Shed More Light on Bloom Filter's Variants

Ripon Patgiri, Sabuzima Nayak, and Samir Kumar Borgohain
Department Of Computer Science & Engineering, National Institute of Technology Silchar, Assam, India

Abstract— *Bloom Filter is a probabilistic membership data structure and it is the excessively used for membership query. Bloom Filter becomes the predominant data structure in approximate membership filtering. Bloom Filter extremely enhances the query response time, and the response time which is $O(1)$ time complexity. Bloom filter (BF) is used to detect whether an element belongs to a given set or not. The Bloom Filter returns **True Positive (TP)**, **False Positive (FP)**, or **True Negative (TN)**. The Bloom Filter is widely adapted in numerous areas to enhance the performance of a system. In this paper, we present a) in-depth insight on the Bloom Filter, b) the prominent variants of the Bloom Filters, and c) issues and challenges of the Bloom Filter.*

Keywords: Bloom Filter, Scalable Bloom Filter, Variants of Bloom Filter, Membership filter, Data Structure, Algorithm

1. Introduction

The Bloom Filter [1] is the extensively used probabilistic data structure for membership filtering. The query response of Bloom Filter is unbelievably fast, and it is in $O(1)$ time complexity using a small space overhead. The Bloom Filter is used to boost up query response time, and it avoids some unnecessary searching. The Bloom Filter is a small sized data structure. The query is performed on the Bloom Filter before querying to the large database. The Bloom Filter saves immense query response time cumulatively. However, there is also a false positive which is known as overhead of the Bloom Filter. Nevertheless, the probability of the false positive is very low. Thus, the overhead is also low. Moreover, a careful implementation of Bloom Filter is required to reduce the probability of false positive.

There are various kind of Bloom Filters available, namely, Blocked Bloom Filter [2], Cuckoo Bloom Filter [3], d-Left CBF (dlCBF) [4], Quotient Filter (QF) [5], Scalable Bloom Filter (SBF) [6], Sliding Bloom Filter [7], TinySet [8], Ternary Bloom Filter (TBF) [9], Bloofi [10], OpenFlow [11], BloomFlow [12], Difference Bloom Filter (DBF) [13], and Dynamic Reordering Bloom Filter [14]. The variants of Bloom Filters are designed based on the requirements of the applications. The Bloom Filter data structure is highly adaptable. Therefore, the Bloom Filter has met a enormous applications. A careful adaptation of the Bloom Filter ameliorates the system. However, it depends on the requirements of the applications. Bloom Filter's improvement potentiality makes the vast applicability of the probabilistic data structure.

The fast query response using Bloom Filter attracts all the researchers, developers, and practitioners. There are tremendous applications of Bloom Filter. For instance, the BigTable uses Bloom Filter to improve disk access time significantly [15]. Moreover, the Metadata Server is drastically enhanced by Bloom Filter [16], [17], [18], [19]. The Network Security is also boosted up using Bloom Filter [20], [21]. In addition, the duplicate packet filter is a very time consuming process. The duplicate packets are filtered in O(1) time complexity using Bloom Filter [22]. Besides, there are diverse applications of Bloom Filter which improve significantly the performance of a system. The Bloom Filter predominant the filtering system, and thus poses some research questions (RQ) which are listed below-

- RQ1: Where should not Bloom Filter be used?
- RQ2: What is the barrier of Bloom Filter?
- RQ3: What are the various kinds of Bloom Filters available?
- RQ4: What are the issues and challenges of Bloom Filter?

The research question (RQ) leads the article to draw a suitable conclusion. The RQ1 exposes the reason for using Bloom Filter. The RQ2 exploits the False Positive of Bloom Filter. The RQ3 exposes the state-of-the-art development of Bloom Filter. And finally, the RQ4 poses the issues and challenges of Bloom Filter.

2. Bloom Filter

The Bloom Filter [1] is a probabilistic data structure to test an element membership in a set [23]. The Bloom Filter uses a small space overhead to store the information of the element set. The True Positive and True Negative enhance the performance of filter mechanism. However, there false positive overhead in the Bloom Filter variants. However, the probability of false positive is negligible. But, some system cannot tolerate False Positive, because the false positive introduces error to the system. For example, duplicate key filtering system. Moreover, it also guarantees that there is no False Negative (FN) except counting variants of Bloom Filter. There are many systems where most of the queries are TN. Let $K = k_1, k_2, k_3, \ldots, k_n$ be elements present in the set S. Let k_i be the random element where $1 \leq i$. The approximate membership query is whether $k_i \in S$ or not. The Bloom Filter returns either positive or negative. The positive is classified into False Positive (FP) and True Positive (TP). The FP of Bloom Filter returns existence

of an element in a set, but $k_i \notin S$. However, the TP correctly identifies the element, and it is in the set. Similarly, the negative is also classified into TN and FN. The TN boosts up the performance of a system and FP degrades the performance of a system. Therefore, the key challenge of Bloom Filter design is to reduce the probability of FP. The Figure 1 depicts the flowchart of Bloom Filter. The Figure 1 clearly exposes the overhead of Bloom Filter in case of FP.

