ISSN No. 0976-5697

Volume 1, No. 3, Sept-Oct 2010

International Journal of Advanced Research in Computer Science

REVIEW ARTICLE

Available Online at www.ijarcs.info

Bloom Filters: A Review

Arulanand Natarajan*
Anna University
Coimbatore, India
line 2: name of organization, acronyms acceptable
arulnat@yahoo.com

K. Premalatha
Department of CSE
Bannari Amman Institute of Technology
Erode, India
kpl_barath@yahoo.co.in

S. Subramanian
Sri Krishna College of Engineering and Technology
Coimbatore, TN, India
dsraju49@gmail.com

Abstract: This paper presents different representations and applications of Bloom filter. A Bloom filter is a simple but powerful data structure that can check membership to a static set. Bloom filters become more popular for networking system applications, spell-checkers, string matching algorithms, network packet analysis tools and network/internet caches and database optimization. This paper will examine and analyze different types of bloom filter and its applications.

Keywords: Bloom Filter, Data Structure, Counting Bloom Filter, Dynamic Bloom Filter, Anomaly Detection

I. INTRODUCTION

Bloom filters are compact data structures for probabilistic representation of a set to support membership queries. Its core concept is associative containers. Given a string X the Bloom filter computes k hash functions on it producing k hash values ranging from 1 to m. It then sets k bits in an m-bit long vector at the addresses corresponding to the k hash values. The same procedure is repeated for all the members of the set. This process is called programming of the filter. The query process is similar to programming, where a string whose membership is to be verified is input to the filter. The Bloom filter generates k hash values using the same hash functions it used to program the filter. The bits in the m-bit long vector at the locations corresponding to the k hash values are looked up. If at least one of these k bits is not found in the set then the string is declared to be a nonmember of the set. If all the bits are found to be set then the string is said to belong to the set with a certain probability. This uncertainty in the membership comes from the fact that those k bits in the m-bit vector can be set by any other n-1 members. Thus finding a bit set does not necessarily imply that it was set by the particular string being queried. However, finding a bit not set certainly implies that the string does not belong to the set.

The cost of this compact representation is a small probability of false positives: the structure sometimes incorrectly recognizes an element as member of the set, but often this is a convenient trade-off. Bloom filters were developed in the 1970's [2] and have been used in database applications to store large amounts of static data [20]. Bloom's motivation was to reduce the time it took to lookup data from a slow storage device to faster main memory and hence could dramatically improve the performance.

A Bloom filter program consists of a set of hash functions, a hash function buffer to store hash results temporarily, a look up array to signify hash values and a decision component made of an AND to test the membership of testing string as shown in figure 1.

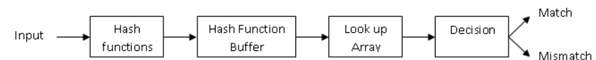

Figure 1. Bloom Filter for Membership Testing

The requirement of designing k different independent hash functions may be prohibitive for large k. For a good hash function with a wide output, there should be little if any correlation between different bit-fields of such a hash, so this type of hash can be used to generate multiple different hash functions by slicing its output into multiple bit fields. Alternatively, pass k different initial values (such as 0, 1, ... ,k-1) to a hash function that takes an initial value; or add these values to the key. For larger m and/or k, independence among the hash functions can be relaxed with negligible increase in false positive rate [10][16]. Specifically, Ref. [11] shows the effectiveness of using enhanced double hashing or triple hashing, variants of double hashing, to derive the k indices using simple arithmetic on two or three indices computed with independent hash functions. The paper deals with the types of bloom filter and applications of bloom filter. Section II gives types of the Bloom filter. The applications of bloom filter are explained in Section III. Section IV provides the summary of Bloom Filter.

II. TYPES OF BLOOM FILTER

A. *Standard Bloom Filter (SBF)*

An empty Bloom filter is a bit array of m bits, all set to 0. There must also be k different hash functions defined, each of which maps or hashes some set element to one of the m array positions with a uniform random distribution. To add an element, each of the k hash functions is feed to get k array positions. The bits at all these positions are set to 1.

To query for an element, each of the k hash functions is feed to get k array positions. If any of the bits at these positions are 0, the element is not in the set. Otherwise all the bits would have been set to 1 when it was inserted. If all are 1, then either the element is in the set, or the bits have been set to 1 during the insertion of other elements. Figure 2 shows the SBF.

