COMS21103

Bloom filters

Ashley Montanaro
ashley@cs.bris.ac.uk

Department of Computer Science, University of Bristol
Bristol, UK

19 November 2013

Introduction

- Imagine we would like to build a web cache application. We would like to store URLs in some space-efficient way such that we can check membership in the cache very efficiently.
- Ideally, we would like to use $O(n)$ space to store n keys (i.e. URLs) picked from a universe of size U, where U is much bigger than n, and would like to be able to check membership in the cache in time $O(1)$.
- These are all the operations we care about: that is, instead of supporting Insert, Delete, Find and Successor operations, we will just want to support Insert and Member.
- The data structure maintains a subset $S \subseteq U$ of keys. The operation Member(k) should just return whether or not the supplied key k is contained within S.

Introduction

Bloom filters are a randomised data structure which achieve this goal. However, they have some important caveats:

- Bloom filters do not support deletion; they only support Insert and Member.
- They are not deterministic but have some risk of false positives.
- That is, when we query the Bloom filter with some key k, if $k \notin S$ there is some small chance (say 1%) that the answer is "yes" when it should be "no". On the other hand, if $k \in S$ the answer is always "yes" .

This is reasonable for applications like a web cache:

- If we incorrectly think that a page is in the cache, this is not a disaster: we check the cache first, find it is not there, and download it directly.
- However, if we incorrectly decide that a page is not in the cache, this is undesirable because we download the page unnecessarily.

Example

The following sequence of operations illustrates what can happen using a Bloom filter.

Operation	Returns
Insert(`www.bbc.co.uk`)	
Insert(`twitter.com`)	
Member(`cs.bristol.ac.uk`)	No
Member(`www.bbc.co.uk`)	Yes
Insert(`facebook.com`)	
Member(`cs.bristol.ac.uk`)	Yes

The last "Yes" is an example of a false positive.

A naïve approach

- The simplest thing we could do to implement the web cache is to maintain a string B of U bits in an array, where bit $B[k]$ is set to 0 or 1 depending on whether $k \in S$.
- For example, if the universe is the integers between 1 and 10, after inserting 3, 6 and 8 we have:

0	0	1	0	0	1	0	1	0	0

- If we would like the storage space used not to depend on U, we will need to compress this string somehow.

Hashing

- One way to do this is by hashing. We maintain an m-bit string B in our structure, for some m to be determined. Assume we have access to a hash function h which maps each key k to an integer $h(k)$ between 1 and m.
- Our structure will set bit number $h(k)$ of B to 1 when key k is inserted.
- Then, to determine whether $k \in S$, we just check whether the bit of B at position $h(k)$ is equal to 1.

Example

Imagine $m = 3$ and we have $h(\texttt{www.bbc.co.uk}) = 2$, $h(\texttt{facebook.com}) = 3$, $h(\texttt{cs.bristol.ac.uk}) = 3$.

Start	0	0	0
Insert(`www.bbc.co.uk`)	0	1	0
Insert(`facebook.com`)	0	1	1
Member(`cs.bristol.ac.uk`) returns Yes	0	1	1

Hashing

- A problem with this idea: if $m < U$, there will be some keys that hash to the same positions (collisions).
- If we call Member(k) for some $k \notin S$, if $h(k) = h(k')$ for some $k' \in S$, we will incorrectly output "yes".
- To make the probability of collisions low for the worst-case input, we pick our hash function $h(k)$ at random.
- For each key k, the value of $h(k)$ is uniformly random: that is, the probability that $h(k) = j$ is equal to $1/m$ for all j between 1 and m.

Hashing

What is the probability of a collision?

- Assume we have already inserted n keys into the structure and we would like to check whether some other key $k \notin S$ is contained in S (so the output should be "no").
- The bit-string B contains at most n 1's, and the value $h(k)$ is uniformly random; so the probability that $B[h(k)] = 1$ is at most n/m.
- So the probability that we incorrectly output "yes" for this key is at most n/m, and we never incorrectly output "no" for any key.
- So it suffices (for example) to take $m = 100n$ to achieve a failure probability of at most 1%. Note that m does not depend on the universe size U.

Can we do better?

We can achieve superior performance by using multiple hash functions.

- A Bloom filter consists of a string B of m bits, and a set of r hash functions $h_1, \dots, h_r$.
- Each hash function maps a key k to an integer between 1 and m.
- For each i, we assume as before that $h_i(k)$ is uniformly random: that is, for each key k, the probability that $h_i(k) = j$ is equal to $1/m$ for all j between 1 and m.
- We will choose the parameters m and r later.

Inserting into a Bloom filter

To insert into a Bloom filter, we use the following simple procedure.

Insert(k)

1. for $i \leftarrow 1$ to r
2. $\quad B[h_i(k)] \leftarrow 1$

To check membership, we just check the bits of B that should be set to 1.

Member(k)

1. for $i \leftarrow 1$ to r
2. $\quad$ if $B[h_i(k)] = 0$
3. $\quad\quad$ return false
4. return true

Example

Imagine $m = 4$, $r = 2$, and we randomly pick the following hash functions:

- h_1(`www.bbc.co.uk`) $= 2$, h_1(`facebook.com`) $= 3$, h_1(`cs.bristol.ac.uk`) $= 3$.
- h_2(`www.bbc.co.uk`) $= 1$, h_2(`facebook.com`) $= 2$, h_2(`cs.bristol.ac.uk`) $= 4$.

