Bloom filters and Cuckoo hashing

RA-MIRI QT Curs 2020-2021

Bloom filter

Given a set of elements S, we want a Data structure for supporting insertions and querying about membership in S.

In particular we wish a DS s.t.

- *minimizes* the use of memory,
- *can check membership* as fast as possible.

Burton Bloom: The Bloom filter data structure. Comm. ACM, July 1970.
A hash data structure where each register in the table is one bit

Query on a list of e-mails

We have a set S of 10^9 e-mail addresses, where the typical e-mail address is 20 bites. Therefore it does not seem reasonable to store S in main memory. We can spare 1 Gigabyte of memory, which is approximately 10^9 bytes or 8×10^9 bites. How can put S in main memory to query it?

Definition Bloom filter

Create a one bit hash table $T[0, \ldots, m-1]$, and a hash function h. Initially all m bits are set to 0.

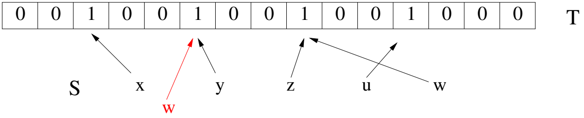
Giving a set $S = \{x_1, \ldots, x_n\}$ define a hashing function $h : S \to T$. For every $x_i \in S$, $h(x_i) \to T[j]$ and $T[j] := 1$. Given a set S a function $h()$ and a table $T[m]$:

Insert (x)
$h(x) \to i$
if $T[i] == 0$ **then**
  $T[i] = 1$
end if

inS(y)
$h(x) \to i$
if $T[i] == 1$ **then**
  return Yes
else
  return No
end if

Notice: once we have hashed S into T we can erase S.

False positives



Bloom filter needs $O(m)$ space and answers membership queries in $\Theta(1)$.

Inconvenience: Do not support removal and may have false positive.

In a query $y \in S$?, a Bloom filter always will report correctly if indeed $y \in S$ ($h(y) \to T[i]$ with $T[i] = 1$),
but if $y \notin S$ it may be the case that $h(y) \to T[i]$ with $T[i] = 1$, which is called a False positive.

How large is the error of having a false positive?

Probability of having a false positives

Let $|S| = n$, we constructed a BF $(h, T[m])$ with all elements in S. If we query about $y \in S$?, with $y \notin S$, and $h(y) \to T[i]$, what is the probability that $T[i] = 1$?

After all the elements of S are hashed into the Bloom filter, the probability that a specific $T[i] = 0$ is $(1 - \frac{1}{m})^n = e^{-n/m}$

(recall that: $e = \lim_{x\to\infty}(1 + \frac{1}{x})^x$, $e^{-1} = \lim_{x\to\infty}(1 - \frac{1}{x})^x$)

Therefore, for a $y \notin S$, the probability of false positive π:

$$\pi = \mathbf{Pr}\left[h(y) \to T[i] \,|\, \text{where } T[i] = 1\right] = 1-(1-\frac{1}{m})^n \sim 1-e^{-n/m}.$$

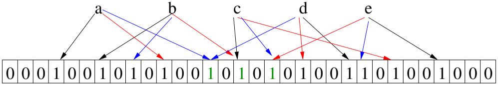
To minimise π, want to maximize $e^{-n/m}$
$\Rightarrow \frac{n}{m}$ has to be small, i.e, $m >> n$.

For ex.: if $m = 100n, \pi = 0.0095$; If $m = n, \pi = 0.632$ and if $m = n/10, \pi = 0.9999$

Alternative: Amplify

Take k different functions $\{h_1, h_2, \ldots, h_k\}$ in the same 2-universal set of functions.

Ex. Bloom filter with 3 hash functions: h_1, h_2, h_3.



When making a query about if $y \in S$, compute $h_1(y), \ldots h_t(y)$, if one of them is 0 we certainty $y \notin S$, else (if all the k hashing go to bits with value 1) $y \in S$ with some probability.
After hashing the n elements k times to T, for an specific $T[i]$:

$$p = \mathbf{Pr}\left[T[i] = 0\right] = (1 - \frac{1}{m})^{kn} = e^{-kn/m}.$$

The probability f of a false positive:

$$f = \left(1 - e^{-kn/m}\right)^k = (1-p)^k$$

Asymptotic estimations for k and m

To minimize the probability of having a false positive: $\frac{\mathrm{d}p}{\mathrm{d}k} = 0$

Let $f(k) = \ln p$ then $f(k) = k \ln(1 - e^{-kn/m})$

$\Rightarrow f'(k) = \ln(1 - e^{-kn/m}) + \frac{kne^{-kn/m}}{m(1-e^{-kn/m})}$

Making $f'(k) = 0$, we get

$$\boxed{k_{\mathrm{opt}} = \frac{m}{n}\frac{1}{2}\ln 2 = \frac{9}{13}\frac{m}{n}}$$

The probability of having a false positive for k_{opt} is

$$p_0 = (1 - e^{\frac{9}{13}\frac{m}{n}\frac{n}{m}})^{\frac{9}{13}\frac{m}{n}} \sim (\frac{1}{2})^{\frac{9m}{13n}} = 0.619223^{\frac{m}{n}}.$$

Optimizing k

Given n and m we want to find the optimal value of k to minimize the probability of a false positive $f(k) = (1 - e^{-kn/m})^k$

Define $g(k) = \ln f(k) = k \ln(1 - e^{-kn/m})$. Minimizing f is equivalent to minimizing g.

