robustBF: A High Accuracy and Memory Efficient 2D Bloom Filter

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 an important probabilistic data structure to reduce memory consumption for membership filters. It is applied in diverse domains such as Computer Networking, Network Security and Privacy, IoT, Edge Computing, Cloud Computing, Big Data, and Biometrics. But Bloom Filter has an issue of the false positive probability. To address this issue, we propose a novel robust Bloom Filter, robustBF for short. robustBF is a 2D Bloom Filter, capable of filtering millions of data with high accuracy without compromising the performance. Our proposed system is presented in two-fold. Firstly, we modify the murmur hash function, and test all modified hash functions in 2D Bloom Filter. Our experimental results show that robustBF is better than standard Bloom Filter and counting Bloom Filter in every aspect. robustBF exhibits nearly zero false positive probability with more than $10 \times$ and $44 \times$ lower memory consumption than standard Bloom filter and counting Bloom Filter, respectively.

Index Terms—Bloom Filter, Membership Filter, Approximation Algorithms, Data Structures, Algorithms.

I. INTRODUCTION

Bloom Filter [1] is a membership filter that is capable of filtering a large number of data with a tiny amount of memory. Therefore, Bloom Filter meets diverse applications and it is applied in diverse domains, namely, Computer Networking [2], [3], Network Security [4] and Privacy [5], IoT [6], Big Data [7], Cloud Computing [8], Biometrics [9] and Bioinformatics [10]. There are diverse variants of Bloom Filter available [11]–[13]; however, these filters are unable to provide high accuracy using tiny memory without compromising query performance. Moreover, the false positive probability is an issue of Bloom Filter, and it cannot be reduced without increasing the memory size. There are many fast filters available, for instance, Morton Filter [14] and XOR Filter [15]. Morton Filter is faster than Cuckoo Filter. Morton Filter extends Cuckoo Filter and implements a compressed Cuckoo Filter.

In this paper, we propose a novel Bloom Filter, called robustBF, is to reduce the false positive probability. robustBF is a 2D Bloom Filter, which modifies the murmur hash function for better performance. The outcomes of the proposed system are outlined as follows:

- robustBF is a fast filtering Bloom Filter, and it is faster than standard Bloom Filter (SBF) [16] and counting Bloom Filter (CBF) [17]. robustBF is $2.038 \times$ and $2.48 \times$ faster in insertion of 10M data than SBF and CBF respectively.
- robustBF consumes $10.40 \times$ and $44.01 \times$ less memory than SBF and CBF on an average, respectively.
- robustBF exhibits false positive probability almost zero in the desired false positive probability setting of 0.001 with lower memory consumption. Thus, the accuracy of robustBF is almost 100%.

We compare robustBF with SBF and CBF. We found that robustBF is better in the false positive probability and more efficient in memory consumption than the other filters’ variants. To the best of our knowledge, robustBF is the only Bloom Filter that can increase its accuracy and lower the memory footprint without compromising the filter’s performance. Moreover, we also compare Cuckoo Filter (CF). Experimental results show that CF is faster than robustBF in insertion and lookup operations. But CF consumes huge memory than robustBF. Moreover, it exhibits the worst false positive probability in disjoint set and random set lookup process.

II. ROBUSTBF- THE PROPOSED SYSTEMS

We propose a novel Bloom Filter based on 2D Bloom Filter (2DBF) [18], called robustBF. We modify the existing hash function and embed the modified hash function to constructs a new Bloom Filter. robustBF modifies the hashing technique to increase its performance. The performance of robustBF depends on the murmur hash functions, which introduces some biases and the number of hash function calls. This bias helps in improving the accuracy, and hence, reduces the false positive probability. robustBF is designed by aiming to provide high accuracy and memory-efficient Bloom Filter.