Start	0	0	0	0
Insert(`www.bbc.co.uk`)	1	1	0	0
Insert(`facebook.com`)	1	1	1	0
Member(`cs.bristol.ac.uk`) returns No	1	1	1	0

Does the Bloom filter work?

- Imagine $|S| = n$ and we query the filter with a key $k \notin S$.
- This is equivalent to checking r random indices $h_1(k), \dots, h_r(k)$ and returning Yes if all of the bits are set to 1. We now upper-bound the probability of this happening.
- If a p fraction of the bits of B are set to 1, the probability that all of the bits checked are set to 1 is precisely p^r.
- At most nr bits of B can be set to 1 (each key inserted sets at most r bits to 1).
- So the fraction of bits set to 1 is at most nr/m.
- So the probability that we incorrectly output 1 is at most $(nr/m)^r$.

Does the Bloom filter work?

We now choose r to optimise this bound.

- By taking the derivative, we find that the minimum of $(nr/m)^r$ is achieved when $r = m/(ne)$, where $e = 2.7818\dots$.
- With this value of r, we get that the failure probability is at most $e^{-m/(ne)} \approx 0.69^{m/n}$.
- So, to achieve failure probability p, we can choose any m such that $e^{-m/(ne)} \le p$, which is equivalent to

$$m \ge -en \ln p.$$

- For small p, this is much better than using one hash function. For example, to achieve $p = 0.01$ (i.e. a 1% failure probability), we can take $m \approx 12.52n$.

So the number of bits m used by the Bloom filter is only a (small) multiple of n, and does not depend on U.

Can we do as well deterministically?

Claim

Any data structure that stores a subset S of n elements of a universe of size U, in such a way that membership in S can be tested with certainty, must use $\Omega(n \log U)$ bits of storage.

Proof

- By testing membership in S of each element of the universe in turn, we can determine S completely, so the structure must contain enough information to identify S.
- Claim: there are at least $\lfloor U/n \rfloor^n$ subsets of U of size n.
- Proof: divide U into n blocks of (nearly) equal size, and consider only subsets with one item in each block. There are $\lfloor U/n \rfloor^n$ such subsets.

...

Lower bounds on storage space

Claim

Any data structure that stores a subset S of n elements of a universe of size U, in such a way that membership in S can be tested with certainty, must use $\Omega(n \log U)$ bits of storage.

Proof

- A data structure that uses b bits of storage can store at most 2^b different bit-strings.
- Thus, unless $2^b \ge \lfloor U/n \rfloor^n$, there must exist two subsets that correspond to the same bit-string.
- If the structure gives the right answer for all subsets, we must have

$$b \ge \log_2(\lfloor U/n \rfloor^n) = n(\log_2 \lfloor U/n \rfloor) = \Omega(n \log U).$$

□

Practical considerations

- We made the unrealistic assumption that each hash function h_i maps a key k to a uniformly random integer between 1 and m.
- In practice, we would pick each hash function h_i randomly from a fixed set of hash functions. One way of doing this for integer keys k (see CLRS §11.3.3) is to do the following for each i:
  1. Pick a prime number $p > U$.
  2. Pick random integers $a \in \{1, \dots, p-1\}$, $b \in \{0, \dots, p-1\}$.
  3. Let h_i be defined by $h_i(k) = 1 + ((ak + b) \mod p) \mod m$.
- Some number theory can be used to prove that this set of hash functions is "pseudorandom" in some sense; however, technically they are not "random enough" for our analysis above to go through.
- Nevertheless, in practice hash functions like this are very effective.

Summary

- Bloom filters provide a way of checking membership in a set which is very efficient in both space and time.
- By improving the analysis, one can show that they need only about $1.44 \log_2(1/\epsilon)$ bits per element of storage space to achieve failure probability ϵ.
- Bloom filters have a number of applications: web caches, databases (e.g. Google BigTable, Apache Cassandra), spell checkers, Bitcoin (!), the Linux kernel, . . .
- They are very efficient in theory and even more efficient in practice.
- There are modifications to Bloom filters to allow deletions ("counting Bloom filter"), storage of key values ("Bloomier filter"), dynamic scaling, . . .

Further reading

- Probability and Computing
  Michael Mitzenmacher and Eli Upfal
  Cambridge University Press
  - Section 5.5.3 – Bloom Filters
- Network Applications of Bloom Filters: A Survey
  Andrei Broder and Michael Mitzenmacher
  `http://www.eecs.harvard.edu/~michaelm/postscripts/im2005b.pdf`
- This year's lecture slides for COMS31900: Advanced Algorithms, for additional / more advanced material.
  - Lecture 5 – Bloom filters

Historical notes

- The Bloom filter was invented by Burton Howard Bloom in 1970, in a paper which now has over 4000 citations.
- His analysis of the structure turned out to have a bug which was only fixed in a paper published in 2008!
- Bloom is sadly lacking a Wikipedia page and online photo.