BLOOM FILTERS: AN INTRODUCTION

Shrivatsa D Perur[1]

[1] *Assistant Professor, Dept. of ISE, GIT, Belagavi, Karnataka, India*

---***---

Abstract—*many network solutions and overlay networks utilize probabilistic techniques to reduce information processing and cost of networking. This article presents a number of frequently used and useful probabilistic techniques. Bloom filters and their variants are of prime importance, and they are heavily used in various distributed systems. This has been reflected in recent research and many new algorithms have been proposed for distributed systems that are either directly or indirectly based on Bloom filters. To keep false positive probabilities low, the size of the bloom filter must be dimensioned a priori to be linear in the maximum number of keys inserted, with the linearity constant ranging typically from one to few bytes.*

Key words*-Bloom filters, probabilistic structures, distributed systems*

I.INTRODUCTION

The bloom filter is a bit-vector data structure that provides a compact representation of a set of elements (keys). It supports insertion of elements and membership queries. A membership answer is probabilistically correct in the sense that it allows a small probability of a false positive (i.e., an incorrect answer for a non-member element). The bloom filter allows tradeoffs between small size (compactness) and low false positives (accuracy). To keep false positives low, the size of the bloom filter must be dimensioned a priori to be linear in the maximum number of keys inserted, with the linearity constant typically ranging from one to few bytes. Fast matching of arbitrary identifiers to values is a basic requirement for a large number of applications. Data objects are typically referenced using locally or globally unique identifiers. Recently, many distributed systems have been developed using probabilistic globally unique random bit strings as node identifiers. For example, a node tracks a large number of peers that advertise files or parts of files. Fast mapping from host identifiers to object identifiers and vice versa are needed. The number of these identifiers in memory may be great, which motivates the development of fast and compact matching algorithms. Given that there are millions or even billions of data elements, developing efficient solutions for storing, updating, and querying them becomes increasingly important. The key idea behind the data structures discussed in this survey is that by allowing the representation of the set of elements to lose some information, in other words to become lossy, the storage requirements can be significantly reduced. Bloom in 1970. Bloom first described a compact probabilistic data structure that was used to represent words in a dictionary. There was little interest in using Bloom filters for networking until 1995, after which this area has gained widespread interest both in academia and in the industry. A bloom filter is simply used to test whether the element is present in the set or not.
Its main properties are:

1. The amount of space needed to store the bloom filter is very less when compared to the amount of data belonging to the set being tested.

2. The time needed to check whether an element is a member of a given set is independent of the number of elements contained in the set.

3. False negatives are not possible.

4. False positives are possible, but their frequency can be controlled. In practice, it is a trade off between space/time efficiency and the false positive frequency.

II. BLOOM FILTER

Whenever a list or set is used, and space is at a premium, consider using a Bloom filters if the effect of false positives can be mitigated. A Bloom filters is an array of m bits for representing a set $S = \{x_1, x_2 \ . \ . \ . \ x_n\}$ of n elements. Initially all the bits in the filters are set to zero. The key idea is to use k hash functions, hi(x), $1 \le i \le k$ to map items $x \in S$ to random numbers uniform in the range 1, . . .m. The hash functions are assumed to be uniform. The MD5 hash algorithm$_{\in}$ is a popular choice for the hash functions. An element x S is inserted into the filters by setting the bits hi(x) to one for $1 \le i \le k$. Conversely, y is assumed a member of S if the bits hi(y) are set, and guaranteed not to be a member if any bit hi(y) is not set. The weak point of Bloom filters is the possibility for a false positive. False positives are elements that are not part of S but are reported being in the set by the filters.