A. Selection of a hash function

The performance of the Bloom Filter depends on the hash functions. There are many hash functions available, namely, xxHash, SuperFastHash, FastHash, FNV, and murmur hash functions. Murmur hash function is the finest hash function among all non-cryptography string hash functions [19]. There are also cryptographic string hash functions, namely, SHA1, SHA2, and MD5. These string hash functions are the best for security purposes. However, these cryptographic string hash functions are unable to reduce the false positive probability [20]. Moreover, it degrades the performance of the
Bloom Filter. Thus, we modify the murmur hash functions and evaluated the best modification among the modified hash functions. Murmur hash function depends on the bit mixture and scanning of the input, and thus, we modify the murmur hash function and term it as H1, H2, H3, H4, H5, H6, and H7. The modified murmur hash function H1 reads 3 bytes of data at a time. Similarly, the modified murmur hash function H2 reads 4 bytes at a time. Thus, H3, H4, H5, H6, H7, H8, and H9 reads 5, 6, 7, 8, 9, 10, and 11 bytes of data at a time. It shows a huge difference in changing the reading of data at a time in murmur hash functions. The experimental results show that the H4 exhibits the best performance.

B. Designing the 2D Bloom Filter

The detailed architecture of 2D Bloom Filter is depicted in Fig. 1. Each cell of 2DBF of Figure 1 contains β bits. Initially, an item is hashed into 2DBF using k distinct hash functions, i.e., the item is mapped into k cells of 2DBF and the bits are set to 1. Let, B_{X,Y} be the 2D Bloom Filter (2DBF) where X and Y are the dimensions, and these are prime numbers. Let, \( H() \) be the modified murmur hash function, and K is an item to be inserted. Let, \( B_{i,j} \) and is the particular cell in \( B_{X,Y} \) filter. To insert, Equation (1) is invoked.

\[
B_{i,j} \leftarrow B_{i,j} \ OR \ P \tag{1}
\]

where \( h = H(K) \), \( i = h \% X \), \( j = h \% Y \), \( \rho = h \% \beta \) and \( P = (1 \ll \rho) \), where % is modulus operator, \( \beta \) is the size of a cell and \( \ll \) is bitwise left shift operator. The \( \beta \) should be also prime number. Let us assume, unsigned long int occupies 64 bits, then the the nearest prime number is 61. Therefore, \( \beta \) should be 61 but it cannot be greater then the size of a cell in 2DBF.

The deletion operation is performed using XOR operator. To remove \( K \) from 2DBF, Equation (2) is invoked.

\[
B_{i,j} \leftarrow ((B_{i,j} \ AND \ P) \ OR \ \Phi) \ L\ OR \ P \tag{2}
\]

For lookup operation (query operation) of an item \( K \), Equation (3) is invoked.

\[
F_{2DBF} \leftarrow (B_{i,j} \ AND \ P) \ OR \ \Phi \tag{3}
\]

If \( F_{2DBF} \) is 1, then the key is a member of 2DBF and otherwise, it is not a member of 2DBF. Equation (1), (2) and (3) depends on the hash function \( H() \).

III. Experimental Results

We evaluate the accuracy, false positive probability, query and insertion performance, and memory consumption of robustBF. We have conducted our experiment in Desktop PC with the configuration of Intel Core i7-7700 CPU @ 3.60GHz x 8, 8GB RAM, 1TB HDD, Ubuntu 18.04.5 LTS, and GCC-7.5.0.

A. Test cases

We have created four different test cases to validate the accuracy, performance, memory efficiency and false positive rate. The four test cases are Same Set, Mixed Set, Disjoint Set and Random Set which are defined in Definition 1, 2, 3 and 4 respectively. These test cases are used to evaluate the strengths and weaknesses of Bloom Filter in every aspect. Let, \( S = \{s_1, s_2, s_3, \ldots, s_m\} \) be the input set and input into the robustBF.

Definition 1: Let, \( Q \) is a set queried where \( Q = S \), then the set \( Q \) is called Same Set.

Definition 2: Let, \( Q = \{Q^1, Q^2\} \) be a query set where \( Q^1 = \{Q^1_1, Q^1_2, Q^1_3, \ldots\} \) and \( Q^1 = \{Q^2_1, Q^2_2, Q^2_3, \ldots\} \) such that \( Q^1 \subset S \) and \( Q^2 \cap S = \phi \), then the set \( Q \) is called Mixed Set.

Definition 3: Let, \( Q \) be a query set where \( Q \cap S = \phi \), then the set \( Q \) is called Random Set.

