Bloom filters

CSE 30331/34331

2014/11/24

1 The idea

Bloom filters are an implementation of the set ADT that are extremely space-efficient *if* your application can tolerate some small rate of false positive errors (i.e., an element is not truly in the set but your data structure reports that it is). The classic example is spell-checking: if a word is correctly spelled and the checker says it is not a word, the user will be annoyed, but if a word is incorrectly spelled and the checker says it is a word once in a while, the user probably won't notice.

In a conventional hash table, we have an array, t, of m pointers. To search for an element x, we compute the hash of x – call it $h(x)$ – and then examine $t[h(x)]$. If it is a null pointer, we return false; otherwise, we have to check that the element stored there is really x and not just a collision. We talked about various strategies to do this (linear probing, open addressing, etc.).

What if we don't bother with that last check? If a word is correctly spelled, we will always return the correct answer, true. If a word is incorrectly spelled, we might return either true or false. The nice thing about this is that we don't even need to store the words themselves – only an array of bits. The bigger the table is, the less likely we are to make a false positive error. So, it would seem we have a tradeoff between compactness and accuracy.

The trick to getting out of this tradeoff is to use *multiple hash functions*, $h_1, \ldots, h_k$. When we add an element, we set all the bits $t[h_1(x)], \ldots, t[h_k(x)]$ to 1. And when we search for an element, we check to see if all of the bits $t[h_1(x)], \ldots, t[h_k(x)]$ are 1. If they are, we return true; if any of them is 0, we return false.

If the probability of a false-positive with a single hash function ($k = 1$) is p, then the probability of a false-positive with k hash functions is p^k. When all the math is worked out (see next section), the size of the data structure turns out to increase only linearly in k, but the probability of error goes down *exponentially* in k.

How do we come up with k different hash functions? It turns out that this works very well:

$$h_i(x) = a(x) + ib(x),$$

where a and b are two independent hash functions.[1] And then, it works fine for a and b to be different hash functions (e.g., Bob Jenkins' SpookyHash) but different seeds.

2 The theory (optional material, not on exam)

Given a desired error rate ϵ and an anticipated number of elements n, can we find the optimal settings of m, the size of the table, and k, the number of hash functions?

ϵ	desired error rate
n	number of elements
m	number of bits
k	number of hash functions

[1]Kirsch and Mitzenmacher, 2007. Less hashing, same performance: building a better Bloom filter.

The probability that any given bit is 1 is

$$\begin{aligned} p_1 &= 1 - \left(1 - \frac{1}{m}\right)^{kn} \\ &= 1 - \left(\left(1 - \frac{1}{m}\right)^m\right)^{kn/m} \\ &\approx 1 - e^{-kn/m} \qquad \text{for large } m \end{aligned}$$

Suppose we look up a element not in the set. The probability that all the bits we lookup are 1 is

$$\begin{aligned} p_e &\approx p_1^k \\ &= (1 - e^{-kn/m})^k \\ \ln p_e &= k \ln(1 - e^{-kn/m}) \end{aligned}$$

Let's maximize $\ln p_e$ by setting its derivative to zero:

$$\begin{aligned} \frac{d(\ln p_e)}{dk} &= \ln(1 - e^{-kn/m}) + kn/m \frac{e^{-kn/m}}{1 - e^{-kn/m}} \\ &= \ln p_1 - \ln(1 - p_1)\frac{1 - p_1}{p_1} := 0 \\ p_1 \ln p_1 &= (1 - p_1)\ln(1 - p_1) \\ p_1 &= \frac{1}{2}. \end{aligned}$$

So, given n and ϵ, the optimal settings are:

$$\begin{aligned} k &= -\log_2 \epsilon \\ m &= kn/\ln 2. \end{aligned}$$