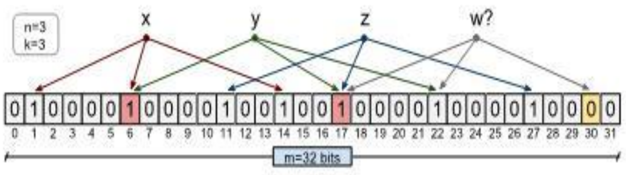

Fig 1. Overview of Bloom Filters

The bloom filter utilizes the hashing technique for the search of best document. The bloom filter gets the Query from the node, it performs multiple hashing in the query and as a result it converts the query into URLs. A BF is a loss but succinct and efficient data structure to represent a set S, which can efficiently process the membership query such as "is element x in set S."

III.HASHING TECHNIQUES

Here I briefly present hashing techniques that are the basis for good Bloom filter implementations. I start with perfect hashing, which is an alternative to Bloom filters when the set is known beforehand and it is static. Double hashing allows reducing the number of true hash computations. Partitioned hashing and multiple hashing deal with how bits are allocated in a Bloom filter. Finally, the use of simple hash functions is considered.

A. Perfect Hashing Scheme: A simple technique called perfect hashing (or explicit hashing) can be used to store a static set S of values in an optimal manner using a perfect hash function.

A perfect hash function is a computable bijection from S to an array of |S| = n hash buckets. The n-size array can be $_\epsilon$used to store the information associated with each element x S [1]. Bloom filter like functionality can be obtained by, given a set of elements S, first finding a perfect hash function P and then storing at each location an $f = 1/\varrho$ bit fingerprint, computed using some (pseudo-)random hash function H.

B. Double Hashing: The improvement of the double hashing technique over basic hashing is being able to generate k hash values based on only two universal hash functions as base generators (or "seed" hashes). As a practical consequence, Bloom filters can be built with less hashing operations without crificing performance. Kirsch and Mitzenmacher have shown [2] that it requires only two independent hash functions, h1(x) and h2(x), to generate additional "pseudo" hashes defined as:

$$h_i(x) = h1(x) + f(i) * h2(x) \qquad (10)$$

where i is the hash value index, f(i) can be any arbitrary function of i (e.g., i^2), and x is the element being hashed. For Bloom filter operations, the double hashing scheme reduces the number of true hash computations from k down to two without any increase in the asymptotic false positive probability [2].

C. Partitioned Hashing: In this hashing technique, the k hash functions are allocated disjoint ranges of m/k consecutive bits instead of the full m-bit array space , probability of a specific bit being 0 in a partitioned Bloom filter can be approximated to:

$$(1 - k/m)^n \approx e^{-kn/m}$$

While the asymptotic performance remains the same, in practice, partitioned Bloom filters exhibit a poorer false positive performance as they tend to have larger fill factors (more 1s) due to the m/k bit range restriction. This can be explained by the observation that: Lookup of x simply consists of computing P(x) and checking whether$_\epsilon$ the stored hash function value matches H(x). When x S, the correct value is always returned, and when x do not belong to S a false positive (claiming the element being in S) occurs with probability at most ϱ. This follows from the definition of 2- universal hashing by Carter and Wengman [3], that any element y not in S has probability at most ϱ of having the same hash function value h(y) as the element in S that maps to the same entry of the array. While space efficient, this approach is disconsidered for dynamic environments, because the perfect hash function.

D. Multiple Hashing: Multiple hashing is a popular technique that exploits the notion of having multiple hash choices and having the power to choose the most convenient candidate. When applied for hash table constructions, multiple hashing provides a probabilistic method to limit the effects of collisions by allocating elements more-or-less evenly distributed. The original idea was proposed by Azar et al. in his seminal work on balanced allocations [4]. Formulating hashing as a balls into bins problem, the authors show that if n balls are placed sequentially into m for m = O(n) with each ball being placed in one of a constant d = 2 randomly chosen.

E. Simple Hash Functions: A common assumption is to consider output hash values as truly random, that is, each hashed element is independently mapped to a uniform location. While this is a great aid to theoretical analyses, hash function implementations are known to behave far worse than truly random ones. On the other hand, empirical works using standard universal hashing have been reporting negligible

differences in practical performance compared to predictions assuming ideal hashing (see [6] for the case of Bloom filters).