Definition 4: Let, \( Q \) be a query set randomly generated, then the set \( Q \) is called disjoint set.

B. Experiments

This section presents the accuracy of Bloom Filters and its memory requirements. robustBF inherits the properties from rDBF. We compare robustBF with standard Bloom Filter (SBF) where SBF is a standard Bloom Filter to benchmark. The robustBF, SBF, and CBF are single-threaded Bloom Filter. Single-threaded Bloom Filters are useful as an enhancer of a system, while multi-threaded Bloom Filter is a complete system. robustBF outperforms SBF and CBF in every aspect, which is presented in this section.

C. Hash function experimentation

robustBF is a two-dimensional Bloom Filter and depends on the murmur hash function. The insertion performances of H1, H2, H3, H4, H5, H6, H7, H8, and H9 are demonstrated in Figure 2. A total of 10M data is inserted into robustBF. The performance of robustBF is the lowest in H1 and the highest in H9. However, the insertion performance is not as important as the lookup operation.

The Bloom Filter is designed to provide higher lookup performance with a tiny amount of memory usage. robustBF is evaluated its performance in Same Set, Mixed Set, Disjoint Set, and Random Set to reveal the strength and weakness. robustBF is evaluated with modified murmur hash function.
H1, H2, H3, H4, H5, H6, H7, H8, and H9. robustBF exhibits a similar lookup performance of 10M queries in H3, H4, H5, H6, H7, H8, and H9. Therefore, we select a modified murmur hash function from H3, H4, H5, H6, H7, H8, and H9. However, it is challenging to conclude the best performer because their performance differs in different test cases.

robustBF measures its accuracy as $A = 1 - FPP$ since there are no false negatives. Figure 4 demonstrates the accuracy of robustBF using modified murmur hash functions H1, H2, H3, H4, H5, H6, H7, H8, and H9. As per our observation, the H4 has the highest accuracy and lowest false positive probability in all test cases. The false positive probability is demonstrated in Figure 5. The H6, H7, H8, and H9 exhibit the highest performance; however, the false positive probability of these hash functions is poor. Therefore, we are simply striking out the modified murmur hash function H6, H7, H8, and H9. The remaining hash functions are H3, H4, and H5 to compare their performance.

D. Comparison of robustBF with other filters

The comparison of insertion time of robustBF and SBF is demonstrated in Figure 7. robustBF, SBF, and CBF perform 5.426 million operations per second (MOPS), 2.182 MOPS, and 1.87 MOPS, respectively. robustBF performance is measured using H3, H4, and H5 by inserting 10M, 20M, 30M, 40M, and 50M data. robustBF performance is similar to H4 and H5; however, its performance is lower in H3 as depicted in Figure 6. We observed that the H4 is the best performer among the modified murmur hash functions. However, H4 is not the best performer in insertion and lookup, but it is best in false positive probability. Therefore, we conclude that H4 is the best choice for robustBF. The rest of the experiment of robustBF is performed using the H4 function because H4 exhibits its best performance in the false positive probability, which is crucial for Bloom Filter.

The comparison of insertion time of robustBF and SBF is demonstrated in Figure 8. robustBF, SBF, and CBF perform 5.426 million operations per second (MOPS), 2.182 MOPS, and 1.87 MOPS, respectively.
and 1.79 MOPS on average, respectively. robustBF is faster than SBF in insertion.

Table I shows the false positive probability of robustBF and SBF in different test cases with 10 million to 100 million of dataset. robustBF and SBF do not exhibit any false positive probability in the Same Set. However, there is a false positive probability in other test cases. robustBF exhibits the highest false positive probability in Mixed Set, which is much lower than SBF. It is almost zero false positive probability. However, the configuration of SBF and CBF is 0.001, and hence, it shows a constantly similar false positive probability.

Table II demonstrates the accuracy of robustBF and SBF. Both robustBF, SBF, and CBF exhibit 100% accuracy in the Same Set. The accuracy of robustBF is much higher than SBF and CBF. The lowest accuracy of robustBF is 99.999% in 20M and 10M datasets with the Mixed Set test cases.

