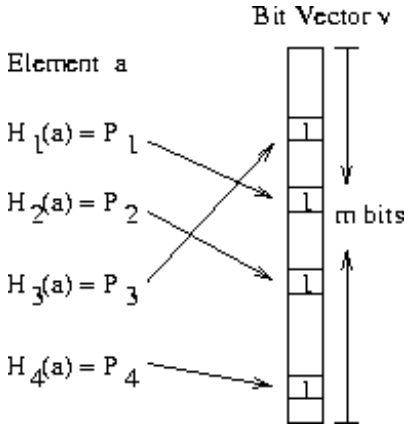
Bloom Filters - the math

A Bloom filter is a method for representing a set $A = \{a_1, a_2, \ldots, a_n\}$ of *n* elements (also called keys) to support membership queries. It was invented by Burton Bloom in 1970 [6] and was proposed for use in the web context by Marais and Bharat [37] as a mechani sm for identifying which pages have associated comments stored within a *CommonKnowledge* server.

Figure 3: A Bloom Filter with 4 hash functions.



The idea (illustrated in Figure 3) is to allocate a vector *v* of *m* bits, initially all set to 0, and then choose *k* independent hash functions, ,each with range $\{1, \ldots, m\}$. For each element $a \in A$, the bits at positions $h_1(a)$, $h_2(a)$, ..., $h_k(a)$ in *v* are set to 1. (A particular bit might be set to 1 multiple times.) Given a query for *b* we check the bits at positions $h_1(b)$, $h_2(b)$, ..., $h_k(b)$. If any of them is 0, then certainly *b* is not in the set *A*. Otherwise we conjecture that *b* is in the set although there is a certain probability that we are wrong. This is called a ``false positive.'' or, for historical reasons, a ``false drop.'' The parameters *k* and *m* should be chosen such that the probability of a false positive (and hence a false hit) is acceptable.

The salient feature of Bloom filters is that there is a clear tradeoff between *m* and the probability of a false positive. Observe that after inserting *n* keys into a table of size *m*, the probability that a particular bit is still 0 is exactly

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

Hence the probability of a false positive in this situation is

The right hand side is minimized for $k = \ln 2 \times m/n$, in which case it becomes

In fact *k* must be an integer and in practice we might chose a value less than optimal to reduce computational overhead. Some example values are:

Table 3 though Table 5 list the false positive retios for common combinations of *m*/*n* and *k*.

Table 3: False positive rate under various *m*/*n* and *k* combinations.

m/*n*	*k*	*k*=1	*k*=2	*k*=3	*k*=4	*k*=5	*k*=6	*k*=7	*k*=8
2	1.39	0.393	0.400						
3	2.08	0.283	0.237	0.253					
4	2.77	0.221	0.155	0.147	0.160				
5	3.46	0.181	0.109	0.092	0.092	0.101			
6	4.16	0.154	0.0804	0.0609	0.0561	0.0578	0.0638		
7	4.85	0.133	0.0618	0.0423	0.0359	0.0347	0.0364		
8	5.55	0.118	0.0489	0.0306	0.024	0.0217	0.0216	0.0229	
9	6.24	0.105	0.0397	0.0228	0.0166	0.0141	0.0133	0.0135	0.0145
10	6.93	0.0952	0.0329	0.0174	0.0118	0.00943	0.00844	0.00819	0.00846
11	7.62	0.0869	0.0276	0.0136	0.00864	0.0065	0.00552	0.00513	0.00509
12	8.32	0.08	0.0236	0.0108	0.00646	0.00459	0.00371	0.00329	0.00314
13	9.01	0.074	0.0203	0.00875	0.00492	0.00332	0.00255	0.00217	0.00199
14	9.7	0.0689	0.0177	0.00718	0.00381	0.00244	0.00179	0.00146	0.00129
15	10.4	0.0645	0.0156	0.00596	0.003	0.00183	0.00128	0.001	0.000852
16	11.1	0.0606	0.0138	0.005	0.00239	0.00139	0.000935	0.000702	0.000574
17	11.8	0.0571	0.0123	0.00423	0.00193	0.00107	0.000692	0.000499	0.000394
18	12.5	0.054	0.0111	0.00362	0.00158	0.000839	0.000519	0.00036	0.000275
19	13.2	0.0513	0.00998	0.00312	0.0013	0.000663	0.000394	0.000264	0.000194
20	13.9	0.0488	0.00906	0.0027	0.00108	0.00053	0.000303	0.000196	0.00014
21	14.6	0.0465	0.00825	0.00236	0.000905	0.000427	0.000236	0.000147	0.000101
22	15.2	0.0444	0.00755	0.00207	0.000764	0.000347	0.000185	0.000112	7.46e-05
23	15.9	0.0425	0.00694	0.00183	0.000649	0.000285	0.000147	8.56e-05	5.55e-05
24	16.6	0.0408	0.00639	0.00162	0.000555	0.000235	0.000117	6.63e-05	4.17e-05
25	17.3	0.0392	0.00591	0.00145	0.000478	0.000196	9.44e-05	5.18e-05	3.16e-05
26	18	0.0377	0.00548	0.00129	0.000413	0.000164	7.66e-05	4.08e-05	2.42e-05