IV. BLOOM FILTER VARIANTS

A number of Bloom filter variants have been proposed that address some of the limitations of the original structure, including counting, deletion, multisets, and space-efficiency. We take up few variants here.

A. Compressed Bloom Filter:

Compressing a Bloom filter improves performance when a Bloom filter is passed in a message between distributed nodes. This structure is particularly useful when information must be transmitted repeatedly, and the bandwidth is a limiting factor[7]. Compressed Bloom filters are used only for optimizing the transmission (over the network) size of the filters. This is motivated by applications such as Web caches and P2P information sharing, which frequently use Bloom filters to distribute routing tables. The key idea in compressed Bloom filters is that by changing the way bits are distributed in the filter, it can be compressed for transmission purposes. This is achieved by choosing the number of hash functions k in such a way that the entries in the m vector have a smaller probability than ½ of being set.

B. Spectral Bloom Filters:

Spectral Bloom filters generalize Bloom filters to storing an approximate multiset and support frequency queries [8]. The membership query is generalized to a query on the multiplicity of an element. The answer to any multiplicity query is never smaller than the true multiplicity, and greater only with probability ǫ. In this sense, spectral refers to the range within which multiplicity answers are given. The space usage is similar to that of a Bloom filter for a set of the same size (including the counters to store the frequency values). The time needed to determine a multiplicity of k is O(logk).The query time is $\Theta(\log(1/\epsilon))$. The answer estimate is given by returning the minimum value of the k counters determined by the hash functions.

C. Space Code Bloom Filter:

Per-flow traffic measurement is crucial for usage accounting, traffic engineering, and anomaly detection. Previous methodologies are either based on random sampling (e.g., Cisco's NetFlow), which is inaccurate, or only account for the ”elephants”. A data structure called Space Code Bloom Filter (SCBF) can be used to measure per-flow traffic approximately at high speeds. SCBF employs a Maximum Likelihood Estimation (MLE) method to measure the multiplicity of an element in the multiset.

D. Decaying Bloom Filter:

The Decaying Bloom Filter (DBF) structure has been proposed for this application scenario. DBF is an extension of the counting Bloom filter and it supports the removal of stale elements from the structure as new elements are inserted. DBF may produce false positive errors, but not false negatives as is the case with the basic Bloom filter. A variant of DBF has been applied for hint-based routing in wireless sensor networks [9]. This motivates approximate detection of duplicates among newly arrived data elements of a data stream. This can be accomplished within a fixed time window.

V. BLOOM FILTERS IN DISTRIBUTED COMPUTING

We have surveyed techniques for probabilistic representation of sets and functions. The applications of these structures are many fold, and they are widely used in various networking systems, such as Web proxies and caches, database servers, and routers.

A. Caching

Bloom filters have been applied extensively to caching in distributed environments. To take an early example, Fan, Cao, Almeida, and Broader proposed the Summary Cache [10], [11] system, which uses Bloom filters for the distribution of Web cache information. The system consists of cooperative proxies that store and exchange summary cache data structures, essentially Bloom filters. When a local cache miss happens, the proxy in question will try to find out if another proxy has a copy of the Web resource using the summary cache. If another proxy has a copy, then the request is forwarded there. In order for distributed proxy-based caching to work well, the proxies need to have a way to compactly summarize available content. In the Summary Cache system, proxies periodically transfer the Bloom filters that represent the cache contents (URL lists).

Google's Bigtable system that is used by many massively popular Google services, such as Google Maps and Google Earth, and Web indexing. Bigtable is a distributed storage system for structured data that has been designed with high scalability requirements in mind, for example capability to store and query petabytes of data across thousands of commodity servers [12]. A Bigtable is a sparse multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp. Each value in the map is an uninterpreted array of bytes. Bigtable uses Bloom filters to reduce the disk lookups for non-existent rows or columns [12]. As a result the query performance of the database has to rely less on costly disk operations and thus performance increases.

