Similarly, a reset mask is used to reset the bit information of a counter in a cell. These two masks are used because of countBF relies on the integer array instead of the bitmap array.

The extract mask are dependent on the number of counters η. Therefore, there are η extract masks which are stored in an array. Extract mask is defined as $\mathcal{M}^e = \{\mathcal{M}^e_1, \mathcal{M}^e_2, \mathcal{M}^e_3, \ldots, \mathcal{M}^e_\eta\}$. For instance, $\mathcal{M}^e_2 = \ldots 0000011111111100000000$ for 8 bits counters. The third extract mask is $0x00000000001FC000$ and it is the correct representation of the extract mask in 64 bits measures for 7 bits counter. Extract masks are used to extract certain counter's value using bit operations. For instance, **unsigned long int** occupies 64 bits and there are 8 counters with 8 bits each. To extract l^{th} counter information, we perform $\mathcal{C}_l = (\mathbb{B}_{i,j} \wedge \mathcal{M}^e_l)$. To remove the trailing zeros, we perform $\mathcal{C}_l = (\mathcal{C}_l >> (\eta * l))$. This $\mathcal{C}_l$ gives the counter information.

Similar to extract mask, there are also η masks for reset a counter to zero. Reset mask is defined as $\mathcal{M}^r = \{\mathcal{M}^r_1, \mathcal{M}^r_2, \mathcal{M}^r_3, \ldots, \mathcal{M}^r_\eta\}$. For instance, $\mathcal{M}^r_2 = \ldots 11111000000011111111$ for a 8 bit counter. The third reset mask is $0xFFFFFFFFFFE03FFF$ and it is the correct representation of the reset mask in 64 bits measures for 7 bits counter. Extract masks are used to extract certain counter's value. Reset masks are used to reset the counter's value to zero. To reset l^{th} counter's value to zero, we need to perform $\mathbb{B}_{i,j} = (\mathbb{B}_{i,j} \wedge \mathcal{M}^r_l)$.

The counting Bloom Filter is comprised of a set of counters to counts the input items. Conventional Bloom Filter uses bitmap array to manipulate the bits to store information of input items. However, countBF does not use bitmap arrays. Instead, it uses a 2D integer array where each cell occupies some memory depending on the data type. For instance, **unsigned long int** occupies 64 bits in modern computers, and therefore, each cell occupies 64 bits memory, and it is initialized by zero. Let η be the total number of counters, α be the bits per counters, and μ be the bits per cell in a 2D array. Therefore, the total number of counters in each cell of countBF is $\eta = \frac{\mu}{\alpha}$, and the remainder is not used. Thus, the total number of masks varies depending on the bits used per counter α.

Algorithm 1 Insertion of an item $\mathcal{K}$ into countBF using k hash functions.

```
1: procedure INSERTION(B_{x,y}, K)
2:     for i = 1 to k do
3:         h = H_i(K, Seed_i)
4:         INCREMENT(B_{x,y}, h)
5:     end for
6: end procedure
```

Algorithm 2 Increment a single counter while inserting an item $\mathcal{K}$ into countBF using a single hash function.

```
1: procedure INCREMENT(B_{x,y}, h)
2:     i = h% x, j = h% y, l = h% η
3:     C_l = B_{i,j} ∧ M^e_l
4:     C_l = C_l >> (α * l)
5:     C_l = C_l + 1
6:     if C_l = MAX then
7:         Counter Overflow.
8:         return
9:     end if
10:    C_l = C_l << (α * l)
11:    B_{i,j} = B_{i,j} ∧ M^r_l
12:    B_{i,j} = B_{i,j} ∨ C_l
13: end procedure
```

countBF is a counting Bloom Filter that comprises many counters. The counters are incremented upon insertion of an item. Let, $\mathbb{B}_{i,j}$ be a 2D Bloom Filter (2DBF) and $\mathcal{C}_l$ be the l^{th} counter in a cell of a 2DBF. Let, $\mathcal{H}()$ be a hash function. We use the murmur hash function. Difference seed values create a different hash value for the same key. For insertion of a single item, countBF calls k hash functions, and the item is inserted into the k counters. Algorithm 1 demonstrates insertion of an item $\mathcal{K}$ using k hash functions. It requires increment the counters' value. Algorithm 2 shows the incrementing process of a counter.

