Bloom Filters – Short Tutorial

Matei Ripeanu, Adriana Iamnitchi

1. Introduction

Bloom filters [2] are compact data structures for probabilistic representation of a set in order to support membership queries (i.e. queries that ask: "Is element *X* in set *Y*?"). This compact representation is the payoff for allowing a small rate of *false positives* in membership queries; that is, queries might incorrectly recognize an element as member of the set.

We succinctly present Bloom filters use to date in the next section. In Section 3 we describe Bloom filters in detail, and in Section 4 we give a hopefully precise picture of space/computing time/error rate tradeoffs.

2. Usage

Since their introduction in [2], Bloom filters have seen various uses:

- *Web cache sharing* ([3]) Collaborating Web caches use Bloom filters (dubbed "cache summaries") as compact representations for the local set of cached files. Each cache periodically broadcasts its summary to all other members of the distributed cache. Using all summaries received, a cache node has a (partially outdated, partially wrong) global image about the set of files stored in the aggregated cache. The Squid Web Proxy Cache [1] uses "Cache Digests" based on a similar idea.
- *Query filtering and routing* ([4, 6, 7]) The Secure wide-area Discovery Service [6], subsystem of Ninja project [5], organizes service providers in a hierarchy. Bloom filters are used as summaries for the set of services offered by a node. Summaries are sent upwards in the hierarchy and aggregated. A query is a description for a specific service, also represented as a Bloom filter. Thus, when a member node of the hierarchy generates/receives a query, it has enough information at hand to decide where to forward the query: downward, to one of its descendants (if a solution to the query is present in the filter for the corresponding node), or upward, toward its parent (otherwise).

  The *OceanStore* [7] replica location service uses a two-tiered approach: first it initiates an inexpensive, probabilistic search (based on Bloom filters, similar to Ninja) to try and find a replica. If this fails, the search falls-back on (expensive) deterministic algorithm (based on Plaxton replica location algorithm). Alas, their description of the probabilistic search algorithm is laconic. (An unpublished text [11] from members of the same group gives some more details. But this does not seem to work well when resources are dynamic.)
- *Compact representation of a differential file* ([9]). A differential file contains a batch of database records to be updated. For performance reasons the database is updated only periodically (i.e., midnight) or when the differential file grows above a certain threshold. However, in order to preserve integrity, each reference/query to the database has to access the differential file to see if a particular record is scheduled to be updated. To speed-up this process, with little

memory and computational overhead, the differential file is represented as a Bloom filter.

- *Free text searching* ([10]). Basically, the set of words that appear in a text is succinctly represented using a Bloom filter

3. Constructing Bloom Filters

Consider a set $A = \{a_1, a_2, ..., a_n\}$ of n elements. Bloom filters describe membership information of *A* using a bit vector *V* of length *m*. For this, *k* hash functions, $h_1, h_2, ..., h_k$ with $h_i : X \to \{1..m\}$, are used as described below:

The following procedure builds an *m* bits Bloom filter, corresponding to a set A and using $h_1, h_2, ..., h_k$ hash functions:

```
Procedure BloomFilter(set A, hash_functions, integer m)
            returns filter
    filter = allocate $m$ bits initialized to 0
    foreach $a_i$ in $A$:
        foreach hash function $h_j$:
            filter[$h_j(a_i)$] = 1
        end foreach
    end foreach
    return filter
```

Therefore, if a_i is member of a set *A*, in the resulting Bloom filter *V* all bits obtained corresponding to the hashed values of a_i are set to 1. Testing for membership of an element *elm* is equivalent to testing that all corresponding bits of *V* are set:

```
Procedure MembershipTest (elm, filter, hash_functions)
            returns yes/no
    foreach hash function $h_j$:
        if filter[$h_j(elm)$] != 1 return No
    end foreach
    return Yes
```

Nice features: filters can be built incrementally: as new elements are added to a set the corresponding positions are computed through the hash functions and bits are set in the filter. Moreover, the filter expressing the reunion of two sets is simply computed as the bit-wise OR applied over the two corresponding Bloom filters.

4. **Bloom Filters – the Math** (this follows the description in [3])

One prominent feature of Bloom filters is that there is a clear tradeoff between the size of the filter and the rate of false positives. Observe that after inserting *n* keys into a filter of size *m* using *k* hash functions, the probability that a particular bit is still 0 is:

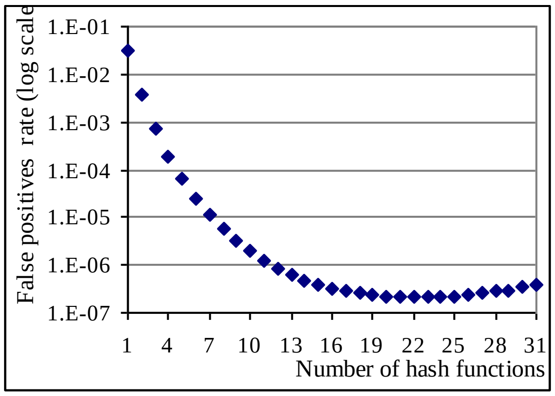
$$p_0 = \left(1 - \frac{1}{m}\right)^{kn} \approx 1 - e^{-\frac{kn}{m}}. \quad (1)$$

(Note that we assume perfect hash functions that spread the elements of *A* evenly throughout the space {1..*m*}. In practice, good results have been achieved using MD5 and other hash functions [10].)