B. P2P Networks

Bloom filters have been extensively applied in P2P environments for various tasks, such as compactly storing keyword- based searches and indices [13], synchronizing sets over network, and summarizing content. The exchange of keyword lists and other metadata between peers is crucial for P2P networks. Ideally, the state should be such that it allows for accurate matching of queries and takes sublinear space (or near constant space). The later versions of the Gnutella protocol use Bloom filters [14] to represent the keyword lists in an efficient manner. In Gnutella, each leaf node sends its keyword Bloom filter to an ultra-node, which can then produce a summary of all the filters from its leaves, and then sends it to neighbouring ultra-nodes. The ultra-nodes are hubs of connectivity, each being connected to more than 32 other ultra node

D. Monitoring and Measurement

Network monitoring and measurement are key application areas for Bloom filters and their variants. We briefly examine some key cases in this domain, for example detection of heavy flows, Iceberg queries, packet attribution, and approximate state machines.

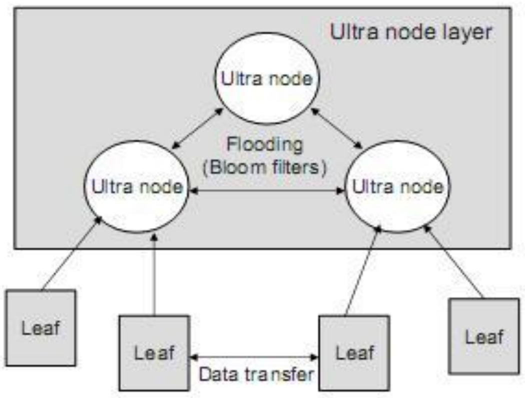

Fig:2-tier gnutella architecture

A. Heavy Flows: Bloom filters have found many applications in measurement of network traffic. One particular application is the detection of heavy flows in a router. Heavy flows can be detected with a relatively small amount of space and small number of operations per packet by hashing incoming packets into a variant of the counting Bloom filter and incrementing the counter at each set bit with the size of the packet. Then if the minimum counter exceeds some threshold value, the flow is marked as a heavy flow [15].

B. Iceberg Queries: An Iceberg query is such that identifies all items with frequency above some given threshold. Bloom filter variants that are able to count elements are good candidate structures for supporting Iceberg queries. In networking, low-memory approximate histogram structures are needed for collecting network statistics at runtime. For example, in some applications it is necessary to track flows across domains and perform, to name a few examples, congestion and security monitoring. Iceberg queries can be used to detect Denial-of-Service attacks.

VI. BLOOM FILTERS FOR SIMILARITY TESTING

Observe that we can view each document to be a set in Bloom filter parlance whose elements are the CDCs that it is composed of. Given that Bloom filters compactly represent a set, they can also be used to approximately match two sets. Bloom filters, however, cannot be used for exact matching as they have a finite false-match probability but they are naturally suited for similarity matching. For finding similar documents, we compare the Bloom filter of one with that of the other. In case the two documents share a large number of 1's (bit-wise AND) they are marked as similar. In this case, the bit-wise AND can also be perceived as the dot product of the two bit vectors. If the set bits in the Bloom filter of a document are a complete subset of that of another filter then it is highly probable that the document is included in the other. Web pages are typically composed of fragments, either static ones (e.g., logo images), or dynamic (e.g., personalized product promotions, local weather) [16]. When targeting pages for a similarity based "grouping", the test for similarity should be on the fragment of interest and not the entire page.

Bloom filters, when applied to similarity detection, have several advantages. First, the compactness of Bloom filters is very attractive for storage and transmission whenever we want to minimize the meta-data overheads. Second, Bloom filters enable fast comparison as matching is a bitwise-AND operation. Third, since Bloom filters are a complete representation of a set rather than a deterministic sample (e.g., shingling), they can determine inclusions effectively.