To minimize the probability of having a false positive: $\frac{\mathrm{d}g(k)}{\mathrm{d}k} = 0$

$\Rightarrow \frac{\mathrm{d}g(k)}{\mathrm{d}k} = \ln(1 - e^{-kn/m}) + \frac{kne^{-kn/m}}{m(1-e^{-kn/m})} = 0,$
$\Rightarrow$ when n, m are given, to minimize f is $k_o = (\ln 2)\frac{m}{n}$.

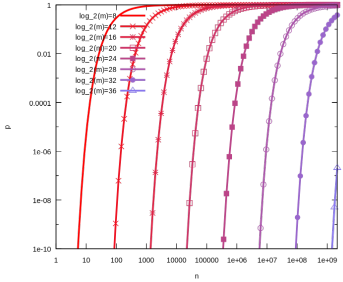
In this case the false positive probability $f_o = 0.6185^{m/n}$.

Bloom filters allow a constant probability of false positive, $m = cn$ for small constant c, i.e. m grows linear wrt n.

For ex.: if $c = 2$ and $k = 6$ the false positive probability is around 2%.

Practical issues

On the other hand although the results shown before are asymptotic, there also work for practical values of n.
(Fig 3 in Takoma, Rothnberg, Lagerpetz: Theory and Practice of Bloom Filters for Distributed Systems) Gives the probability of false positive (y) wrt to n (x), and as function of m, with $k = \ln 2 \frac{n}{m}$.



Further applications of Bloom filters

Bloom filters are useful when a set of keys is used and space is important.

- The Google Chrome web browser used to use a Bloom filter to identify malicious URLs. Any URL was first checked against a local Bloom filter, and only if the Bloom filter returned a positive result was a full check of the URL performed (and the user warned, if that too returned a positive result)
- Packet routing: Bloom filters provide a means to speed up or simplify packet routing protocols.
- IP Tracebook
- Useful tool for measurement infrastructures used to create data summaries in routers or other network devices.

A. Broder, M. Mitzenmacher: *Network applications of Bloom filters: A survey.* Internet Mathematics, 1,4: 485-509, 2005

Cuckoo Hashing

Pagh, Rodler: *Cuckoo Hashing*. ESA-2001

Cuckoo hashing is a hashing technique where:

- Lookups are $\Theta(1)$ worst-case.
- Deletions are $\Theta(1)$ worst-case.
- Insertions are $O(1)$ in expectation.

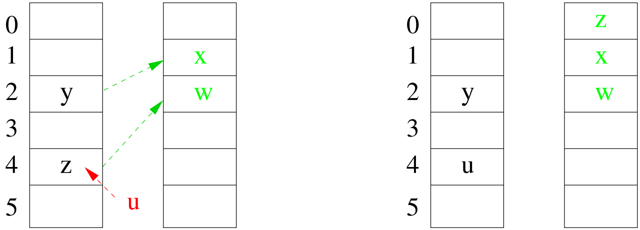
Cuckoo Hashing

- We have two hash tables T_1, T_2 with size m each and two hash functions h_1 for T_1 and h_2 for T_2.
- Can use for instance $h_1(k) = k \mod m$ and $h_2(k) = \lceil k/m \rceil \mod m$
- Every element $k \in \mathcal{U}$ can be only in two positions: at $h_1(k)$ in T_1 or at $h_2(k)$ in T_2.
- Lookups take $\Theta(1)$ because we only need to check 2 positions.
- Deletions take $\Theta(1)$ because we only need to check 2 positions.
- To insert $k \in \mathcal{U}$, try $h_1(k)$, if the slot is empty put k there, if the slot contains k', kick out the k', k stay there, and k' repeats the behavior of k on T_2.
- Repeat this process, bouncing between tables, until all elements stabilize.

Cuckoo Hashing: Long cycles of insertion

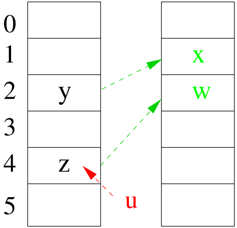
One complication is that the cuckoo may loop for ever. The probability of such an event is small. In such a case choose an upper bound in the number of slot exchanges, and if it exceeds, do a rehash: choose new functions and start .

Example: We have $\{x, y, w, z, u\}$
$h_1(x) = 2; h_1(y) = 2; h_1(w) = 4; h_1(z) = 4, h_1(u) = 4$
$h_2(x) = 1; h_2(y) = 1; h_2(w) = 2; h_2(z) = 0, h_2(u) = 2$



Cuckoo Hashing: Long cycles of insertion

What happens if
$h_1(x) = 2; h_1(y) = 2; h_1(w) = 4; h_1(z) = 4, h_1(u) = 4$
$h_2(x) = 1; h_2(y) = 1; h_2(w) = 2; h_2(z) = 0, h_2(u) = 2$?