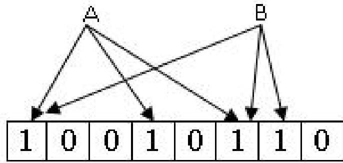

Figure 2 Standard Bloom Filter

Removing an element from this SBF is not possible. The element maps to k bits, and although setting any one of these k bits to zero suffices to remove it. This has the side effect of removing any other elements that map onto that bit, and there is no way of determining whether any such elements have been added. Such removal may introduce a possibility for false negatives, which are not allowed.

Removal of an element from a Bloom filter can be simulated by having a second Bloom filter that contains items that have been removed. However, false positives in the second filter become false negatives in the composite filter, which are not permitted. This approach also limits the semantics of removal since re-adding a previously removed item is not possible. However, it is often the case that all the keys are available but are cumbersome to enumerate. When the false positive rate gets too high, the filter can be regenerated; this should be a relatively rare event.

B. *Counting Bloom Filter (CBF)*

The set of elements is changing over time in that case the insertions and deletions in the Bloom filter become important. Inserting elements into a Bloom filter hash the element k times and set the bits to 1. However, deletion process is hashing the element to be deleted k times and set the corresponding bits to 0, is not possible. This is because setting a location to 0 that is hashed to by some other element in the set, and the resultant Bloom filter is no longer correctly reflects all elements in the set. To avoid this problem, counting Bloom filter was introduced as an extension to Bloom filters [7]. Here, each entry in the Bloom filter is not a single bit but instead a small counter. When an item is inserted, the corresponding counters are incremented; and when an item is deleted, the corresponding counters are decremented. Figure 3 shows an example Counting Bloom Filter. Large counters can be used to avoid counter overflow. The analysis from Ref. [12] brings out that 4 bits per counter should be sufficient for most applications.

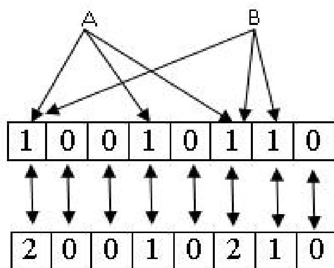

Figure 3 Count Bloom Filter

C. *Hierarchical Counting Bloom Filter (HCBF)*

The data structure of HCBF (Yuan et al., 2008) *is* composed of several sub CBFs. The number of these sub filters is h. Each sub filter has different counter length and bit array length. Each counter length is c_0, c_1… c_{h-1} and each bit array length is m_0, m_1, …, m_{h-1} respectively. $m_0 > m_1 > \ldots > m_{h-1}$. HCBF data structure is shown in Figure 4.

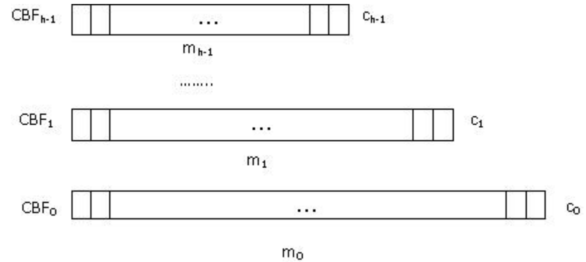

Figure 4 Hierarchical Counting Bloom Filter

The element frequency is represented by these multi-layer sub filters. The bit array length of the high layer sub filter is short and can only store a few elements; its false negative rate might be a bit higher. If all sub filters must be queried in getting the element frequency, error rate will be very high. Therefore, the system adopts a Level Counter to store the element maximal level. It query the filters whose level are less than the maximal level, so that the query complexity and error rate can be reduced

D. *Spectral Bloom Filter*

The Spectral Bloom filter [6] is an extension of the SBF to multi-sets, allowing the filtering of elements whose multiplicities are below a threshold given at query time. The spectral Bloom filter replaces the bit vector with a vector of m counters C. the counters in C roughly represent multiplicities of items, all the counter in C are initially set to 0. When inserting an item, it increases the counters $C_{h1(s)}$, $C_{h2(s)}$, …, $C_{hk(s)}$ by 1 and it stores the frequency of each item. It allows deletion by decreasing the same counters.