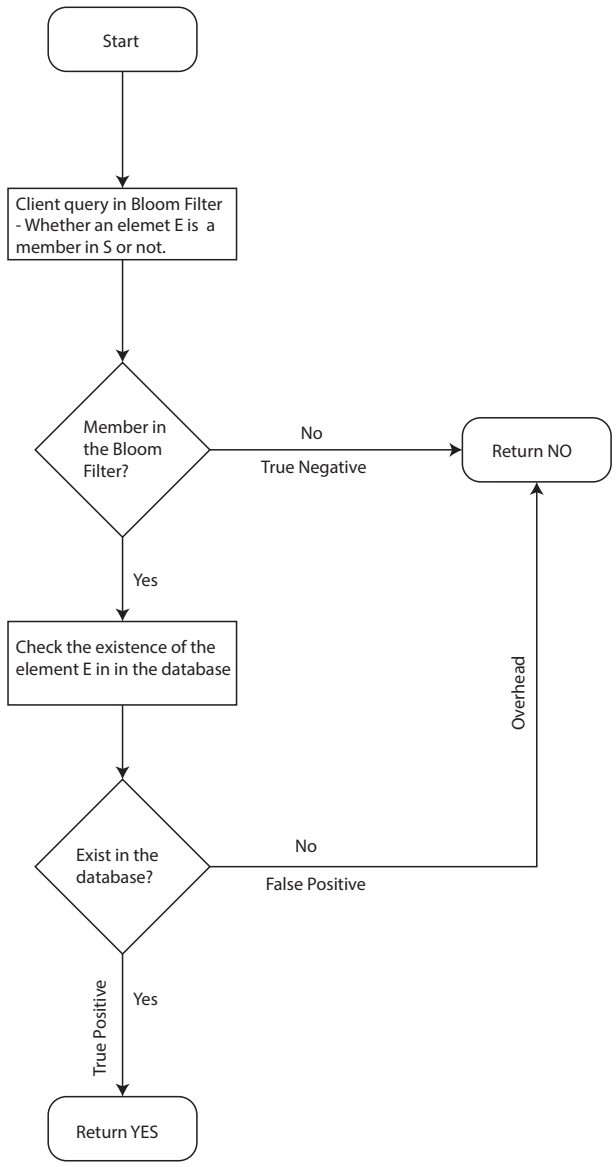

Fig. 1: Flowchart of Bloom Filter. Figure demonstrates the overhead of Bloom Filter

2.1 Analysis

The FP affects on performance of Bloom Filter and this is an overhead of a system as shown in Figure 1.

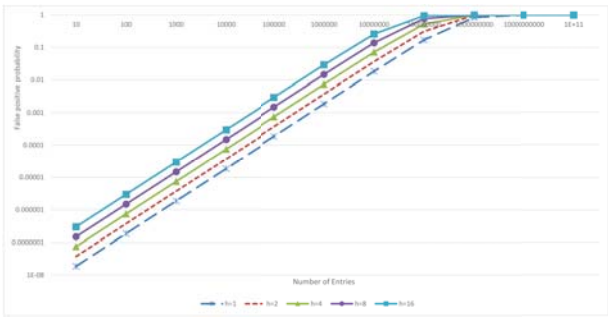

Fig. 2: The false positive probability of Bloom Filter with $m = 64MB$, and $h = 1,\ h = 2,\ h = 4,\ h = \ 8,\ and\ h = 16$. The X-axis represents the number of entries n.

However, almost all variants of the Bloom Filter reduce the FP probability. Let us assume, m is the number of bits available in the array. The probability of particular bit to be 1 is $\frac{1}{m}$. The probability of particular bit to be 0 is

$$\left(1 - \frac{1}{m}\right)$$

Let h be the number of hash functions and the probability of that bit remain 0 is [24], [23]

$$\left(1 - \frac{1}{m}\right)^h$$

There are total n element insertion into the array, therefore, the probability of that bit still 0 is

$$\left(1 - \frac{1}{m}\right)^{nh}$$

Now, the probability of that particular bit to be 1 is

$$\left(1 - \left(1 - \frac{1}{m}\right)^{nh}\right)$$

What is the optimal value of hashing h? The probability of all bits 1 is

$$\left(1 - \left(1 - \frac{1}{m}\right)^{nh}\right)^h \approx \left(1 - e^{-hn/m}\right)^h$$

The probability of false positive increases with the large size of entries n. However, it is reduced by increasing the value of m. Therefore, minimizing the false positive probability is

$$h = \frac{m}{n} ln2$$

Let us p be the desired false positive, and hence,

$$p = \left(1 - e^{-(\frac{m}{n} ln2n)/m}\right)^{(\frac{m}{n} ln2)}$$

$$ln\ p = -\frac{m}{n}(ln2)^2$$

$$m = -\frac{nln\ p}{(ln2)^2}$$

$$\frac{m}{n} = -\frac{n \log_2 p}{ln2} \approx -1.44 \log_2 p$$

Therefore, the optimal hash functions required

$$h = -1.44 \log_2 p$$

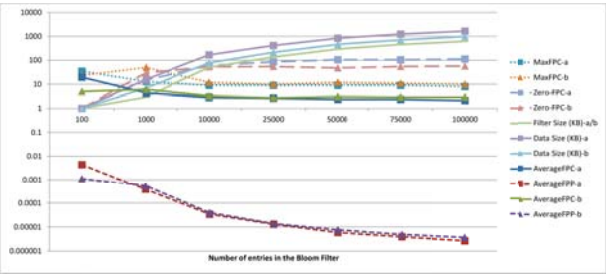

Fig. 3: Statistics on various data size during 1000 round queries.

