Lecture 09/07/23

Scribed by: Ben Leebron (bhl2), John Lynch (jbl5), Daniel Li (djl15)

Bloom Filters

- Referring back to our previous problem of a malicious URL detector for Chrome
    - Given a database of 1 million malicious URLs, storing all of them with the browser is prohibitively expensive (50mb)
    - We need a local solution under 2 MB
- We discussed membership checking, i.e. you are given a set and you are questioned whether or not an element e belongs in a Set, for this problem whether or not a URL belongs in the set of malicious URL. If we store all the objects and use all the memory, we could do it, but the data is too ‘heavy’. How can we do membership testing efficiently?

The question we are trying to answer: is our URL e in our set of malicious URLs S?

Review of Hashing + Example

Okay, here’s a potential solution:

- Let’s hash every string and put it in a bitmap (an array of 1’s and 0’s).
- Given a query q, if $h(q) = 1$, we’ve “seen” q, otherwise we haven’t
- Chance of reaching a false positive assuming there are N strings?
    - $< 1 - (1 - \frac{1}{R})^N$, where R is the size of the bit map
    - This is still a pretty high rate. Say we have a bitarray of size $2N$, we have a false positive rate ~50%
- At the end of the day, false positives are inevitable, we are simply trying to minimize the probability of a false positive
- The false positive rate is the resource, and we want to minimize it but understand the No Free Lunch principle

- Chance of false positive goes down exponentially as you double, triple your resources; a linear cost for exponential gain

A better solution: Bloom filters

This relies on the “Power of K choices” principle. Let’s use k-independent hash functions instead of just 1

- All we need to get k-independent hash functions is to select our seeds independently.
- Using the same bitmap principle, membership checking is now just seeing that *all* of our k hashing functions are set in the bitmap.

Example

$$h_1(x) = x \ mod \, 5, h_2(x) = (2x + 3) \ mod \, 5$$

To start, we have an empty bitmap:

0	**1**	**2**	**3**	**4**
0	0	0	0	0

Then, lets insert 3 into the bloom filter:

$h_1(3) = 3, h_2(3) = 4$

So, we set 3 and 4 in the bitmap:

0	**1**	**2**	**3**	**4**
0	0	0	1	1

Now, if we query for 1, we can see that $h_1(1) = 1, h_2(1) = 3$. Even though 3 is in the bitmap, 1 is not. So, we can say with certainty that we have not seen 1.

Like with using a single hash function, there are no false negatives. If the query was inserted before, bloom filters always return true.

However, it can return true for an element which was not inserted (False positive)

A bit (haha) of analysis

- Given N strings are inserted:
  - $P(a\ bit\ is\ not\ set) = (1 - \frac{1}{R})^{KN}$
  - $P(a\ bit\ is\ set) = 1 - (1 - \frac{1}{R})^{KN} Pr(a\ bit\ is\ set) = 1 - (1 - \frac{1}{R})^{KN}$
  - Again, R is the size of the bitmap, K is the number of hash functions
- Probability that all the bits $h_1(q), h_2(q), \cdots, h_K(q)$ is set WITHOUT seeing q (false positive):
  - $(1 - (1 - \frac{1}{R})^{KN})^K \approx (1 - e^{\frac{KN}{R}})^K$
  - This is the rate that we want to make as small as possible.
  - Minimized at $K = ln(2) * \frac{R}{N}$. If R is say 10N, then K = 6.9 or 7
  - Optimum false positive is approx. $0.618^{\frac{R}{N}}$ which is < 0.008 given the above numbers
  - Compare that with 0.1 at k =1
- So with N strings we need 10N bits space. Compare with hash table of N * 8 * 50 → A reduction of 40x in memory
- N - can't really control this number. But we can control R and K.

Generic Set Compression

- Given a set *S of* n *objec*ts with each object being heavy such as strings, etc.
- Bloom filter can compress S to less memory around 10 bit per object and still answer membership queries efficiently with rare chances of false positives
- It can up updated dynamically on the fly
- How about deletions?
  - One of the malicious URLs is not malicious anymore!
  - Deleting is disallowed because unsetting these bits can create false negatives for other functions!