VII. SUMMARY

Bloom filters are a general aid for network processing and improving the performance and scalability of distributed systems. The space required by the bloom filter is very less when compared to the size of the data in the element set. Compressed bloom filters are used to optimize the data to be transmitted in the distributed system. This automatically increases the performance of the system in distributed system. The bloom filters can be used for the large data to know whether the particular element is present in the set or not. Per flow traffic can be measured easily in the heavy traffic by the usage of space code bloom filters . the spectral bloom filters generalize bloom filters to storing an approximate multiset and support frequency queries. Spectral refers to the range within which multiplicity answers are given. Decaying bloom filter supports for the removal of stale elements from the structure as new elements are inserted. It may produce false positive errors but not false negatives as in the case of the basic bloom filters.

REFERENCES

[1] A. Pagh, R. Pagh, And S. S. Rao, "An Optimal Bloom FilterReplacement," In Soda '05: Proceedings Of The Sixteenth Annual Acm-Siam Symposium On Discrete Algorithms. Philadelphia, Pa, Usa: Society For Industrial And Applied Mathematics, 2005, Pp. 823–829.

[2] A. Kirsch And M. Mitzenmacher, "Less Hashing, Same Performance:Building A Better Bloom Filter," In Esa'06:Proceedings Of The 14th Annual European Symposium On Algorithms. London, Uk: Springer Verlag, 2006, Pp. 456–467.

[3] J. L. Carter And M. N. Wegman, "Universal Classes Of Hash Functions(Extended Abstract)," In Stoc '77: Proceedings Of The Ninth Annual Acm Symposium On Theory Of Computing. New York, Ny, Usa: Acm, 1977, Pp. 106–112. Y. Azar, A. Z. Broder, A. R. Karlin, And E. Upfal, "Balanced Allocations," Siam J. Comput., Vol. 29, No.1, Pp. 180–200, 2000.

[4] B. V¨ Ocking, "How Asymmetry Helps Load Balancing," J. Acm, Vol. 50, No. 4, Pp. 568–589, 2003.

[5] M. Mitzenmacher, "Compressed Bloom Filters," In Podc '01: Proceedings Of The Twentieth Annual Acm Symposium On Principles Of Distributed Computing. New York, Ny, Usa: Acm, 2001, Pp. 144–150.

[6] L. Fan, P. Cao, J. Almeida, And A. Z. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," Sigcomm Comput. Commun. Rev., Vol. 28, No. 4, Pp. 254–265, 1998.

[7]"Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," Ieee/Acm Trans. Netw., Vol. 8, No. 3, Pp. 281–293, 2000.

[8]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," In Osdi '06: Proceedings Of The 7th Usenix Symposium On Operating Systems Design And Implementation. Berkeley, Ca, Usa: Usenix Association, 2006, Pp.15–15.

[9] J. Risson And T. Moors, "Survey Of Research Towards Robust Peer-To-Peer Networks: Search Methods," Comput. Netw., Vol. 50, No. 17, Pp. 3485–3521, 2006.

[10] H. Cai, P. Ge, And J. Wang, "Applications Of Bloom Filters In Peer- To-Peer Systems: Issues And Questions," In Nas '08: Proceedings Of The 2008 International Conference On Networking, Architecture, And Storage. Washington, Dc, Usa: Ieee Computer Society, 2008, Pp.97– 103.

[11] W.-C. Feng, K. G. Shin, D. D. Kandlur, And D. Saha, "The Blue Active Queue Management Algorithms," Ieee/Acm Trans. Netw., Vol. 10,No. 4, Pp. 513–528, 2002.

[12] A. Z. Broder And A. R. Karlin, "Multilevel Adaptive Hashing," In Soda'90: Proceedings Of The First Annual Acm-Siam Symposium On Discrete Algorithms. Philadelphia, Pa, Usa: Society For Industrial And Applied Mathematics, 1990, Pp. 43–53.