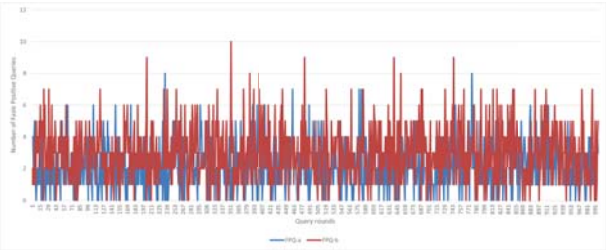

Fig. 4: False positive queries found on input size on 100000 elements during 1000 round queries. X-axis represent the number of query round and Y-axis represent the number of false positive queries.

Figure 3 depicts the experiments on various data size with 1000 round of queries [25]. We have generated random string dataset of various size and combination of strings dataset of various size. The Table 1 describes the parameters of Figure 3. Figure 3 represents the dynamic scaling of Filter Size according to data size. The MaxFPC refers to a maximum number of false positive detected in 1000 round queries. The zero FPC refers to total number of zero false positive count in 1000 round queries. The AverageFPC and AverageFPP are the mean false positive count and the mean false positive probability in 1000 round queries respectively.

Figure 4 depicts the snapshot by keeping the number input to 100000 elements [25]. The experiment is conducted by fixing the number input elements in random string and combination of alphabets. Those strings are input to study the behavior of a number of false positive queries hit in 1000 round queries. The dataset a and dataset b consist of random string and combination of the alphabets to form strings in different sizes. The input elements vary from 100 elements to 100000 elements to study the behavior of the false positive queries and the probability of false positive.

Table 1: Parameters description of Figure 3

Name	Description
a	Represents random strings dataset
b	Combination of strings dataset
MaxFPC	Maximum number of false positive count in 1000 round queries
Zero FPC	Total number of no false positive count (Not found FP) in 1000 round queries
AverageFPC	$\frac{Total\ FPC}{1000}$
AverageFPP	$\frac{Total\ FPP}{1000}$
Filter Size	The Bloom Filter array size.
Data Size	Total number of input entries.

2.2 Discarding Compressed Bloom Filter

The Compressed Bloom Filter (ComBF) [26] reduces the extra space requirements, and maps an element into a single bit. However, there is an important tradeoff between performance and space complexity. Therefore, the ComBF does not exhibit a good performance.

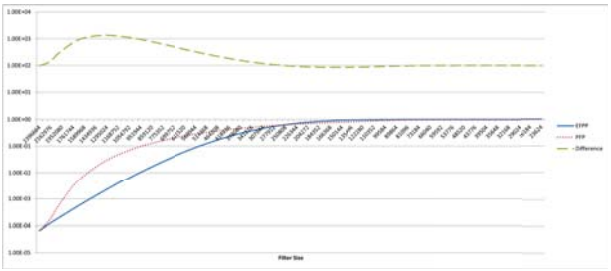

Fig. 5: Compression vs false positive on input of 100000 random string

Figure 5 exposes the trade off between the compression and false positive. The high compression results in increment of false positive probability. In Figure 5, the input strings are generated randomly and input to the compressed Bloom Filter. Hence, the result shows that high compression rate increases false positive which is not desirable for Bloom Filter. Figure 5 Effective False Positive Probability (EFPP), Probability of False Positive (PFP), and difference between both (Difference) [25]. The difference is calculated as follows

$$Differnce = 100 \times \frac{PFP}{EFPP}$$

3. Variants of Bloom Filter

3.1 Scalable Bloom Filter

Scalable Bloom Filter (SBF) [6] is a Bloom filter having a series of one or more Bloom Filters. In each Bloom filter, the array is partitioned into k slices. Each hash function produces one slice. During insertion operation, for each element k hash functions produces an index in their respective slice. So each element is described using k bits. When one Bloom filter is full another Bloom filter is added. During query operation, all filters are searched for the presence

of that element. The k bit description of each element makes this filter more robust where no element is especially sensitive to false positives. In addition, this Bloom Filter have the advantage of scalability by adapting to set growth by adding a series of classic Bloom filters and making the error probability more tighter as per requirement.

3.2 Adaptive Bloom Filter

Adaptive Bloom Filter (ABF) [27] is a Bloom Filter based on the Partial-Address Bloom Filter [28]. ABF is used in tracking the far-misses. Far-misses are those misses that is hits, if the core is permitted to use more cache. To each set of each core, a Bloom Filter array (BFA) with $2^(k)$ bits is added. When a tag is removed from the cache, tag's k least significant bit is used to index a bit of the BFA, which is 1. During Cache miss, using the k least significant bit the BFA is looked for the requested tag. A far-miss is detected when the array bit becomes 1.