E. *Split Bloom Filter*

The one type of bloom filter is Split Bloom filters [24]. It increases the capacity by allocating a fixed s×m bit matrix instead of an m-bit vector as used by the SBF to represent a set. A certain number of s filters each with m bits, are employed and uniformly selected when inserting an item of the set. The false match probability increases as the set cardinality grows. The basic idea is, in element addition operation, before going to map element *x* into the standard bloom filter s, it first checks the bloom filters from 1 to s-1 whether they have response that element *x* is a member of set *A*. If the response is false, it makes sure that there is no false positive probability in first s-1 bloom filters, so it maps the element *x* into bloom filter s; otherwise, it just go ahead to the next element with no any operation on element *x*.

The scheme is multi dimension dynamic Bloom filters to support concise representation and approximate membership queries of dynamic sets [13]. The basic idea is to add a new bit array when all the previous arrays are full. It starts with one bit array (i.e., one Bloom filter). Afterward, multiple bit arrays form a bit matrix of Bloom filters.

© 2010, IJARCS All Rights Reserved

F. *Weighted Bloom Filter*

Ref. [3] generalized the traditional Bloom filter to Weighted Bloom Filter, which incorporates the information on the query frequencies and the membership likelihood of the elements into its optimal design. In many applications, some popular elements are queried much more often than the others. Weighted Bloom filter derives the optimal configuration of the Bloom filter with query-frequency and membership likelihood information.

In Weighted Bloom filter's false positive probability is a weighted sum of each individual element's false positive probability, where the weight corresponding to an individual element is positively correlated with the element's query frequency and is negatively correlated with the element's probability of being a member. So it needs to assign more hash functions to an element with a higher query frequency or with a lower probability of being a member, in order to reduce the false-positive probability of an element with a higher weight.

G. *Dynamic Bloom Filter (DBF)*

The DBF can support concise representation and approximate membership queries of dynamic set instead of static set. The basic idea of DBF is to represent a dynamic set with a dynamic s×m bit matrix that consists of s bloom filters. Here s is initialized to 1, but it is not a constant as split bloom filter. It can increase during the continuous increasing process of the set size.

H. *Improved Dynamic Bloom Filter (i-DBF)*

The DBF has a disadvantage such as the addition operation which mapped element x into bloom filter s will become no sense, if some of the first s-1 bloom filters have already responded that element x is in set A with some false positive probability. To check an element a_m is a member of set it checks whether all the $h_i(a_m)$ are set to 1 for any one bloom filter used by the DBF one by one where $1 \leq i \leq k$; If the response is false then a_m is not a member of A; Otherwise, a_m is in *A* with some false positive probability.

In i-DBF [22] in element addition operation, map element *x* into the standard bloom filter s. It first checks the bloom filters from 1 to s-1 whether they have response that element *x* is a member of set A. If the response is false then there is no false positive probability in first s-1 bloom filters, so it maps the element *x* into bloom filter s; otherwise, we just go ahead to the next element with no any operation on element *x*.

I. *Matrix Bloom Filter (MBF)*

For representing dynamic set DBF and split bloom filter Split Bloom Filter have been developed. Both DBF and Split Bloom Filter can support representation and membership queries of dynamic set instead of static set. Split Bloom Filter declares that it uses an s×m bit matrix that consists of s bloom filters to represent a dynamic set. Both the bloom filters are not matrix representation method at all. They are just a set of s bloom filters whose length is m. Ref. [23] introduced a matrix representation method of bloom filter to represent a dynamic set.

The MBF representation is a dynamic set A with an s×m bit matrix which has s rows and m columns. Here s is a constant and must be predefined according to the estimation of the maximum value of set size, may be based on increasing history record, or some other factors. For constructing a MBF, it checks the value of m and the error rate of each row according to the application needs and it calculates the number of hash functions k and the max number of elements that each row can contain. It confirms which row the element should be added. It used the first hash function h_1 to do this job. h_1 is a special hash function which differs from the others. Because there are s rows in MBF, h_1 is with the range {1,…,s} the row among 1 to s. It uses h_1 to determine the row with the formula row = h_1(ele). In this row, the hash functions h_2 to h_{k+1} to map the elements. For $2 \leq i \leq k+1$, set the bit h_i(ele) to 1.