Hence, the probability of a false positive (the probability that all k bits have been previously set) is:

$$p_{err} = (1 - p_0)^k = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \qquad (2)$$

In (2) p_{err} is minimized for $k = \frac{m}{n} \ln 2$ hash functions. In practice however, only a small number of hash functions are used. The reason is that the computational overhead of each hash additional function is constant while the incremental benefit of adding a new hash function decreases after a certain threshold (see Figure 1).



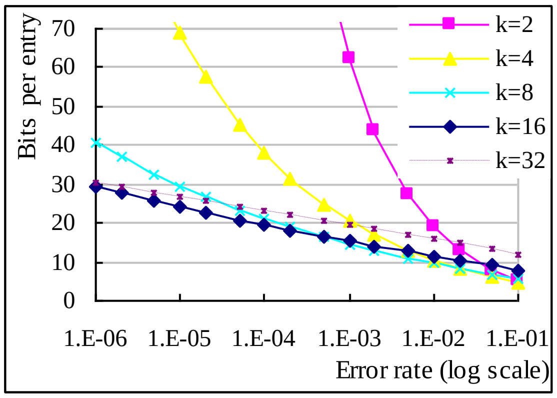


Figure 1: False positive rate as a function of the number of hash functions used. The size of the Bloom filter is 32 bits per entry (m/n=32). In this case using 22 hash functions minimizes the false positive rate. Note however that adding a hash function does not significantly decrease the error rate when more than 10 hashes are already used.

Figure 2: Size of Bloom filter (bits/entry) as a function of the error rate desired. Different lines represent different numbers of hash keys used. Note that, for the error rates considered, using 32 keys does not bring significant benefits over using only 8 keys.

(2) is the base formula for engineering Bloom filters. It allows, for example, computing minimal memory requirements (filter size) and number of hash functions given the maximum acceptable false positives rate and number of elements in the set (as we detail in Figure 2).

$$\frac{m}{n} = \frac{-k}{\ln\left(1 - e^{\frac{\ln p_{err}}{k}}\right)} \text{ (bits per entry)} \qquad (3)$$

To summarize: Bloom filters are compact data structures for probabilistic representation of a set in order to support membership queries. The main design tradeoffs are the number of hash functions used (driving the computational overhead), the size of the filter and the error (collision) rate. Formula (2) is the main formula to tune parameters according to application requirements.

5. Compressed Bloom filters

Some applications that use Bloom filters need to communicate these filters across the network. In this case, besides the three performance metrics we have seen so far: (1) the computational overhead to lookup a value (related to the number of hash functions used),

(2) the size of the filter in memory, and (3) the error rate, a fourth metric can be used: the size of the filter transmitted across the network. M. Mitzenmacher shows in [8] that compressing Bloom filters might lead to significant bandwidth savings at the cost of higher memory requirements (larger uncompressed filters) and some additional computation time to compress the filter that is sent across the network. We do not detail here all theoretical and practical issues analyzed in [8].

6. References

1. http://www.squid-cache.org/.
2. Bloom, B. Space/time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, *13* (7). 422-426.
3. Fan, L., Cao, P., Almeida, J. and Broder, A., Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. in *Proceedings of ACM SIGCOMM'98*, (Vancouver, Canada, 1998).
4. Gribble, S.D., Brewer, E.A., Hellerstein, J.M. and Culler, D., Scalable, Distributed Data Structures for Internet Service Construction. in *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, (San Diego, CA, 2000).
5. Gribble, S.D., Welsh, M., Behren, R.v., Brewer, E.A., Culler, D., Borisov, N., Czerwinski, S., Gummadi, R., Hill, J., Joseph, A.D., Katz, R.H., Mao, Z., Ross, S. and Zhao, B. The Ninja Architecture for Robust Internet-Scale Systems and Services. *Special Issue of Computer Networks on Pervasive Computing*.
6. Hodes, T.D., Czerwinski, S.E., Zhao, B.Y., Joseph, A.D. and Katz, R.H. An Architecture for Secure Wide-Area Service Discovery. *Wireless Networks*.
7. Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C. and Zhao, B., OceanStore: An Architecture for Global-Scale Persistent Storage. in *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, (Cambridge, MA, 2000).
8. Mitzenmacher, M., Compressed Bloom Filters. in *Twentieth ACM Symposium on Principles of Distributed Computing (PODC 2001)*, (Newport, Rhode Island, 2001).
9. Mullin, J.K. A second look at Bloom filters. *Communications of the ACM*, *26* (8). 570-571.
10. Ramakrishna, M.V. Practical performance of Bloom filters and parallel free-text searching. *Communications of the ACM*, *32* (10). 1237-1239.
11. Rhea, S. and Weimer, W., Data Location in the OceanStore. in *unpublished*, (1999), UC Berkeley.