3.3 Blocked Bloom filter

Blocked Bloom Filter [2] is a cache-efficient Bloom Filter. It is implemented by fitting a sequence of b standard Bloom Filter in each cache block/line. Usually for better performance, the bloom filters are made cache-line-aligned. When an element is added, the first hash function determines the Bloom filter block t0 use. The other hash functions are used for mapping of the element to k array slots, but within this block. Thus, this lead to only one cache miss. This is further improved by taking a single hash function instead of k hash functions. Hence, this single hash function determines the k slots. In addition, this hash operation is implemented using fewer SIMD instructions. The main disadvantage in using one hash function is, two elements are mapped to same k slots causes a collision. And this leads to increased false positive rate (FPR).

3.4 Dynamic Bloom Filter

Dynamic Bloom Filter (DBF) [29] is an extension of Bloom Filter which changes dynamically with changing cardinality. A DBF consist of some CBF (Counting Bloom Filter), say s. Initially s is 1 and the status of CBF as active. A CBF is called active when a new element is inserted or an element is deleted from it. During insertion operation, DBF first checks whether the active CBF is full. If it is full a new CBF is added and its' status is made active. If not, then new element is added to active CBF. During query operation, the response is given after searching all CBF. And, during deletion operation, first the CBF is found which contains the element. If a single CBF contains that element, then it is deleted. However, if multiple CBFs are there, then that deletion operation is ignored but deleted response (i.e. the operation is completed) is delivered. Furthermore, if the sum of two CBF capacities is less than a single CBF then they are united. For that, addition of counter vectors is done. The time complexity of insertion is same, whereas query and deletion operation is $O(k \times s)$ where k is the number of hash functions.

3.5 Deletable Bloom Filter

Deletable Bloom filter (DlBF) [30] is a Bloom Filter that enables false-negative-free deletions. In this Bloom Filter, the region of deletable bits is encoded compactly and saved in the filter memory. DlBF, divide the Bloom Filter array into some regions. This region is marked as deletable or non-deletable using bitmap of size same as the number of regions. During insertion operation, when an element maps to an existing element slot, i.e. collision, then the corresponding region is marked as non-deletable i.e bitmap is assigned value 1. This information is used during deletion. The elements under deletable region are only allowed to be deleted. Insertion and query operations in DIBF are same as the traditional Bloom Filter.

3.6 Index-Split Bloom Filters

Index-split Bloom filter (ISBF) [31] helps in reducing memory requirements and off-chip memory accesses. It consist of many groups of on-chip parallel CBFs and a list of off-chip items. When a set of items is stored, the index of each item is divided into some B groups. Each group contains b bits, where $B = \lceil log_2 n/b \rceil$. So the items are split into 2^b subsets. Each subset is represented by a CBF. Thus, total 2^b CBFs per group are constructed in on-chip memory. During query operation, after matching the query element and the index of an item found by the B group of on chip parallel CBFs, response is given. Also, for deletion operation, a lazy algorithm is followed. Because, deletion of an item requires adjustment of indexes of other off-chip items and reconstruction of all on-chip CBFs. Moreover, the average time complexity for off-chip memory accesses for insertion, query and, deletion is O(1).

3.7 Quotient filter

Quotient Filter (QF) [5] is a Bloom Filter where each element is represented by a multi-set F. The F is an open hashtable with a total buckets of m=2^q, called quotienting[32]. Besides, F stores p-bit fingerprint for each element which is the hash value. In this technique, a fingerprint f is partitioned into r least significant bits, which stores the remainder. The q=p-r is the most significant bits which stores the quotient. Both quotient and remainder is used for reconstruct of the full fingerprint. During insertion operation, F stores the hash value. During query operation, F is searched for the presence of the hash value of the element. And, during deletion operation, the hash value of that element is removed from F. QF has the advantage of dynamical resizing i.e. it expands and shrunk as elements are added or deleted. However, the QF insertion throughput deteriorates towards the maximum occupancy.

3.8 NameFilter

Name Filter [33] is a two-tier filter which is helps in looking up names in Named Data Networking. The first tier determines the length of the name prefix and second tier makes use of the prefix determined in the previous stage to make a search in a group of Bloom Filters. In the first stage the name prefixes are mapped to Bloom Filter. Thereafter, the process of building up a Counting Bloom Filter is taken up. This filter is built for the concerned prefix set and then it is converted to take the form of a conventional Boolean Bloom Filter. As a final step, the second stage use the merged Bloom Filter. In the first stage, the calibration of the name prefixes to the Bloom Filter is done on the basis of their lengths. It maps the k hash function into a single word. Hence, the Bloom Filter is called One Memory Access Bloom Filter as the query access time is $O(1)$ instead of $O(k)$. First, it acquires the hash output of the prefix using the DJB hash method. Then, the later hash value is calculated using the previous hash value. Thus, after $k-1$ loops, it obtains a single hash value and stores it in a word. This value is input for the calculation of the address in Bloom Filter, and the rest bits are calculated from one AND operation. So, when $k-1$ bits are 1s, then a graceful identification is declared. The aim of this stage is to find the longest prefix. In second stage, the prefixes are divided into groups based on their associated next-hop port(s). All groups are stored in the Bloom Filter. And, the desired port is found in this stage. In MBF, each slot stores a bit string with machine word-aligned. The Nth bit stores the Nth Bloom Filter's hash value and rest bits are padded with 0s. To obtain the forwarding port number, AND operation is done on K bit strings with respect to k hash functions. The location of 1 in the result gives the port number.