J. *Scalable Bloom Filter*

The Scalable Bloom filter [19] represents dynamic data sets well and provides a way to effectively solve the scalability problem of Bloom filters. It solves the scalability problem of Bloom filters by adding Bloom filter vectors with double length when necessary. Initially a SBF_0 {n,m,k} is given, and the tolerant false positive rate f_0 is assumed. It calculates the maximum number of elements n_0 to keep $f \leq f_0$. The data set is expanded, when $n>n_0$, a new SBF_1 is added to the SBF with vector length $m_1=2\times m$. When $n>3n_0$, another new SBF_2 is added to the SBF with vector length $m_2 = 4\times m$. When the SBF extends i times, i.e., $n> (2i-1)n_0$, a new SBFi is appended with vector length $m_i = 2_i\times m$. The scalable Bloom filter also is expected to find many new and widespread applications in the future.

III. APPLICATIONS OF BLOOM FILTER

A. *Compressed Bloom Filter for Message Passing*

Compressing Bloom filter improves performance when the Bloom filter is passed as a message between nodes, particularly when information must be transmitted repeatedly, and its transmission size is a limiting factor. For example, Bloom filters have been suggested as a means for sharing Web cache information. In this setting, proxies do not share the exact contents of their caches, but instead periodically broadcast Bloom filters representing their cache. However, to choose an optimal value for k to minimize the false probability the value of $p = 1/2$. Under this assumption of independent random hash functions, the bit array is essentially a random string of 0's and 1's, with each entry being 0 or 1 with probability 1/2. It would therefore seem that no gain in compression when sending such Bloom filters. On the other hand, large sparse Bloom Filters can be greatly compressed [12]. An m-bit filter can be compressed to mH(p) bits where p is the probability that a bit in the filter is 0 and H(p) is the entropy function. Hence, by using such compressed Bloom filters, proxies can reduce the number of bits broadcast, the false positive rate, and/or the amount of computation per lookup. The cost is the processing time for compression and decompression, which usually uses simple arithmetic coding, and more memory use at the proxies, which utilizes the larger sparser array of uncompressed form of the Bloom filter.

B. *Multi-level Bloom Filter for XML*

Traditional Bloom filters can be extended to be used on hierarchical documents. Ref. [29] introduced extensions to Bloom filters based on two alternative ways of hashing XML trees to support path expressions. They are Breadth Bloom Filter and Depth Bloom Filter.

The Breadth Bloom Filter (BBF) for an XML tree T with j levels is a set of Bloom filters $\{BBF_0, BBF_1, BBF_2, \ldots, BBF_i\}$, $i \leq j$. There is one Bloom filter, denoted BBF_i, for each level i

© 2010, IJARCS All Rights Reserved

of the tree. In each BBF_i, it inserts the elements of all nodes at level i. To improve performance, it constructs an additional Bloom filter denoted BBF_0. In this Bloom filter, it inserts all elements that appear in any node of the tree.

The procedure that checks whether a BBF matches a query distinguishes between path queries starting from the root and partial path queries. In both cases, it checks whether all elements in the query appear in BBF_0. Only if there is a match for all elements, it proceeds by examining the structure of the path. For a root query every level i from 1 to p of the filter is checked for the corresponding a_i. The algorithm succeeds, if it has a match for all elements. For a partial path query, for every level i of the filter, the first element of the path is checked. If there is a match, the next level is checked for the next element and the procedure continues until either the whole path is matched or there is a miss. If there is a miss, the procedure repeats for level i + 1. For paths with the ancestor descendant axis, and the sub-paths are processed. All matches are stored and compared to determine whether there is a match for the whole path.

In Depth Bloom Filter (DBF) different Bloom filters are used to hash paths of different lengths. The Depth Bloom Filter (DBF) for an XML tree T with j levels is a set of Bloom filters $\{DBF_0, DBF_1, DBF_2, \ldots, DBF_{i-1}\}$, There is one Bloom filter, denoted DBF_i, for each path of the tree with length i (i.e., a path of i + 1 nodes), it inserts all paths of length i. Regarding the size of the filters, as opposed to BBF, all DBF is have the same size, since the number of paths of different lengths is of the same order. The procedure, that checks whether a DBF matches a path query, first checks whether all elements in the path expression appear in DBF_0. If this is the case, it continues treating both root and partial paths queries the same. For a query of length p, every sub-path of the query from length 2 to p is checked at the corresponding level. If any of the sub-paths does not exist, the algorithm returns a miss. For paths that include the ancestor descendant axis and the resulting sub-paths are checked. If we have a match for all sub-paths the algorithm succeeds, else we have a miss.