Lookup or query operation is similar to insertion operation except the increment steps. Querying an item $\mathcal{K}$ requires k

Algorithm 3 Lookup an item $\mathcal{K}$ in countBF using k hash functions.

```
1: procedure LOOKUP(B_{x,y}, K)
2:     for i = 1 to k do
3:         h = H_i(K, Seed_i)
4:         flag ← flag ∧ TEST(B_{x,y}, h)
5:     end for
6:     return flag
7: end procedure
```

Algorithm 4 Lookup in a single counter while query an item $\mathcal{K}$ in countBF using a single hash function.

```
1: procedure TEST(B_{x,y}, Hashvalue h)
2:     i = h% x, j = h% y, l = h% η
3:     C_l = B_{i,j} ∧ M_l^e
4:     C_l = C_l >> (α * l)
5:     if C_l ≥ 1 then
6:         return true
7:     else
8:         return false
9:     end if
10: end procedure
```

hash functions with k seed values. The seed values and hash functions of lookup procedure cannot be different from the seed values and hash functions of insertion and delete operations. Algorithm 3 calls murmur hash function k times and TEST() function k times. The TEST() function is demonstrated in Algorithm 4 that returns either $true$ or $false$. The variable $flag$ holds the final result of all test functions and returns the variable $flag$. The $flag$ is a Boolean variable that can hold either $true$ or $false$. All TEST() function results are ANDed which produce Boolean value of $true$ or $false$ and assigned to $flag$.

Algorithm 5 Delete an item $\mathcal{K}$ in countBF using k hash functions.

```
1: procedure DELETE(B_{x,y}, Keys K)
2:     for i = 1 to k do
3:         h = H_i(K, Seed_i)
4:         DECREMENT(B_{x,y}, h)
5:     end for
6: end procedure
```

Conventional Bloom Filter does not support delete operation due to false negatives. The counting Bloom Filter was introduced to address the issue of false negatives [2]. The delete operation creates the issue of false negatives in conventional Bloom Filter. Therefore, counting Bloom Filter is used in many domains. Similar to conventional counting Bloom Filter, countBF also supports a delete operation without any false negative issue. To delete an item $\mathcal{K}$, countBF requires k hash functions call and k DECREMENT() functions calls as demonstrated in Algorithm 5. The insertion and delete operations are required the same steps except for the decrement of the counters in delete operation, as shown in Algorithm 6.

Algorithm 6 Decrement a single counter while deleting an item $\mathcal{K}$ from countBF using a single hash function.

```
1: procedure DECREMENT(B_{x,y}, Hashvalue h)
2:     i = h% x, j = h% y, l = h% η
3:     C_l = B_{i,j} ∧ M_l^e
4:     C_l = C_l >> (α * l)
5:     C_l = C_l − 1
6:     if C_l < 1 then
7:         No deletion.
8:         return
9:     end if
10:    C_l = C_l << (α * l)
11:    B_{i,j} = B_{i,j} ∧ M_l^r
12:    B_{i,j} = B_{i,j} ∨ C_l
13: end procedure
```

III. EXPERIMENTAL RESULTS

Our proposed algorithm is evaluated in 8GB RAM, Intel® Core™ i7-7700 CPU @ 3.60GHz × 8, Ubuntu 18.04.5 LTS and GCC version 7.5.0. We created four different datasets, namely, Same Set, Mixed Set, Disjoint Set, and Random Set. Let, $\mathcal{S} = \{x_1, x_2, x_3, \ldots, x_n\}$ be an inserted set into the Bloom Filter, $\mathcal{Q}$ be the query set. The Same Set defines $\mathcal{S} = \mathcal{Q}$ whereas Disjoint Set is defined as $\mathcal{S} \cap \mathcal{Q} = \phi$. The definition of the Mixed Set follows any one condition, either $q_1 \in \mathcal{S}$ and $q_2 \notin \mathcal{S}$ or $q_1 \notin \mathcal{S}$ and $q_2 \in \mathcal{S}$ where $q_1 \subset \mathcal{Q}$ and $q_2 \subset \mathcal{Q}$. However, the Random Set is randomly generated dataset. These test cases are able to unearth the strengths and weaknesses of a Bloom Filter. We have assessed our proposed countBF for various counter's sizes for the fair judgement.