Interestingly, robustBF uses a tiny amount of memory. SBF uses higher memory than robustBF, as shown in Figure 10. robustBF, SBF, and CBF require 1.382 bits, 14.378 bits, and 57.511 bits memory per elements. Therefore, robustBF consumes 10.40× and 44.01× lower memory than SBF and CBF, respectively. Therefore, robustBF is better than SBF and CBF in memory consumption.

Now, let us compare robustBF with Cuckoo Filter [21] which is not a Bloom Filter. Cuckoo Filter is a filter based...
TABLE I: False positive probability of robustBF and SBF in various test cases. Lower is better.

| Dataset | MS_robustBF | MS_SBF | MS_CBF | DS_robustBF | DS_SBF | DS_CBF | RS_robustBF | RS_SBF | RS_CBF |
|---------|-------------|--------|--------|-------------|--------|--------|-------------|--------|--------|
| 10M     | 0.000010    | 0.001028 | 0.001027 | 0.001012    | 0.000996 | 0.000996 | 0.000998    | 0.000989 | 0.000998 |
| 20M     | 0.000010    | 0.001027 | 0.001028 | 0.001002    | 0.000999 | 0.000999 | 0.000998    | 0.000989 | 0.001002 |
| 30M     | 0.000007    | 0.000998 | 0.001000 | 0.001003    | 0.000996 | 0.000996 | 0.000998    | 0.000981 | 0.000976 |
| 40M     | 0.000003    | 0.001019 | 0.001001 | 0.001022    | 0.001033 | 0.000999 | 0.000998    | 0.000981 | 0.000977 |
| 50M     | 0.000002    | 0.000994 | 0.000999 | 0.001008    | 0.001008 | 0.000999 | 0.000998    | 0.000975 | 0.000986 |

TABLE II: Accuracy of robustBF, SBF, and CBF in Same Set (SS), Mixed Set (MS), Disjoint Set (DS) and Random Set (RS). Higher is better.

| Interval | MS_robustBF | MS_SBF | MS_CBF | DS_robustBF | DS_SBF | DS_CBF | RS_robustBF | RS_SBF | RS_CBF |
|----------|-------------|--------|--------|-------------|--------|--------|-------------|--------|--------|
| 10M      | 99.999      | 99.8972 | 99.8973 | 100         | 99.8988 | 99.9004 | 99.9999     | 99.9011 | 99.9002 |
| 20M      | 99.999      | 99.8991 | 99.9004 | 99.9999     | 99.8998 | 99.9001 | 100         | 99.902    | 99.8998 |
| 30M      | 99.9993     | 99.9002 | 99.9001 | 100         | 99.8997 | 99.9004 | 100         | 99.9019   | 99.9024 |
| 40M      | 99.9997     | 99.8981 | 99.8999 | 100         | 99.8978 | 99.8997 | 100         | 99.9       | 99.9023 |
| 50M      | 99.9998     | 99.8987 | 99.9006 | 100         | 99.8992 | 99.9001 | 100         | 99.9024   | 99.9014 |

on cuckoo hashing. Figure 11 demonstrates the insertion time taken by robustBF and CF. CF is the fastest filter in the insertion of 10M data into the filters. Here, robustBF takes the highest times, which is much slower than CF.

Fig. 12: Comparison of lookup time of 10M in robustBF, and CF in second. Lower is better.

Similar to insertion, robustBF is slower in lookup than CF, as shown in Figure 12. CF is the fast filter to perform the lookup of 10M data.

TABLE III: Comparison of false positive probability in lookup of 10M in robustBF and CF. Lower is better.

| Test cases | robustBF | CF  |
|------------|----------|-----|
| Same Set   | 0        | 0   |
| Mixed Set  | 1E-05    | 0.0005592 |
| Disjoint Set | 0      | 0.5749 |
| Random Set | 0        | 0.995351 |

The false positive probability of robustBF is almost zero, which is demonstrated in Table III. But, CF exhibits good false positive probability in the Same Set and Mixed set, whereas it shows the worst in Disjoint Set and Random Set of data due to the kicking process.

Figure 13 demonstrates the memory consumption in 10M data by robustBF and CF. robustBF and CF occupy 1.55MB, and 24MB for 10M data, respectively.