3.9 Cuckoo Filter

A Cuckoo Bloom Filter [3] is based on Cuckoo hash table [34]. This Bloom Filter stores fingerprint instead of key-value pairs. Whereas, fingerprint means the hash value of the element. For insertion, index for two candidate buckets are calculated. One is the hash value of the element and another is the XOR operation between the hash value of the element and the hash value of the fingerprint of that element. This is called partial-key cuckoo hashing. This method reduces hash collision and improves the table utilization. After finding the indexes, the element is stored in any free bucket. otherwise cuckoo hash tables' kicking [34] of elements is done. For query operation, two candidate buckets are calculated as done in insertion operation, then if the element is present in any one of them true is returned otherwise false. For deletion operation, same procedure as lookup is followed, whereas instead of returning true or false, element is deleted. The advantage of the basic algorithms (i.e. insertion,deletion and lookup) is they are independent of hash table configuration (e.g. number of entries in each bucket). However, the disadvantage of using partial-key cuckoo hashing for storing fingerprints leads to slow increase in fingerprint size to increase in filter size. In addition, if the hash table is very large, but stores short fingerprints then hash collision increases. This lead to the chances of insertion failure and also reduces the table occupancy.

3.10 Multi-dimensional Bloom Filter

Crainiceanu et. al. proposed a Bloom Filter called Bloofi [10]. Bloofi is a Bloom Filter index. It is implemented like a tree. The Bloom Filter tree construction is done as follows. The leaves are Bloom Filters. And, the bitwise OR on the leaf Bloom Filters is done to obtain the parent nodes. This process continues till root is obtained. During lookup operation, the element is checked at root if it does not match then it returns false. Because if an element in leaf does not match then it will not match from the leaf to the root. Whereas, if the element matches, the query further moves to roots' children Bloom Filters till it reaches the leaf. During insertion of a new node, search for most similar node to the new node is done. As Bloofi want to keep similar nodes together. So, when found this new node is inserted as its sibling. If an overflow occurs, then the same procedure is followed as in a B+ tree. During deletion operation, the parent node deletes the pointer to the node. And, when underflow occurs, the same procedure is followed as in B+ tree.

3.11 Sliding Bloom Filter

Sliding Bloom Filter [7] is a Bloom Filter having a sliding window. It has parameters (n, m, ε). The sliding window remains over last n elements and the value of the slots is 1. In other words, the window only shows the elements that are present. The m numbers of elements that appear before the window elements does not have restrictions on the value. And ε is the at most probable of slot being 1. This Bloom Filter is a dictionary based and use the Backyard Cuckoo hashing [35]. To this hashing a similar lazy deletion method is applied as used by Thorup [36]. A parameter c is used, which is the trade off between the accuracy of the index stored and the number of elements stored in the dictionary. After optimizing the parameter c the Sliding Bloom filter shows good time and space complexity. The algorithm uses a hash function selected from the family of Universal hash functions. For each element in the dictionary D, stores its hash value and location where it previously appeared. The stream of data is divided into generations of size n/c each, where c is optimized later. Generation 1 is the first n/c elements; generation 2 is next n/c elements and so on. Current window contains last n elements and at most c+1 generations. Two counters are used, one for generation number (say g) and another for the current element in the generation (say i). For every increment of i, g get incremented to mod (c+1). For insertion, first obtain the

ith hash value and checks whether it is present in D, if exists, the location of the element is updated with the current generation. Otherwise, it stores the hash value and generation number. Finally, update the two counters. If g changes, then scan D and delete all elements with associated data equal to the new value of g.

3.12 Bloom Filter Trie

Bloom Filter Trie (BFT) [37] helps to store and compress a set of colored k-mers, and efficient traversal of the graph. It is an implementation of the colored de Bruijn graph (C-DBG). It is based on burst trie which stores k-mers along with the set of colors. Colors are bit array initialized with 0. A slot assigns the value 1 if that index k-mer has that color. Later, this set of color is compressed. BFT is defined as $t = (V_t, E_t)$ having the maximum height as k where the k-mers is split into k substrings. A BFT is a list of compressed containers. An uncompressed container of a vexter V is defined as $< s, color_{ps} >$ where s is the suffix and p is the prefix which represents the path from root to V. Tuples are ordered lexicographically based on their suffixes. BST support operations for storing, traversing, and searching of a pan-genome. And, it also helps in extracting relevant information of the contained genomes and subsets. The time complexity for insertion of a k-mer is $O(d+2^{\lambda}+2q)$ where d is the worst lookup time, λ is the number of bits to represent the prefix and q is the maximum number of children. And, the time complexity of lookup operation is $O(2^{\lambda} + q)$.