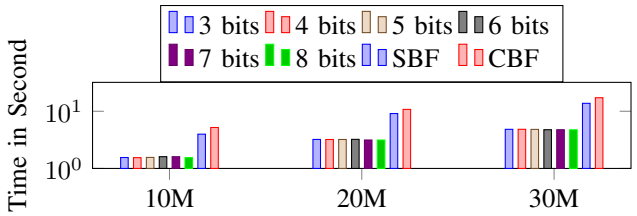

Fig. 2. Comparison of insertion time of countBF using 3 bits, 4 bits, 5 bits, 6 bits, 7 bits, and 8 bits counters with SBF and CBF. Lower is better.

countBF is compared with standard Bloom Filter (SBF) [3] and conventional counting Bloom Filter (CBF) [2]. The countBF is evaluated by setting the counter's bits by 3 bits, 4 bits, 5 bits, 6 bits, 7 bits, and 8 bits. A counter's bit can be a maximum of 64-bits. In our experimental evaluation, each cell occupies 64 bits, and hence, each cell has different counters. The total number of counters for 3 bits, 4 bits, 5 bits, 6 bits, 7 bits, and 8 bits counters are 21, 16, 12, 10, 9, and 8 counters in each cell of countBF. In this configuration, we conduct the experiments, and Figure 2 demonstrates the insertion time taken by countBF, SBF, and CBF. On an average, countBF is

faster than SBF and CBF in the insertion of items. countBF with 4 bits counter is slowest among countBF with 3 bits, 5 bits, 6 bits, 7 bits, and 8 bits counters and countBF with 7 bits counter is fastest among countBF with 3 bits, 4 bits, 5 bits, 6 bits, 7 bits, and 8 bits counters in the insertion of items. Individually, countBF with 4 bits, 8 bits, 7 bits, 7 bits, and 7 bits counters are the fastest in the insertion of 10M, 20M, and 30M dataset, respectively. Overall, countBF with 7 bits is the fastest counter in the insertion operation.

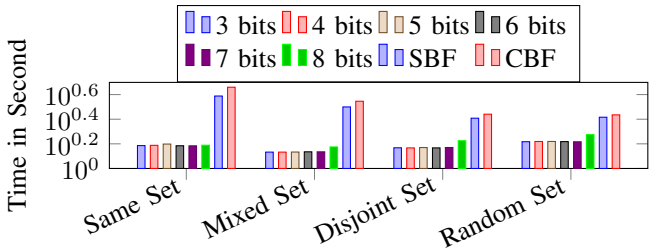

Fig. 3. Comparison of 10M items lookup time by countBF using 3 bits, 4 bits, 5 bits, 6 bits, 7 bits, and 8 bits counters with SBF and CBF. Lower is better.

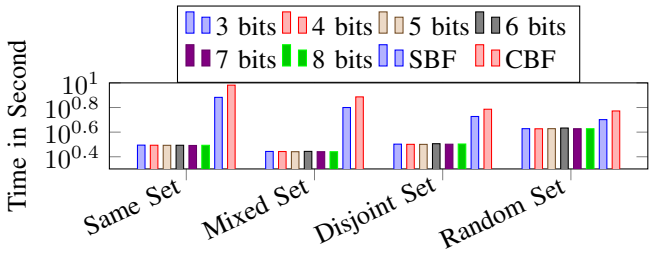

Fig. 4. Comparison of 20M items lookup time by countBF using 3 bits, 4 bits, 5 bits, 6 bits, 7 bits, and 8 bits counters with SBF and CBF. Lower is better.

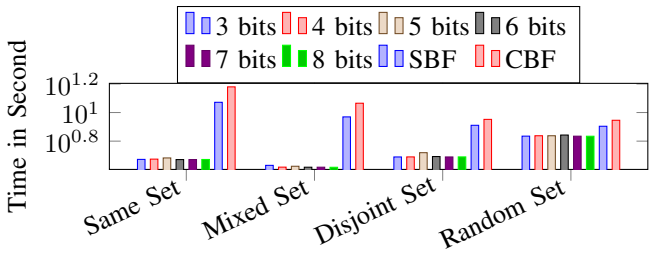

Fig. 5. Comparison of 30M items lookup time by countBF using 3 bits, 4 bits, 5 bits, 6 bits, 7 bits, and 8 bits counters with SBF and CBF. Lower is better.