If insertion gets into a cycle, we perform a rehash: choose new h_1, h_2 and insert all elements back into the table.

Cuckoo Hashing: An example

We wish to hash the set of keys:$(20, 50, 53, 75, 100, 67, 105, 3, 36, 39, 6)$ using $h_1(k) = k \mod 11$ and $h_2(k) = \lfloor \frac{k}{11} \rfloor \mod 11$.

	h_1	h_2
20	9	1
50	6	4
53	9	4
75	9	6
100	1	9
67	1	6
105	6	9
3	3	0
36	3	3
39	6	3
6	6	0

	T_1	T_2
0		
1	100	20
2		
3		
4		53
5		
6	50	
7		
8		
9	75	
10		

Cuckoo Hashing: An example

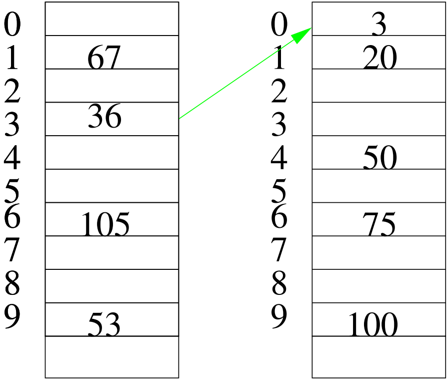
	h_1	h_2
20	9	1
50	6	4
53	9	4
75	9	6
100	1	9
67	1	6
105	6	9
3	3	0
36	3	3
39	6	3
6	6	0

0	
1	67
2	
3	
4	
5	
6	105
7	
8	
9	53

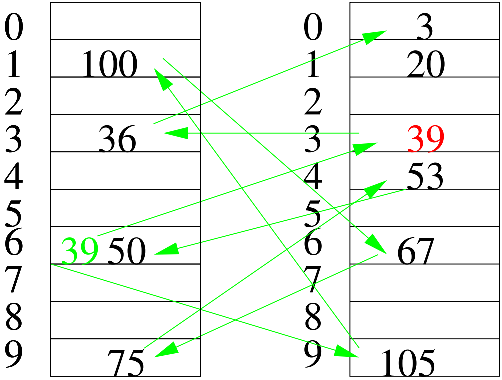
0	
1	20
2	
3	
4	50
5	
6	75
7	
8	
9	100

Cuckoo Hashing: An example

	h_1	h_2
20	9	1
50	6	4
53	9	4
75	9	6
100	1	9
67	1	6
105	6	9
3	3	0
36	3	3
39	6	3
6	6	0



	h_1	h_2
20	9	1
50	6	4
53	9	4
75	9	6
100	1	9
67	1	6
105	6	9
3	3	0
36	3	3
39	6	3
6	6	0



With 6 we have to rehash!!!

Complexity

Cuckoo hashing has a complexity:

- *Search an element x:* constant worst case complexity (x only can be in the 2 positions $h_1(x)$ or in $h_2(x)$)
- *Delete an element:* constant worst case complexity (look at the 2 positions and erase the element)
- *Inserte an element:* expected constant complexity.

Analyzing Cuckoo Hashing

- Cuckoo hashing is tricky to analyze:
  - Elements move around and can be in one of two different places.
  - The sequence of displacements can jump chaotically over the table.
- The framework for analyzing cuckoo hashing requires analysis on random bipartite graphs and random graph processes.

Analyzing Cuckoo Hashing

- The cuckoo graph is a bipartite graph derived from a cuckoo hash table.
- Each table slot is a node.
- Each element x is an edge from $(h_1(x), h_2(x))$

Analyzing Cuckoo Hashing

- An insertion traces a path through the cuckoo graph.
- An insertion of x succeeds iff the connected component containing edge x contains at most one cycle.

Analyzing Cuckoo Hashing

- Analyze the probability that a connected component has more than one cycle.
- Under the assumption that no connected component has more than one cycle, analyze the expected cost of an insertion.
  The cost of inserting x into a cuckoo hash table is proportional to the size of the CC containing x.

Analyzing Cuckoo Hashing

Theorem

If $m = (1+\epsilon)n$, for some $\epsilon > 0$, the probability that the cuckoo graph contains a connected component with more than one cycle is $O(1/m)$.

Theorem

If $m \geq (1+\epsilon)n$, for any $\epsilon > 0$, the expected number of nodes in a connected component of the cuckoo graph is at most $1 + 1/\epsilon$.

So, expectec time of insertion is $O(1)$

Analyzing Cuckoo Hashing

- The time for insertion is $1 + 1/\epsilon$.
  The expected cost of a single rehash, assuming that it succeeds, is $O(m + n/\epsilon)$.
- As a rehash succeeds with probability $1 - O(1/m)$, on expectation, only $1/(1 - O(1/m)) = O(1)$ rehashes are necessary.
- The expected cost due to rehash is $O(m + n/\epsilon)$.