3.13 Autoscaling Bloom Filter

Autoscaling Bloom Filter [38] is a generalization of CBF, which allows adjustment of its capacity based on probabilistic bounds on false positives and true positives. It is constructed by binarization of the CBF. The construction of Standard Bloom Filter is done by assigning all nonzero positions of the CBF as 1. And, given a CBF, the construction of ABF is done by assigning all the values which are less than or equal to the threshold value as 0.

3.14 d-left Counting Bloom filter

d-Left CBF (dlCBF) [4] is an improvement of the CBF. To implement this it uses the d-left hash table. This hash table consists of buckets, where each bucket has fixed number of cells. Each cell is of fixed size to hold a fingerprint and a counter. This arrangement makes the hash table appear as a big array. Each element has a fingerprint. And each fingerprint has two parts. The first part is a bucket index, which stores the element. Second part is the remainder part of the fingerprint. The range of bucket index is $[B]$ and the remainder is $[R]$. So the hash function is H: $U \rightarrow [B] \times [R]$. During element insertion, hash the element and store in appropriate remainders in the cell of each bucket. And increment the counter. And during deletion, decrement the counter. dlCBF solves the problems arise due to use of a single hash function. The hashing operation has two phases. In the first phase, apply a hash function, which gives the true fingerprint. And in the second phase, find the d locations of the element using additional (pseudo)-random permutation. One small disadvantage in the obtained d locations is, these are not independent, and uniform and as it is determined by the choices of the permutation.

3.15 Ternary Bloom Filter

Ternary Bloom Filter (TBF) [9] is another improvement of CBF. This Bloom filter introduces another parameter v for each hash value, which can have possible values as 0, 1, X. During insertion operation, if an element is mapped to a hash value for the first time, assign value 1 to v. If another element is mapped to the same hash value, then assign value X to v. During lookup operation, if an element's every v value for each hash value is X then it is defined as indeterminable. Indeterminable means, the element cannot be identified as negative or positive. And, value 1 indicates, the element is present and value 0 indicates the element is absent. Similarly, in deletion operation, if an elements' every v value for each hash value is X then it is defined as undeletable. Undeletable means, the element cannot be deleted from TBF. And, if v is value 1 it assigns value 0. TBF allocates the minimum number of bits to each cell which saves memory. In addition, it also gives much lower false positive rate compared to the CBF, when the same amount of memory used by both filters.

3.16 Difference Bloom Filter

Difference Bloom Filter (DBF) [13] is a probabilistic data structure based on Bloom Filter. It has multi-set membership query which is more accurate and has a faster response speed. It is based on two main design principles. First, to make the representation of the membership of elements exclusive by writing a different number of 0s and 1s in the same filter. Second, use of DRAM memory to increase the accuracy of the filter. DBF consist of a SRAM and a DRAM chaining hash table. The SRAM filter is an array of m bits with k independent hash functions. During the insertion function, elements in the set i are mapped to k bit of the filter. Arbitrarily $k - i + 1$ bits are set to value 1 and other $i - 1$ bits are set value 0. This is called $< i,\ k >$ constraint. If the new element gets conflicted with another element in the filter, DBF use dual-flip strategy to make this bit shared. Dual-flip is to change a series of mapping bits of the filters, so that the filter satisfy the $< i,\ k >$ constraint. During lookup operation, if exactly $k - i + 1$ bits are 1 then it returns true. During deletion operation, for each bit of the k bits of an element, DBF decides whether to reset it or not with the help of DRAM table.

3.17 Self-adjustable Bloom Filter

TinySet [8] is a Bloom Filter that has more space efficiency compared to standard Bloom Filter. Its' structure is

similar to the blocked Bloom filter. Whereas, each block is a chain based hash table [39]. It uses a single hash function, $H \rightarrow B \times L \times R$, where B is the block number, L is the index of the chain within that block, and R is the remainder (or fingerprint) that is stored in that block. All operations (insertion, deletion and lookup) initially follow three common steps. First, apply hash function is to the element and obtain the B, L, and R values. Second, use B to access the specific block. Third, calculate the Logical Chain Offset (LCO) and Actual Chain Offset (ACO) values. During insertion operation, shift all fingerprints from offset to the end of the block to right. The first bits in a block contain a fixed size index (I). Unset I means chain is empty. If the I bit is unset, it is made to set and the new element is marked as the last of its chain. During deletion operation, if I is unset, then the operation is terminated as it indicates the element is absent. Otherwise, shift all bits from the ACO value to end of the block to left by a single bit. If the deleted element is marked last then previous is made last or mark entire chain as empty. In lookup operation, if I is unset similarly the operation is terminated. Otherwise, search the chain. TinySet is more flexible due to its ability to dynamically change its configuration as per the actual load. It accesses only a single memory word and partially support deletion of elements. However, delete operation gradually degrade its space efficiency over time.