IV. ANALYSIS

Let, \( m \) be the number of bits of the Bloom Filter \( B \) and the probability to be a particular bit is not set to 1 is \( (1 - \frac{1}{m}) \). There are \( k \) the number of hash functions and \( n \) number of total items to be inserted into the Bloom Filter \( B \), then the probability of a bit not set to 1 is \( (1 - \frac{1}{m})^n k \). Therefore, the probability of a particular bit is set to 1 is

\[
1 - \left(1 - \frac{1}{m}\right)^{nk} \tag{4}
\]

Thus, the desired false positive probability is

\[
\varepsilon = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right) \approx \left(1 - e^{-kn/m}\right)^k \tag{5}
\]

The number of hash functions \( k \) cannot be too large or too small. Bloom Filter requires an optimal number of hash functions. The assumption of the optimal number of hash function is \( k = \frac{m}{n} \ln 2 \). Optimal number of hash function is an assumption and it works well with conventional Bloom Filter. It gives the optimal number of hash functions. Solving Equation (5) using the assumption of number of hash functions, we get

\[
\varepsilon = \left(1 - e^{-\frac{(\frac{m}{n} \ln 2)n}{m}}\right)^{\frac{m}{n} \ln 2} \tag{6}
\]
By simplifying Equation (5), we get

$$\ln \varepsilon = -\frac{m}{n} (\ln 2)^2, \quad \implies m = - \frac{n \ln \varepsilon}{(\ln 2)^2} \quad (7)$$

The memory requirement is $m$ depending on the desired false positive probability $\varepsilon$ and number of input items $n$. robustBF uses approximately half of the desired memory calculated by Equation (7).

### A. Memory requirements of robustBF

robustBF is a 2D Bloom Filter (2DBF), and thus, the memory requirement not only depends on the $n$ and $\varepsilon$ but also its dimensions of the array. Let, $X$ and $Y$ be the dimensions of robustBF. The dimensions $X \neq Y$ and these are prime numbers. If $X$ and $Y$ are not prime numbers, then false positive increases. We need to calculate $X$ and $Y$ from $m$. Let, $P_i$ be the array of prime numbers and $q = \frac{m}{\sqrt{2\ln 2}}$ where the $\beta$ is the number of bits per cell in robustBF. Now, we calculate $t = \sqrt{\varepsilon}$ and call the function $i = \text{SELECTPRIME}(t)$ as shown in Algorithm 1.

**Algorithm 1:** Index calculation of prime number array $P_i$

```plaintext
1: procedure SELECTPRIME(t)
2: for $i = 1$ to TotalNumberOfPrime do
3: if $P_i > t$ then return $i$;
4: end if
5: end for
6: end procedure
```

The dimension $X = P_{i+3}$ and $Y = P_{i-3}$. Thus, robustBF allocated unsigned long int 2D array, i.e., the total number of memory bit is $X \times Y \times \beta$ where $\beta$ is the bit size of unsigned long int. Also, robustBF uses exactly half of the number of optimal hash functions calculated because it performs the modulus operation using dimensions $X$ and $Y$ to place an item in the filter.

### V. Conclusion

In this paper, we present a novel Bloom Filter, called robustBF. We have demonstrated that robustBF is able to reduce the false positive probability to almost zero with desired false positive probability. Also, robustBF consumes approximately $10 \times$ and $44 \times$ less memory than SBF and CBF with the same settings. We have also demonstrated its accuracy, which is satisfactorily high. The lookup and insertion performance of robustBF is higher than SBF and CBF. No Bloom filter can increase its accuracy by lowering the memory footprint significantly to the best of our knowledge. There are diverse faster filters available than robustBF. The key objectives of robustBF is to improve accuracy and lower memory footprint without compromising the filter's performance. Therefore, the robustBF performance is compared with SBF and CBF. We agree that there are many faster filter available than robustBF, for instance, Morton Filter [14], and XOR Filter [15]; but these filters use large memory per items. Therefore, robustBF can improve various applications' performance using a tiny amount of memory in Computer Networking, Security and Privacy, Blockchain, IoT, Big Data, Cloud Computing, Biometrics, and Bioinformatics.