C. Time-Decaying Bloom Filters (TBF) for Data Streams with Skewed Distributions

To enable queries for multiplicities of multi-sets, the bitmap in a Bloom Filter as replaced by an array of counters whose values increment on each occurrence. In a data stream model, however, data items arrive at varying rates and recent occurrences are regarded as more significant than past ones. In most data stream applications, it is critical to handle time-sensitivity. In addition, data streams with skewed distributions are common in many emerging applications, engineering and billing, intrusion detection, trudging surveillance and outlier detection. For applications, it is to allocate counters of uniform size to all buckets. In TBF [5], it maintains the frequency count for each item in a data stream, and the value of each counter decays with time. For data streams with highly skewed distributions, it allows dynamically allocating free counters to the large items.

D. Bloom Filter for Network Anomaly Detection

Ref. [9] presented a hardware-based technique using Bloom filters, which can detect strings in streaming data without degrading network throughput. They group signatures according to their length (in bytes) and store each group of string in a unique Bloom filter. An analyzer is employed to resolve false positives. They have also proposed a technique for reducing packet inspection time by using parallel Bloom filters.

Ref. [1] proposed a space-efficient method to follow and detect signatures that are fragmented over multiple packets. They used a data structure called prefix Bloom filters and a heuristic, chain heuristic to make the filters space-efficient. A fault in Bloom filters, however, may cause false negatives to occur. For a string already programmed in a Bloom filter, a faulty hashing unit might generate an incorrect location (i.e., a hash value) at which 0 is stored instead of 1, resulting in a false negative. For a given fault, the probability that false negatives will occur is high unless some provisions are made to detect and eliminate them.

Ref. [12] proposed Bloom Filter array for Network Anomaly Detection. It applied 2D matching feature to network anomaly detection. A counter is introduced for insertion-removal pair vector to support counting and removal operation in a Bloom filter. A sliding window is designed to reduce the false alarm probability caused by the boundary effect due to discrete-time sampling. A random-keyed hash functions are designed, which provide both security and convenient extension of Bloom filter. It is applied for network anomaly detection, more specifically, feature extraction for network anomaly detection.

Ref. [30] presented a hardware-based fault-tolerant Bloom filter which detects and eliminates false-negatives during normal operation. It is based on property checking of a Bloom filter with some extra hardware circuits. The design is simple to implement with negligible overhead. As a result, packets may proceed at line speed, regardless of the added circuits.

E. Longest Prefix Matching (LPM) using Bloom Filter

Due to the growth of the Internet, Classless Inter-Domain Routing (CIDR) was widely adopted to prolong the life of Internet Protocol Version 4 (IPv4). CIDR requires Internet routers to search variable-length address prefixes in order to find the longest matching prefix of the IP destination address and retrieve the corresponding forwarding information for each packet traversing the router. This computationally intensive task, commonly referred to as IP Lookup, is often the performance bottleneck in high-performance Internet routers.

Ref. [8] proposed LPM using Bloom filter by sorting the forwarding table entries by prefix length, associating a Bloom filter with each unique prefix length, and programming each Bloom filter with prefixes of its associated length. A search begins by performing parallel membership queries to the Bloom filters by using the appropriate segments of the input IP address. The result of this step is a vector of matching prefix lengths, some of which may be false matches. Hash tables corresponding to each prefix length are probed in the order of longest match in the vector to shortest match in the vector, terminating when a match is found or all of the lengths represented in the vector are searched. The performance is determined by the number of dependent memory accesses per lookup, can be held constant for longer address lengths or additional unique address prefix lengths in the forwarding table given that memory resources scale linearly with the number of prefixes in the forwarding table.

F. Mining frequent Items using Bloom Filter based on Damped model (MIBFD)

Ref. [22] presented for finding the frequent items in the data stream based on Damped window model. The Damped model, also called the Time-Fading model, mines frequent