3.18 Multi-stage Bloom Filter

BloomFlow [12] is a multi-stage Bloom Filter which is used for multicasting in Software-defined networking (SDN). It helps to achieve reductions in forwarding state while avoiding false positive packet delivery. The BloomFlow extends the OpenFlow [11] Forward action with a new virtual port called BLOOM_PORTS to implement Bloom filter forwarding. When a flow specifies an output action to BLOOM_PORTS forwarding Element (FE) implements an algorithm. The algorithm first reads from the start of the IP option field, the Elias gamma encoded filter length b, and the number of hash function of k fields. Then the algorithm treats the rest of the bits of the IP option field as a Bloom Filter. And this Bloom Filter is copied to a temporary cache for further processing. The remainder of IP options fields and the IP payload are shifted left to remove first stage filter from the packet header. Then the algorithm iterates through all interfaces and check for membership test for each interface's bloom identifier in the cached bloom filter. Bloom identifier is a unique, 16 bit integer identifier. The Bloom identifier is assigned by the network controller to every interface on the network that participates in multicast forwarding. If the membership test returns true, the packet is forwarded from the matched interface.

3.19 Dynamic Reordering Bloom Filter

Dynamic Reordering Bloom Filter [14] is another type of Bloom Filter that saves the searching cost of Bloom Filter. It dynamically reorders the searching sequence of multiple Bloom Filter using One Memory Access Bloom Filter (OMABF) and the order of checking is saved in Query Index (QI). This approach considers two factors. First, policy of changing the query priority of Bloom Filter. Second, reduction of overhead caused due to change in the order. This approach reduces the searching time of the query by sorting and saving the query data in Bloom Filter based on popularity. Sorting is done based on the query order, i.e popularity of data. So when the request comes from that data it quickly gives the response. And, when the popularity of a data becomes more, its query order is made a level higher in the Bloom Filter. However, this change of query order imposes overheads. To solve this, Query Index (QI) is used. QI saves the query priority of each block. When membership is checked Bloom Filter are checked according to the order saved in QI.

4. Conclusion

The Bloom Filter is the widely used data structure. The Bloom Filter also associates with a system to improve the performance dramatically. Moreover, it does not waste more spaces of main memory. The Bloom Filter provides a fast lookup system with a few KB of memory spaces. The Bloom Filter returns either 0 (False) or 1 (True). However, this Boolean value is classified into four categories, namely, TN, TP, FP, and FN. The TN and TP boost up the lookup performance of a system. On the contrary, the FP, and FN become an overhead to the system. Nevertheless, the FN is not common for all variants of Bloom Filter. The FP is the key barrier of Bloom Filter. Therefore, there are several kinds of Bloom Filters in the market. The key objective of the modern Bloom Filter is to reduce the probability of FP. In addition, the modern Bloom Filter also deals with high scalability, space efficiency, adaptability, and high accuracy. Besides, the Bloom Filter meets copious applications, and thus, extensive experiment has been done on Bloom Filter. The paper discusses a few selected applications to highlight the efficacy of the Bloom Filter. However, it is observed that the Bloom Filter is applied extensively in computer networking. Moreover, the efficiency, and accuracy of Bloom Filter depends on the probability of false positive. Therefore, reducing the false positive probability is a prominent challenge to achieve. Finally, the Bloom Filter will be able to reduce the false positive probability approximately to zero.

In this paper, we presented the theoretical and practical analysis of Bloom Filter. Moreover, there are abundant of Bloom Filter variants, those are discussed in this paper. Furthermore, issue and challenges of Bloom Filter are discussed. Also, we have exposed the disadvantages of compressed

bloom filter through a experiment. Moreover, the FP analysis is also shown through an experiment.

References

[1] B. H. Bloom, "Space/time trade-o s in hash coding with allowable errors," *Comm. of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[2] F. Putze, P. Sanders, and J. Singler, "Cache-, hash-, and space-efficient bloom filters," *J. Exp. Algorithmics*, vol. 14, pp. 4:4.4–4:4.18, Jan. 2010.

[3] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM Intl. Conf. on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14, 2014, pp. 75–88.

[4] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, *An Improved Construction for Counting Bloom Filters*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 684–695.

[5] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok, "Don'T Thrash: How to cache your hash on flash," *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1627–1637, July 2012.

[6] P. S. Almeida, C. Baquero, N. Preguica, and D. Hutchison, "Scalable bloom filters," *Information Processing Letters*, vol. 101, no. 6, pp. 255–261, 2007.

[7] M. Naor and E. Yogev, "Tight bounds for sliding bloom filters," *Algorithmica*, vol. 73, no. 4, pp. 652–672, 2015.

[8] G. Einziger and R. Friedman, "Tinyset- an access efficient self adjusting bloom filter construction," *IEEE/ACM Transactions on Networking*, vol. 25, no. 4, pp. 2295–2307, 2017.

[9] H. Lim, J. Lee, H. Byun, and C. Yim, "Ternary bloom filter replacing counting bloom filter," *IEEE Communications Letters*, vol. 21, no. 2, pp. 278–281, 2017.

[10] A. Crainiceanu and D. Lemire, "Bloofi: Multidimensional bloom filters," *Information Systems*, vol. 54, no. Supplement C, pp. 311 – 324, 2015.