Figures 3, 4, and 5 demonstrates the lookup operations of 10M, 20M, and 30M items respectively by countBF, SBF and CBF. countBF is faster than SBF and CBF in all sized query with all test cases. On an average, countBF with 3 bits, 7 bits, 7 bits, 6 bits, and 3 bits counters are the fastest in 10M, 20M, and 30M items lookup, respectively. It is observed that countBF with 8 bits counter exhibits the worst performance overall. For any test case, countBF with 4 bits, 8 bits, 8 bits, 7 bits, and 6 bits are the fastest in the insertion of 10M, 20M, and 30M dataset, respectively. Overall, countBF with 7 bits is the fastest filter. countBF outperforms SBF and CBF by 50.73% and 55.68%, 45.33% and 54.67%, and 44.84% and 53.82% in 10M, 20M, and 30M items' lookup respectively. Overall, our proposed system has a higher lookup performance rate than SBF and CBF.

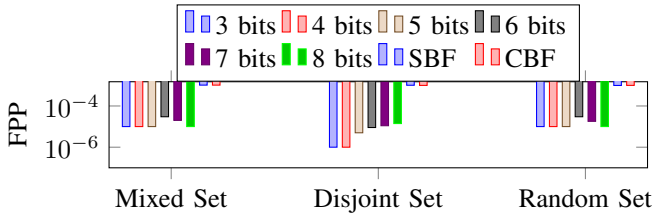

Fig. 6. Comparison of FPP in 10M lookup of countBF using 3 bits, 4 bits, 5 bits, 6 bits, 7 bits, and 8 bits counters with SBF and CBF in desired FPP of 0.001 setting. Lower is better.

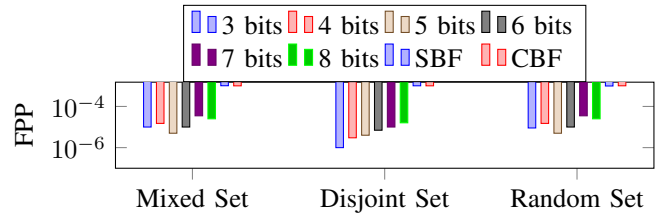

Fig. 7. Comparison of FPP in 20M lookup of countBF using 3 bits, 4 bits, 5 bits, 6 bits, 7 bits, and 8 bits counters with SBF and CBF in desired FPP of 0.001 setting. Lower is better.

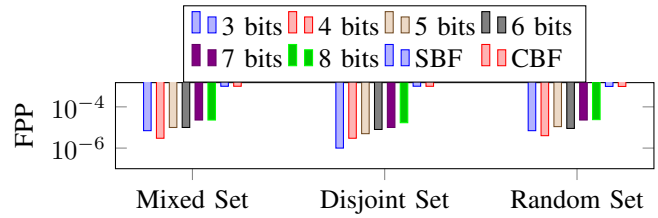

Fig. 8. Comparison of FPP of 30M lookup in countBF using 3 bits, 4 bits, 5 bits, 6 bits, 7 bits, and 8 bits counters with SBF and CBF in desired FPP of 0.001 setting. Lower is better.

Figures 6, 7, and 8 demonstrate the false positive probability of countBF, SBF and CBF in log scale. Counting Bloom Filters are designed to deal with the false negative issue. Therefore, conventional counting Bloom Filter has an issue of trade-off between false positive probability and memory consumption. Normally, counting Bloom Filters have a higher false positive probability than other variants of the membership filter. However, countBF outperforms in the false positive probability in Mixed Set, Disjoint Set, and Random Set queries. There is no false positive probability for the Same Set query. countBF exhibits an excellent false positive probability in Disjoint Set query, and it is zero in Disjoint set for the lookup of 10M dataset, which is demonstrated

in Figure 7. However, the highest false positive probability of countBF is recorded as 0.000035. countBF with 7 bits counter exhibits the highest false positive probability in 20M dataset lookup. On an average, the lowest and highest false positive probability of countBF is 0.000006 and 0.000022, respectively. Overall, the false positive probability of countBF is 0.000013. On an average, the false positive probability of SBF and CBF are 0.00100453 and 0.00099913, where CBF exhibits a lower false positive probability. It is possible due to the higher memory footprint of CBF than SBF. Thus, our proposed system exhibits the lowest false positive probability, whereas SBF and CBF exhibit equivalent false positive probability. Both SBF and CBF are configured to the desired false positive probability as 0.001, and therefore, their false positive probability is equivalent.