© 2010, IJARCS All Rights Reserved

items in stream data in which each transaction has a weight and this weight decreases with age. Older transactions contribute less weight toward items frequencies. This model is suitable for applications in which old data has an effect on the mining results, but the effect decreases as time goes on. MIBDF used Extensible and Scalable Bloom Fitler (ESBF). ESBF is made up of a series of EBFs (Extensible Bloom Filter). EBF is Counting Bloom Filter whose counters can be extended to larger ones and the number of the filters in it is scalable. EBF is made up of following three parts: two kinds of counters, Basic Counters (BC) with length of x bits and Large Counters (LC) with length of 2x bits, and a bit vector (OF) indicating whether a BC counter has become overflow, the length of the OF is equal to the number of the BC counters. Initially, there are only BC counters and OF in the EBF, and the values of BC counters and all bits of OF are set to 0. When any of the BC counters become overflow, a LC counter is created, and the value of the overflow counter is turned into a pointer pointing to the newly created LC counter, which is used for counting the corresponding item, which means, the filter is extensible when the current EBF in the ESBF gets full due to the limit on its capacity, a new EBF is created and is added to the ESBF for the newly monitored items arriving in the stream. The size of newly added filter is M_0s^i, where M0 is the size of the first filter, i is the current number of the filters. MIBFD is efficient both in processing time and in memory usage and does not require the priori knowledge about the data stream to be processed.

G. Bloom Filter for Time-Dependent Multi Bit-Strings for Incremental Set

Ref. [25] proposed a Time-dependent Multiple bit-strings Bloom Filter (TMBF) which roots in the DBF and targets on dynamic incremental set. TMBF uses multiple bit-strings in time order to present a dynamic increasing set and uses backward searching to test whether an element is in a set.

TMBF uses a set of fixed size bloom filter bit-string to represent S(t). Each bit-string binds with a time stamp. The search algorithm tests whether the element is in the bit strings one by one with backward time order. During adding an element x, if the number of element reaches the upper bound of a bit string (n0), a new bit string will be created. All bits in the new bit string set to 0. And all the bits at those position calculated by k hash functions set to 1. The false positive rate of TMBF is $1-(1-(1-e^{-kn_0/Ls})^k)^s$. Where L is the length of the Bit String, n_0 is the number of maximum elements represented by a bit string, k is the number of hash functions and s is the number of bit strings. TMBF reduces the storage space and network communication cost.

H. Dynamic Bloom Filter (DBF) for Membership Queries of Dynamic Set

The SBF just focus on how to represent a static set and decrease the false positive probability to a sufficiently low level and CBF focus to avoid false negative. By investigating mainstream applications based on the Bloom filter, the dynamic data sets are more common and important than static sets. The existing variants of the Bloom filter cannot support dynamic data sets well. To address this issue, Guo et al (2010) proposed dynamic Bloom filters to represent dynamic sets, as well as static sets and design necessary item insertion, membership query, item deletion, and filter union algorithms. The dynamic Bloom filter can control the false positive probability at a low level by expanding its capacity as the set cardinality increases. A Dynamic Bloom Filter consists of s homogeneous SBFs. The initial value of s is one, and the initial SBF is active. The DBF only inserts items of a set into the active SBF, and appends a new SBF as an active SBF when the previous active SBF becomes full. The first step to implement a DBF is initializing the following parameters: the upper bound on false match probability of the DBF, the largest value of s, the upper bound on false match probability of the SBF, the filter size m of the SBF, the capacity c of the SBF, and number of hash functions k of the SBF.

I. Aging Bloom Filter with two Active Buffers

To support dynamic sets, Bloom filters need delete operation to make space for new incoming data. For this purpose, the simplest scheme would be cold cache [4]. Once the Bloom filter is full, all data are removed. Then, the next arriving data are programmed. The problem is that no data remain in the memory whenever delete occurs. This causes a load spike that is not tolerable in certain applications like real-time systems. The alternative choice would be the CBF. Using counters rather than bits, the CBF can delete a previously inserted data. However, this cannot delete stale data selectively unless the list of old data to delete is maintained in a separate space. In CBF each cell is of multiple bits, which ranges from 0 to a certain maximum value. When a new element is programmed, the corresponding cells are set to the maximum value. Like a Bloom filter, the false positive ratio is related to the ratio of zero bits. To keep the false positive ratio below the threshold, some cells are selected and their values are decreased to keep the number of zero bits larger than a certain value. However, this approach has some limitations. First, if the maximum value of a cell is large, the memory space is wasted. Note that two-bit cells decrease the number of cells by half. We stress that the number of zero cells should be large enough to keep the false positive ratio below the threshold. Second, if a cell is randomly selected and decreased to zero, this may fail the membership checking of more than one element. Yoon (2010) proposed a double buffering scheme to support the selective deletion of old data from Bloom filters in first-in-first-out. In this scheme, the memory space is divided into two independent groups: active cache and warm-up cache. The size of each cache is $\frac{m}{2}$ and the allowed false positive ratio of one cache should equal to f. The active cache stores all the recent data and the warm-up cache is always a subset of the active cache.