Bloom Filters In The Wild (Source: Wikipedia)

- All over the place. Literally everywhere

- Any CDN using caching, e.g. Akamai
    - If an object is only requested once, it shouldn't be cached. Akamai uses a bloom filter to keep track of the first request. Then, if its requested again, its cached.
    - Single-hit requests make up about 66% of their requests
- Database systems use them to reduce disk lookups for nonexistent rows.
- Chrome, obviously, to lookup malicious URLs
- Squid web proxy cache uses bloom filters for cache digets
- Bitcoin uses bloom filters to speed up wallet synchronization
- The venti archival storage system uses Bloom filters to detect previously stored data
- The SPIN model checker uses boom filters to track the reachable state space for large verification problems
- The Exim mail transfer agent (MTA) uses bloom filters in its rate-limit feature
- Ethereum uses bloom filters for quickly finding logs on the Ethereum blockchain
- Cascading analytics framework using bloom filters to speed up asymmetric joins, where one of the joined datasets is significantly larger than the other set. This is often referred to as a bloom join.

Deletion

Deletion: Option 1

- Use two bloom filters
    - One to keep track of added elements
    - One to keep track of deleted elements
- What are the chances of false negatives?
    - Yes. You could hit a false positive in the deleted filter, thus making a false negative
- What are the chances of false positives?

- They have actually decreased a bit, since the false negative rate has increased

A popular alternative

- Counting filters
  - Fan, Li; Cao, Pei; Almeida; Jussara; Broder, Andrei (2000), “Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol”
  - Basically, instead of using a bitarray, use, e.g. a byte array to store a counter. +1 when inserting an item, -1 when removing.

Union of two bloom filters?

- Given Bloom filter B_1 for set S_1 and another bloom filter B_2 of set S_2 with same hash functions.
- What is the bloom filter of $S_1 \cup S_2$?
- Let’s say you Region 1 has a bloom filter with n objects and Region 2 has a bloom filter with m objects, when you take the union you have just increased the chance of false positive. You will never know what n is in practice. I will build a bloom filter for as much as I can allocate.
- Bloom filters can be organized in distributed data structures to perform fully decentralized computations of aggregate functions. Decentralized aggregation makes them ideal for several application by avoiding costly communication.
- Bloom filters must be of same size and same hash function.

Shrink size of bloom filters?

- Can we shrink the bloom filter to half its size?
  - I.e. we don’t know what was inserted into our current one and want to shrink it
  - We can just modify the hash function: $h'(x) : h(x)\ mod\ 2^{k-1}$, where k is the previous number of bits of the hash.
  - Then, we can just chop off the MSB of everything currently in the set. (That is, just move every element in the bit array to half its current index)
- How about doubling its size (without inserting more)?

    - That would reduce the false positive rate- repeated halving of the size means FP rate approaches zero, allowing us to recover the whole set
    - This is obviously impossible, so no, we can’t double the size of a bloom filter

Weaknesses of Bloom filters

- Big limitation: we are relying on having independent k hash functions
- Uses 1.44x more space than we theoretically should

Food for thought: sharing friends on Facebook

- Do two people share friends on Facebook?
- We are only given a compressed bloom filter of their friend network graph
    - Example: person A has 5 friends, those 5 people are hashed into their own network graph, so, if hash functions $h_1(x), h_2(x), \cdots, h_k(x)$ return 1, person A is friends with person x

Notes for HW1 Problem 3

- Computing $E[x_1 + x_2 + x_3]$ is easy
- $E[(x_1 + x_2 + x_3)^2]$ is a little harder. But we can use expansion to get
- $E[\sum x_i^2 + \sum x_i x_j]$. If the x_i and x_j are pairwise independent, and their expected value is 0, then those terms go away and we just get $E[\sum x_i^2]$. This can be extended to the 3rd, 4th, etc. moments