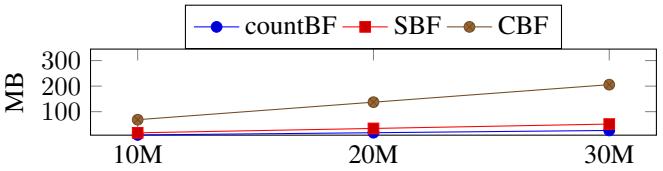

Fig. 9. Comparison of memory consumption of countBF, SBF and CBF in desired FPP of 0.001 setting. Lower is better.

As we know that the CBF has a higher false positive probability than SBF. On the contrary, the experimental results show that SBF and CBF similar false positive probability. We have indeed configured the desired false positive probability to 0.001, and therefore, the false positive probability of SBF and CBF are equivalent. However, the memory requirements are different. CBF uses higher memory than SBF. It means CBF has a higher false positive probability than SBF. We adjust the memory allocation to achieve the desired false positive probability. Therefore, CBF has allocated more memory to achieve the desired false positive probability. Thus, counting Bloom Filter occupies more memory to achieve the desired false positive probability. Notably, counting Bloom Filter consume more memory than other variants of Bloom Filter or membership filter to achieve certain false positive probability. On the contrary, SBF uses $1.96\times$ more memory than countBF, and CBF uses $7.85\times$ more memory than countBF on an average. The lower memory footprint of countBF shows the highest accuracy and the lowest false positive probability. On an average, countBF, SBF and CBF consume memory of 26.11 MB, 51.42 MB, and 205.67 MB for all datasets, respectively. The memory, false positive probability, and accuracy are the key decisive factor of the Bloom Filter. However, countBF also faster than SBF and CBF. But there are much faster membership filters available, however, countBF is more accurate than any other counting variant of Bloom Filters.

Figure 10 demonstrates the bits per item of the countBF, SBF and CBF. countBF uses the lowest bits per item, and CBF uses the highest bits per item. countBF, SBF, and CBF use 7.32 bits, 14.38 bits, and 57.51 bits per item on average, respectively. Our proposed counting Bloom Filter uses the lowest bits per item and provides a lower false positive probability.

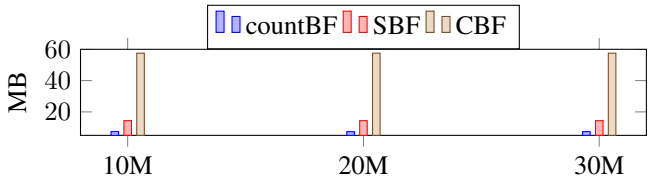

Fig. 10. Comparison of bits per item of countBF, SBF and CBF. Lower is better.

IV. Conclusion

This article has demonstrated our proposed counting Bloom Filter, called countBF, significantly improved over SBF and CBF. countBF can insert, query, and delete an item in $O(k)$ time complexity while preserving high accuracy, low false positive probability, low memory footprint, and fast execution time. Moreover, the properties of countBF make more room for incoming items. We have evaluated the false positive probability using various test cases experimentally. The false positive probability is lower than the SBF and CBF. Alternatively, the accuracy of countBF is higher than SBF and CBF. Therefore, countBF is able to outperform the existing Bloom Filter in terms of false positives, accuracy, memory footprint, and performance. Also, we have compared with various state-of-the-art counting Bloom Filters and it shows that our proposed counting Bloom Filter is an ideal solution. Moreover, we have demonstrated how to adapt countBF in frequency count, similar to CMS. It can also be applied in diverse domains where delete operation is a crucial part of the system.

References

[1] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[2] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.
[3] A. Kirsch and M. Mitzenmacher, *Less Hashing, Same Performance: Building a Better Bloom Filter*. LNCS, Springer, 2006, pp. 456–467.
[4] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better Bloom filter," *Random Structures Algorithms*, vol. 33, no. 2, pp. 187–218, Sep 2008.
[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, 2008.
[6] R. Patgiri, S. Nayak, and S. K. Borgohain, "rDBF: A r-dimensional bloom filter for massive scale membership query," *Journal of Network and Computer Applications*, vol. 136, pp. 100 – 113, 2019.
[7] A. Appleby, "Murmurhash," Retrieved on August 2018 from https://sites.google.com/site/murmurhash/.
[8] A. Appleby, "Statistics," Accessed on May 2020 from https://sites.google.com/site/murmurhash/statistics, 2020.
[9] R. Patgiri, S. Nayak, and N. B. Muppalaneni, "Is bloom filter a bad choice for security and privacy?" in *2021 International Conference on Information Networking (ICOIN)*, 2021, pp. 648–653.