Some network applications require high-speed processing of packets. For this purpose, Bloom filters should reside in a fast and small memory, SRAM. In this case, due to the limited memory size, stale data in the Bloom filter should be deleted to make space for new data. Namely the Bloom filter needs aging like LRU caching. The new aging scheme for Bloom filters is, active-active buffering. They assumed that m, the total memory size, and f, the allowed false positive ratio, are already fixed.

They proposed A2 buffering to make Bloom filters age smoothly for dynamic sets. With this scheme, it is possible to update Bloom filters with recently used data. Consequently, Bloom filters can be useful for dynamic sets as well as static sets.

© 2010, IJARCS All Rights Reserved

J. Two-level Bloom Filter (TBF) for Object-based Storage Network

Ref. [15] proposed TBF for meta data server in Object-based Storage Network. TBF is composed of attribute and object Bloom filters to support the efficient representation and query of objects. The attribute Bloom filter is a series of counter-based arrays. Each array can store one attribute of an object. The object Bloom filter captures a verification value of an object, which can reflect the inherent dependency of all attributes for the object. When a query request arrives, it first checks whether the attributes of an object exist in the first level. If the attributes are in the attribute BFs, it checks whether the valid attributes belong to one object based on the verification values, which have been stored in the object BF. When the queries in the two-level architecture receive answers True, it determines that the queried object exists in the metadata server.

IV. SUMMARY

The benefit of Bloom filters is that they provide a tradeoff between the memory requirement and the false positive ratio. Bloom filter based solution is used to protect the list of members or to defend to publish a directory of everyone, because it doesn't contain the list of keys in a discoverable format. The formulas also don't rely on the size of the keys that is storing in the filter. Whether it is a short user names, absolute URIs to the feeds, or even the text of entire books into the bloom filter and the calculations would remain the same. Bloom Filters pay for this is in the amount of time spend calculating the hash of a key, which would be much longer for a book over a username. This review shows the numerous applications where such a data structure is required. Particularly when space is a concern, a Bloom filter may be an excellent alternative for keeping an explicit list. The main drawback of using a Bloom filter is that it allows false positives. Their consequence must be carefully considered for each specific application to determine whether the impact of false positives is acceptable. Bloom filter is space saving and easily parallelizable. Bloom filters are applied in Collaborating in overlay and peer-peer networks, Resource routing, Packet routing, IP traceback, Proxy cache, Dictionaries, Databases and Rule mining.

V. REFERENCES

[1] Artan N S and Chao H J., "Multi-packet signature detection using prefix bloom filters", Proceedings of IEEE Conference on Global Telecommunications, vol.3, 2005, p.6.

[2] Bloom B, "Space/time tradeoffs in hash coding with allowable errors", Communications of the ACM, Vol.13, No.7, pp. 422–426, 1970.

[3] Bruck J., Gao J and Jiang A., "Weighted Bloom Filter", IEEE International Symposium on Information Theory, pp.2304-2308, 2006,.

[4] Chang F., Feng W and Li K., "Approximate caches for packet classification", Proceedings of Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies, vol.4, pp.2196-2207, 2004.

[5] Cheng K., Xiang L., Iwaihara M and Sangyo K., "Time-decaying Bloom Filters for data streams with skewed distributions", 15th International Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications, pp.63-69, 2005.

[6] Cohen S and Matias Y, "Spectral bloom filters", Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pp. 241-252, 2003.

[7] Czerwinski S., Zhao B. Y., Hodes T., Joseph A. D, and R. Katz, An "Architecture for a secure service discovery service" In Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking(MobiCom '99), pp. 24—35. New York: ACM Press, 1999.

[8] Dharmapurikar S., Krishnamurthy P and Taylor D E., "Longest prefix matching using bloom filters", IEEE/ACM Transactions on Networking, vol.14, no.2, pp.397-409, 2006.

[9] Dharmapurikar S., Lockwood T S and J.W., "Deep packet inspection using parallel bloom filters", IEEE Computer Society, vol.24, no.1, pp.52-61, 2004.

