Lecture 1 — January 14, 2002

Lecturer: John Byers *BOSTON UNIVERSITY* *Scribe: Marwan Fayed*

This day's lecture, we learned how to *Compactly Represent Bags of Integers* using **Bloom filters**. The formal definition of the problem, and its solution, appear below, but let us first understand why this is necessary.

1.1 A Sample Application

Suppose we have a set of valid URLs, U. Suppose further that each URL is approximately 100 characters in length; thus each URL requires 800 bits for proper representation. It should be clear that any subset of URLs in U of size n requires $800n$ bits.

Now consider the following basic caching structure from [2] where

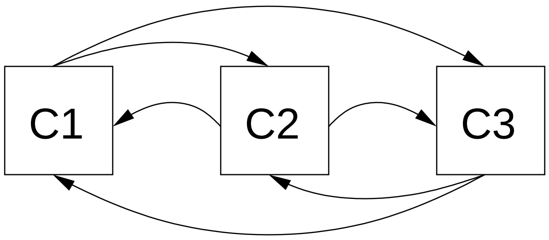

Table 1.1. A simple shared caching structure.

- C_1, C_2, C_3 are caches, each with a set of documents stored; documents are indexed by URL
- edges represent the xfer of a cache's set of URLs
- generally, a query for URL x is sent to one of the caches; that cache must determine which (if any) of the other caches has x.

One can imagine that if the set of URLs being exchanged is large, the total transfer time can be enourmous. In the above case, for example, if every cache stores $10,000$ entries then a total of $48,000,000$ bits[1] are exchanged.

Clearly this is not acceptable. As it turns out, if we are willing to accept a small margin of error then Bloom filters allow us to reduce the bits required by a factor of ~ 80.

1.2 Compactly Representing Bags of Integers

From that above application we can see the importance of needing to represent large amounts of information in a compact manner. Subsequent sections detail a formalization of the problem, Bloom filters, an Bloom filter example and a proof that Bloom filters work.

[1] 3 caches with 10,000 entries each; each chache shares with 2 other caches, and each entry requires 800 bits

1.2.1 Formalization

Now that we have seen the importance of compactly representing information we need to formalize the problem. Suppose that we are interested in n-element sets of the universe,

$$X_i = \{x_{i1}, x_{i2}, ..., x_{in}\}, x_{ij} \in U, |U| = u, u \gg n. \tag{1.1}$$

On this set we need to perform the queries and operations,

- **Membership:** $y \in X_i$?
- **Insertion :** add y to X_i
- Deletion[2]: remove y from X_i

1.2.2 Evaluation

In any solution we are concerned with running time and space complexity (transfer size, in our context). However, we might also consider the accuracy of a data structure. For now, we address the space required to represent X_i.

There are $\binom{u}{n}$ possible sets of size n.

Aside. How many possible multisets of size n exist?[3] Proving the answer is left as an exercise.

In order to determine the number of bits needed to represent $\binom{u}{n}$ sets we use the following theorem and corollaries:

Theorem 1.1. *Stirling's Approximation:* $a! = \sqrt{2\pi a}\left(\frac{a}{e}\right)^a(1 + o(1))$

Corollary 1.2. $\binom{a}{b} \geq \left(\frac{a}{b}\right)^b$

Corollary 1.3. $\binom{a}{b} \leq \left(\frac{ae}{b}\right)^b$

Corollary 1.4. $\binom{a}{b} \sim \frac{a^b}{b!}$*, for large* a.

We can safely say that the number of bits required to represent our sets is at least $log_2\binom{u}{n}$. Using corollary 1.2,

$$\begin{aligned} \log\binom{u}{n} &\geq \log\left(\frac{u}{n}\right)^n \\ &\geq n\log(u) - n\log(n) \\ &= \Omega(n\log(u)) \end{aligned}$$

So we can fully represent our set using $nlog(u)$ bits. But exactly how is this possible?

1.2.3 Using $n\log(n)$ Bits...

This is quite easy. We need simply make sure to

1. map elements of U to integers;
    - ie. map $U \to N$ so that $U_N = \{1..u\}$
2. it is now easy to enumerate the elements in X_i;
3. look-up and insert using some lexicographical structure.

This is all we can do if we wish to represent our set in an exact manner. However, as we saw in the example application above, this results in a high messaging complexity. As it turns out, we are able to do much better using Bloom filters if we are willing to accept a small penalty in accuracy.

[2]Deletion may be required to a lesser extent and will be discussed in asides as necessary.

[3]Answer: $\left(\binom{u}{n}\right) = \binom{u+n-1}{n}$

1.3 Bloom Filters

Definition. A Bloom filter is a bit vector, v, of m bits with k independent hash functions, $h_1(x)$, $h_2(x)$, $\ldots$, $h_k(x)$. Of course, x is our search value. So,

$$\forall i,\ h_i : U \to \{0, \ldots, m-1\},\ m \ll u$$

The required operations are implemented in the following manner:

- Initialization:
  1. Set all m bits of v to zero.
- Insertion of $x \in U$:
  1. Compute $h_1(x)$, $h_2(x)$, $\ldots$, $h_k(x)$.
  2. Set $v[h_1(x)] = v[h_2(x)] = \ldots = v[h_k(x)] = 1$.
- Lookup of $x \in U$:
  1. Compute $h_1(x)$, $h_2(x)$, $\ldots$, $h_k(x)$.
     (a) If any of $v[h_i(x)]$ is zero, then x is not in the filter.
     (b) Otherwise, report x in filter. Given the nature of compression, and query may return a positive result when no such x exists.