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[12] A. Craig, B. Nandy, I. Lambadaris, and P. Koutsakis, "Bloomflow: Openflow extensions for memory efficient, scalable multicast with multi-stage bloom filters," *Computer Communications*, vol. 110, no. Supplement C, pp. 83 – 102, 2017.

[13] D. Yang, D. Tian, J. Gong, S. Gao, T. Yang, and X. Li, "Difference bloom filter: A probabilistic structure for multi-set membership query," in *2017 IEEE International Conference on Communications (ICC)*, 2017, pp. 1–6.

[14] D. C. Chang, C. Chen, and M. Thanavel, "Dynamic reordering bloom filter," in *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2017, pp. 288–291.

[15] 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.

[16] R. Anitha and S. Mukherjee, "'maas': Fast retrieval of data in cloud using metadata as a service," *Arabian Journal for Science and Engineering*, vol. 40, no. 8, pp. 2323–2343, 2015.

[17] Y. Zhu, H. Jiang, and J. Wang, "Hierarchical bloom filter arrays (hba): A novel, scalable metadata management system for large cluster-based storage," in *CLUSTER '04, Proceedings of the 2004 IEEE International Conference on Cluster Computing*, 2004, pp. 165–174.

[18] Y. Zhu, H. Jiang, J. Wang, and F. Xian, "Hba: Distributed metadata management for large cluster-based storage systems," *IEEE transactions on parallel and distributed systems*, vol. 19, no. 6, pp. 750 – 763, 2008.

[19] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, "Supporting scalable and adaptive metadata management in ultralarge-scale file systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 4, pp. 580 – 593, 2011.

[20] D. Zhu and M. Mutka, "Sharing presence information and message notification in an ad hoc network," in *Pervasive Computing and Communications, 2003.(PerCom 2003). Proceedings of the First IEEE International Conference on*. IEEE, 2003, pp. 351–358.

[21] L. Maccari, R. Fantacci, P. Neira, and R. M. Gasca, "Mesh network firewalling with bloom filters," in *Communications, 2007. ICC'07. IEEE International Conference on*. IEEE, 2007, pp. 1546–1551.

[22] G. Fernandez-Del-Carpio, D. Larrabeiti, and M. Uruena, "Forwarding of multicast packets with hybrid methods based on bloom filters and shared trees in mpls networks," in *2017 IEEE 18th International Conference on High Performance Switching and Routing (HPSR)*, 2017, pp. 1–8.

[23] F. Grandi, "On the analysis of bloom filters," *Information Processing Letters*, vol. 129, no. Supplement C, pp. 35 – 39, 2018.

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

[25] A. Partow, "C++ bloom filter library," Accessed on December, 02, 2017 from http://www.partow.net/programming/bloomfilter/index.html and https://github.com/ArashPartow/bloom.

[26] M. Mitzenmacher, "Compressed bloom filters," *IEEE/ACM Transactions on Networking*, vol. 10, no. 5, pp. 604–612, 2002.

[27] K. Nikas, M. Horsnell, and J. Garside, "An adaptive bloom filter cache partitioning scheme for multicore architectures," in *2008 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2008, pp. 25–32.

[28] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai, "Bloom filtering cache misses for accurate data speculation and prefetching," in *ACM International Conference on Supercomputing 25th Anniversary Volume*. ACM, 2014, pp. 347–356.

[29] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The dynamic bloom filters," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 1, pp. 120–133, 2010.

[30] C. E. Rothenberg, C. A. B. Macapuna, F. L. Verdi, and M. F. Magalhaes, "The deletable bloom filter: a new member of the bloom family," *IEEE Communications Letters*, vol. 14, no. 6, pp. 557–559, 2010.

[31] K. Huang and D. Zhang, "An index-split bloom filter for deep packet inspection," *Science China Information Sciences*, vol. 54, no. 1, pp. 23–37, Jan 2011.

[32] D. E. Knuth, *The art of computer programming: sorting and searching*. Pearson Education, 1998, vol. 3.

[33] Y. Wang, T. Pan, Z. Mi, H. Dai, X. Guo, T. Zhang, B. Liu, and Q. Dong, "Namefilter: Achieving fast name lookup with low memory cost via applying two-stage bloom filters," in *2013 Proceedings IEEE INFOCOM*, 2013, pp. 95–99.

[34] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[35] Y. Arbitman, M. Naor, and G. Segev, "Backyard cuckoo hashing: Constant worst-case operations with a succinct representation," in *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, 2010, pp. 787–796.

[36] M. Thorup, "Timeouts with time-reversed linear probing," in *Infocom, 2011 Proceedings Ieee*. IEEE, 2011, pp. 166–170.

[37] G. Holley, R. Wittler, and J. Stoye, "Bloom filter trie: an alignment-free and reference-free data structure for pan-genome storage," *Algorithms for Molecular Biology*, vol. 11, 2016.

[38] D. Kleyko, A. Rahimi, and E. Osipov, "Autoscaling bloom filter: Controlling trade-off between true and false positives," *CoRR*, vol. abs/1705.03934, 2017. [Online]. Available: http://arxiv.org/abs/1705.03934

[39] R. L. Rivest and C. E. Leiserson, *Introduction to algorithms*. McGraw-Hill, Inc., 1990.