**COMP 480/580 — Probabilistic Algorithms and Data Structure** Sep 17, 2024

Lecture 6

*Lecturer: Anshumali Shrivastava* *Scribe By: Theodoros Tyrovouzis, Joshua-James Claybon*

# 1 Bloom Filters

## 1.1 Motivating example problem

Consider the problem of creating a malicious URL detector on the browser. There are a million known malicious URLs with an average length of 50, making the entire dictionary $\approx$ 50MB in total. However, the implementation must abide by the following requirements:

- It must be $<$ 2MB of size locally
- It must be fast, requiring no remote requests *almost* always.

If we used a normal hash table, we cannot store 50MBs of strings even after compression, so we need to find a better algorithm.

More generally, we need an algorithm with fast checking of whether an element exists, no false negative answers (return not contained when it is), and give a false positive answer (return contained when it actually is not) with very low probability.

## 1.2 Bitmaps

The simplest option would be to consider a bitmap, since we only care about checking for existence and do not need to store the actual string values. A bitmap is a function $h : S \to \{0, 1\}$ that maps the input strings/objects to a single bit (zero or one). For the mapping itself, we can use a universal hash function. In that case, the collision rate is:

$$\Pr(Collision) \leq 1 - (1 - 1/R)^N$$

where R is the size of the bitmap and N is the number of strings inserted into the bitmap.

## 1.3 False Positives and False Negatives

A false positive is defined as returning a result when the key is not in the map. This can happen when multiple elements hash to the same set of bits (i.e. all of an element's corresponding bits could be set to 1, even though it was never inserted. A bloom filter can guarantee no false negatives (i.e. if bloom filter specifies that an element is *not* in the set, it is definitely not in the set). This is because all bits corresponding to an inserted element are set to 1, thus the filter never represents it as absent.

## 1.4 Bloom filters

In an attempt to reduce the collision rate further, we can use $K$ independent hash functions. Each input string is hashed by each hash function and sets the corresponding bit. In order for an input string to be considered contained in the set, **all** hash functions must return 1. This method is called a *bloom filter*.

The probability that a bit is *not* set by one hash function after a string is inserted is

$$\Pr(\textit{single bit not set}) = 1 - \Pr(\textit{single bit set}) = 1 - \frac{1}{R}$$

assuming the probability that each bit is set throughout the bitmap is uniform.

The probability a bit is *not* set given that N strings were inserted is:

$$\Pr(\textit{not set}) = \left(1 - \frac{1}{R}\right)^{KN}$$

since the bit has to not be set for each of the K hash functions and each of the N strings.

Therefore, the false positive rate, which is the probability that all bits returned from each hash function were set at the time of testing is:

$$FPR = \prod_{i=1}^{K} FPR_i = \prod_{i=1}^{K}(1 - \Pr(\textit{not set})) = \left[1 - (1 - 1/R)^{NK})\right]^K \approx \left[1 - e^{-NK/R})\right]^K$$

where the last approximation is true for large R.

We generally want to find the optimal K that minimizes the false positive rate. The FPR becomes minimized at:

$$K = \log(2)\frac{R}{N}$$

in which case the false positive rate is

$$0.618^{R/N}$$

For example, if we allocate 10 bits for each string to be inserted ($R = 10N$), the optimal K would be $\approx 7$ and the probability of a false positive would be less than 0.8%

A nice property of bloom filters is that any string that was inserted at initialization is guaranteed to return 1, so there are no false negatives. On problems such as the one mentioned, false positives are fine as long as they are rare enough. For malicious algorithm detection, if the bloom filters returns true for a URL, the browser shall make a remote request to a server for a precise calculation before concluding that the URL is malicious due to the possibility of a false positive.

## 1.5 Properties

- They are fast
- Unlike traditional hash tables, bloom filters do not require storing entire objects and can answer queries precisely with as low as 10 bits per object
- Bloom filters support addition
- Bloom filters *do not* support deletion by default
- For this reason their use is ubiquitous in caches

### 1.6 Extensions

- In order to support deletion, we could use two bloom filters. One keeps track of all the added elements and one keeps track of all the deleted ones. If an element is found in both added and deleted bloom filters, that means that it has been deleted and can be ignored. This also reduced the probability of false positives.

- The union of two bloom filters is exceptionally easy to compute. Simply bit-OR the two bit arrays. This makes them an easily distributed data structure which is very convenient for web caches.