1.3.1 An Example...

To make the above operations clear, a concise example is given. Consider an instance where,

$$\begin{aligned} &m = 5,\ k = 2 \\ &h_1(x) = x\%5 \\ &h_2(x) = (2x+3)\%5 \end{aligned}$$

Now perform the following operations. First we initialize our vector.

Initialize v:	0	0	0	0	0

Now insert 9 and 11. Remember that we compute all hash functions on the inserting value, then set the corresponding value in the Bloom filter to 1.

	$h_1(x)$	$h_2(x)$					
Insert(9):	4	1	0	1	0	0	1
Insert(11):	1	0	1	1	0	0	1

Finally, let us attempt some lookups. Remember that if the value corresponding to the index of any calculated hash function is 0, then the item cannot be in the filter. Given the filter as constructed above, attempt the following operations.

	$h_1(x)$	$h_2(x)$	*Result*
Lookup(15):	0	3	$v[3] = 0$; "Not in Filter."
Lookup(16):	1	0	"16 is **in** Filter."

Notice that 16 was never inserted. As a result, **false positives are possible.**

This is obviously a contrived example but it does demonstrate that Bloom filters are not always accurate. The likelihood of a false positive is discussed in the next section.
One may wish to note that deletions from may be supported with the use of *counting* filters. That is, any insertion increases the appropriate value by 1 rather than simply set it to 1. In keeping with this idea a deletion decreases appropriate values in the filter by 1. This, of course, requires that insertions be tracked.

1.4 Analysis

As seen above, Bloom filters can report false positives (ie. report $a \in X$ when $a \notin X$). The probability of false positives,p, is called the *false positive rate* [3], and the key to understanding them is to analyse their relationship to m.

$$\boxed{\text{FACT: } \left(1 - \tfrac{1}{x}\right)^y \approx e^{-y/x}}$$

The probability that one hash fails to set a given a bit is $\left(1 - \frac{1}{m}\right)$. Using the above fact, p is

$$\begin{aligned} Pr[\text{bit is still zero}] &= \left(1 - \frac{1}{m}\right)^{\overbrace{kn}^{no.\ of\ hashes}} \\ &\approx e^{-\frac{kn}{m}} \end{aligned}$$

$$\boxed{\text{Note: This assumes that hash functions are independent and random.}}$$

If we let $p = e^{-kn/m}$ then we can say that

$$\begin{aligned} Pr[\text{false positive}] &= Pr[\text{All k bits are 1}] && (1.2) \\ &= \underbrace{\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)}_{Prob\ one\ bit\ is\ 1}{}^{k} && (1.3) \\ &\approx \left(1 - e^{-\frac{kn}{m}}\right)^k && (1.4) \\ &= (1-p)^k && (1.5) \end{aligned}$$

Now minimize f, the false positive rate, by finding the optimal number of hash functions. That is, minimize f as a function of k by taking the derivative. To simplify the math, minimize the logarithm[4] of f with respect to k.

$$\text{Let } g = \ln(f) = k \ln\left(1 - \left(1 - \tfrac{1}{m}\right)^{nk}\right). \text{ Then, } \tfrac{dg}{dk} = \ln\left(1 - e^{-\frac{kn}{m}}\right) + \tfrac{kn}{m}\tfrac{e^{-\frac{kn}{m}}}{1-e^{-\frac{kn}{m}}}.$$

We find the optimal k, or right number of hash functions to use, when the derivative is 0. This occurs when $k = \frac{\ln 2m}{n}$. Substitute this value into (1.4), above,

$$f\left(\ln 2\tfrac{m}{n}\right) = \left(\tfrac{1}{2}\right)^k = (0.6185)^{\frac{m}{n}}$$

Of course as m grows in proportion to n, the false probability rate decreases. To illustrate, when $m = 8n$ there exists a 2% chance of error; when $m = 10n$ the false positive rate is less than 1%. As one can see, Bloom filters are effective data structures which space and messaging complexity, while maintaining an acceptable level of accuracy.

[4] I was reminded that minimizing the logarithm of a function is equivalent to minimizing the function itself.

Bibliography

[1] B. Bloom. "*Space/time trade-offs in hash coding with allowable errors*," Communications of the ACM, 13(7):422-426, 1970.

[2] L. Fan, P. Cao, J. Almeida and A. Z. Broder, "*Summary Cache: A Scalable Wide-area Cache Sharing Protocol*," in Proceedings of ACM SIGCOMM '98.

[3] M. Mitzenmacher. "*Compressed Bloom Filters*," in Proceedings of PODC 2001.