27	18.7	0.0364	0.0051	0.00116	0.000359	0.000138	6.26e-05	3.24e-05	1.87e-05
28	19.4	0.0351	0.00475	0.00105	0.000314	0.000117	5.15e-05	2.59e-05	1.46e-05
29	20.1	0.0339	0.00444	0.000949	0.000276	9.96e-05	4.26e-05	2.09e-05	1.14e-05
30	20.8	0.0328	0.00416	0.000862	0.000243	8.53e-05	3.55e-05	1.69e-05	9.01e-06
31	21.5	0.0317	0.0039	0.000785	0.000215	7.33e-05	2.97e-05	1.38e-05	7.16e-06
32	22.2	0.0308	0.00367	0.000717	0.000191	6.33e-05	2.5e-05	1.13e-05	5.73e-06

Table 4: False positive rate under various *m*/*n* and *k* combinations.

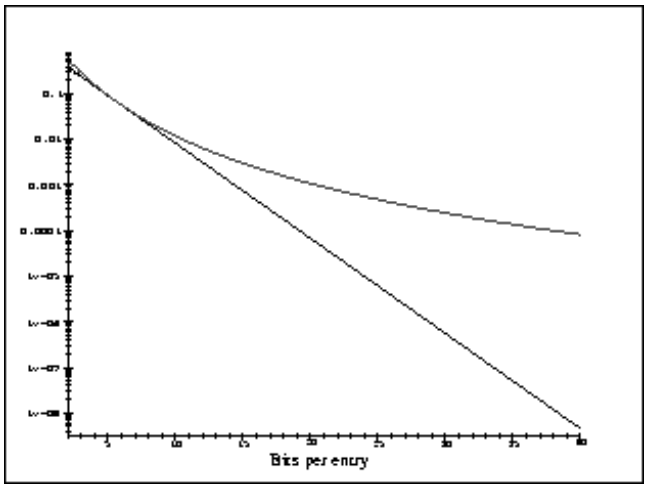
m/*n*	*k*	*k*=9	*k*=10	*k*=11	*k*=12	*k*=13	*k*=14	*k*=15	*k*=16
11	7.62	0.00531							
12	8.32	0.00317	0.00334						
13	9.01	0.00194	0.00198	0.0021					
14	9.7	0.00121	0.0012	0.00124					
15	10.4	0.000775	0.000744	0.000747	0.000778				
16	11.1	0.000505	0.00047	0.000459	0.000466	0.000488			
17	11.8	0.000335	0.000302	0.000287	0.000284	0.000291			
18	12.5	0.000226	0.000198	0.000183	0.000176	0.000176	0.000182		
19	13.2	0.000155	0.000132	0.000118	0.000111	0.000109	0.00011	0.000114	
20	13.9	0.000108	8.89e-05	7.77e-05	7.12e-05	6.79e-05	6.71e-05	6.84e-05	
21	14.6	7.59e-05	6.09e-05	5.18e-05	4.63e-05	4.31e-05	4.17e-05	4.16e-05	4.27e-05
22	15.2	5.42e-05	4.23e-05	3.5e-05	3.05e-05	2.78e-05	2.63e-05	2.57e-05	2.59e-05
23	15.9	3.92e-05	2.97e-05	2.4e-05	2.04e-05	1.81e-05	1.68e-05	1.61e-05	1.59e-05
24	16.6	2.86e-05	2.11e-05	1.66e-05	1.38e-05	1.2e-05	1.08e-05	1.02e-05	9.87e-06
25	17.3	2.11e-05	1.52e-05	1.16e-05	9.42e-06	8.01e-06	7.1e-06	6.54e-06	6.22e-06
26	18	1.57e-05	1.1e-05	8.23e-06	6.52e-06	5.42e-06	4.7e-06	4.24e-06	3.96e-06
27	18.7	1.18e-05	8.07e-06	5.89e-06	4.56e-06	3.7e-06	3.15e-06	2.79e-06	2.55e-06
28	19.4	8.96e-06	5.97e-06	4.25e-06	3.22e-06	2.56e-06	2.13e-06	1.85e-06	1.66e-06
29	20.1	6.85e-06	4.45e-06	3.1e-06	2.29e-06	1.79e-06	1.46e-06	1.24e-06	1.09e-06
30	20.8	5.28e-06	3.35e-06	2.28e-06	1.65e-06	1.26e-06	1.01e-06	8.39e-06	7.26e-06
31	21.5	4.1e-06	2.54e-06	1.69e-06	1.2e-06	8.93e-07	7e-07	5.73e-07	4.87e-07
32	22.2	3.2e-06	1.94e-06	1.26e-06	8.74e-07	6.4e-07	4.92e-07	3.95e-07	3.3e-07