[10] Dillinger, Peter C. Manolios, Panagiotis "Fast and accurate bitstate verification for SPIN", Proceedings of the 11th International Spin Workshop on Model Checking Software, Springer-Verlag, Lecture Notes in Computer Science 2989 , 2004a

[11] Dillinger, Peter C. Manolios, Panagiotis, "Bloom Filters in probabilistic verification", Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design, Springer-Verlag, Lecture Notes in Computer Science 3312, 2004b

[12] Fan J., Wu D., Lu k and Nucci A., NIS04-3: Design of Bloom Filter Array for Network Anomaly Detection, Proceedings of IEEE Conference on Global Telecommunications, pp.1-5, 2006.

[13] Guo D., Wu J., Chen H and Luo X., "Theory and network applications of dynamic bloom filters", Proceedings of 25th IEEE International Conference on Computer Communications, pp.1-12, 2006.

[14] Guo D., Wu J., Chen H., Yuan Y and Luo X., "The Dynamic Bloom Filters", IEEE Transactions on Knowledge and Data Engineering, vol.22, no. 1, pp.120-133, 2010.

[15] Hua U., Feng D and Xiao B., "TBF: an efficient data architecture for metadata server in the object-based storage network", Proceedings of 14th IEEE International Conference on Networks, vol.1, pp.1-6, 2006.

[16] Kirsch, Adam, Mitzenmacher, Michael, "Less hashing, same performance: building a better bloom filter, algorithms", 14th Annual European Symposium, Springer-Verlag, Lecture Notes in Computer Science 4168, pp. 456–467, 2006.

[17] Lee M H and Choi Y H., "A fault-tolerant bloom filter for deep packet inspection", 13th Pacific Rim International Symposium on Dependable Computing, pp.389-396, 2007.

[18] Mitzenmacher M., "Compressed Bloom filters", IEEE/ACM Transactions on Networking, vol.5, no.5, pp.604-612, 2002.

[19] Kun Xie, Yinghua Min, Dafang Zhang, Jigang Wen and Gaogang Xie. "scalable bloom filter for membership

© 2010, IJARCS All Rights Reserved

queries". Proceedings of the 50th Annual IEEE Globe Communications Conference (IEEE Globecom 2007) . Washington DC, USA, 2007.

[20] Mullin j.k., and Margoliash D. J, "A tale of three spelling checkers", Software — Practice and Experience Vol. 20, no. 6, pp. 625—630, 1990.

[21] Wang J., Xiao M and Dai Y., "MBF: a real matrix bloom filter representation method on dynamic set", Proceedings of IFIP International Conference on Network and Parallel Computing Workshops, Liaoning , pp. 733 – 736, 2007.

[22] Wang J., Xiao M., Jiang J and Min B., "i-DBF: an improved bloom filter representation method on dynamic set", Proceedings of Fifth International Conference on Grid and Cooperative Computing Workshops, Hunan, 2006.

[23] Wang S., Hao X., Xu H and Hu Y., "Mining frequent items based on bloom filter", Proceedings of Fourth International Conference on Fuzzy Systems and Knowledge Discovery, vol.4, pp.679-683, 2007.

[24] Xiao M., Dai Y and Li X., "Split Bloom Filter", Chinese Journal of Electronic, vol.32, no.2, pp.241-245, 2004.

[25] Xiao M., Kong X., Liu J and Ning J., "TMBF: Bloom filter algorithms of time-dependent multi bit-strings for incremental set", Proceedings of International Conference on Ultra Modern Telecommunications & Workshops, pp.1-4, 2009.

[26] Xie K., Min Y., Zhang D., Wen J and Xie G., "A Scalable bloom filter for membership queries", Proceedings of IEEE Conference on Global Telecommunications, Washington, DC, pp. 543 – 547, 2007.

[27] Yoon M., "Aging bloom filter with two active buffers for dynamic sets", IEEE Transactions on Knowledge and Data Engineering, vol.22, no.1, pp.1134-138, 2010.

[28] Yuan Z., Chen Y., Jia Y and Yang S., "Counting evolving data stream based on hierarchical counting bloom filter", Proceedings of International Conference on Computational Intelligence and Security, 2008.

[29] Koloniari G and Pitoura E, "Filters for XML-based service discovery in pervasive computing", Computer Journal: Special Issue on Mobile and Pervasive Computing, vol.47, pp.16-27, 2004.

[30] Myeong-Hyeon Lee, Yoon-Hwa Choi, "A fault-tolerant bloom filter for deep packet inspection" 13th Pacific Rim International Symposium on Dependable Computing pp.389-396, 2007.

© 2010, IJARCS All Rights Reserved