Table 5: False positive rate under various *m*/*n* and *k* combinations.

m/*n*	*k*	*k*=17	*k*=18	*k*=19	*k*=20	*k*=21	*k*=22	*k*=23	*k*=24
22	15.2	2.67e-05							
23	15.9	1.61e-05							
24	16.6	9.84e-06	1e-05						
25	17.3	6.08e-06	6.11e-06	6.27e-06					
26	18	3.81e-06	3.76e-06	3.8e-06	3.92e-06				
27	18.7	2.41e-06	2.34e-06	2.33e-06	2.37e-06				
28	19.4	1.54e-06	1.47e-06	1.44e-06	1.44e-06	1.48e-06			
29	20.1	9.96e-07	9.35e-07	9.01e-07	8.89e-07	8.96e-07	9.21e-07		
30	20.8	6.5e-07	6e-07	5.69e-07	5.54e-07	5.5e-07	5.58e-07		
31	21.5	4.29e-07	3.89e-07	3.63e-07	3.48e-07	3.41e-07	3.41e-07	3.48e-07	
32	22.2	2.85e-07	2.55e-07	2.34e-07	2.21e-07	2.13e-07	2.1e-07	2.12e-07	2.17e-07

The graph in Figure 4 shows the probability of a false positive as a function of the number of bits allocated for each entry, that is, the ratio $\alpha = n/m$. The curve above is for the case of 4 hash functions. The curve below is for the optimum number of hash functions. The scale is logarithmic so the straight line observed corresponds to an exponential decrease. It is clear that Bloom filters require very little storage per key at the slight risk of some false positives. For instance for a bit array 10 times larger than the number of entries, the probability of a false positive is 1.2% for 4 hash functions, and 0.9% for the optimum case of 5 hash functions. The probability of false positives can be easily decreased by allocating more memory.

Figure 4: Probability of false positives (log scale). The top curve is for 4 hash functions. The bottom curve is for the optimum (integral) number of hash functions.



Since in our context each proxy maintains a local Bloom filter to represent its own cached documents, changes of set *A* must be supported. This is done by maintaining for each location ℓ in the bit array a count $c(\ell)$ of the number of times that the bit is set to 1 (that is, the number of elements that hashed to ℓ under any of the hash functions). All the counts are initially 0. When a key *a* (in our case, the URL of a document) is inserted or deleted, the counts $c(h_1(a))$, $c(h_2(a))$, ..., $c(h_k(a))$ are incremented or decremented accordingly. When a count changes from 0 to 1, the corresponding bit is turned on. When a count changes from 1 to 0 the corresponding bit is turned off. Hence the local Bloom filter always reflects correctly the current directory.

Since we also need to allocate memory for the counts, it is important to know how large they can become. The asymptotic expected maximum count after inserting *n* keys with *k* hash functions into a bit array of size *m* is (see [22, p. 72])

$$\Gamma^{-1}(m)\left(1+\frac{\ln(kn/m)}{\ln\Gamma^{-1}(m)}+O\left(\frac{1}{\ln^2\Gamma^{-1}(m)}\right)\right),$$

and the probability that any count is greater or equal *i* is

As already mentioned the optimum value for k (over reals) is so assuming that the number of hash functions is less than we can further bound

$$\Pr(\max(c)\geq i)\leq m\left(\frac{e\ln 2}{i}\right)^i.$$

Hence taking *i*=16 we obtain that

In other words if we allow 4 bits per count, the probability of overflow for practical values of *m* during the initial insertion in the table is minuscule.

In practice we must take into account that the hash functions are not truly random, and that we keep doing insertions and deletions. Nevertheless, it seems that 4 bits per count would be amply sufficient. Furthermore if the count ever exceeds 15, we can simply let it stay at 15; after many deletions this might lead to a situation where the Bloom filter allows a false negative (the count becomes 0 when it shouldn't be), but the probability of such a chain of events is so low that it is much more likely that the proxy server would be rebooted in the meantime and the entire structure reconstructed.

Next: Bloom Filters as Summaries **Up:** Summary Cache **Previous:** Summary Representations

Pei Cao
7/5/1998