COMP 480/580 — Probabilistic Algorithms and Data Structure Jan 22, 2019

Lecture 5

Lecturer: Anshumali Shrivastava *Scribe By: Spencer Chang*

1 Problem: Memory Constraints

To see why we have a need for bloom filters, consider the problem we saw in the first lecture: if we have need to make a malicious URL detector but have a memory constraint and cannot hold a completely accurate mapping in memory, how can we do better? The usual trade-off we have is that we can do better in space at the cost of computation and accuracy. This problem can be broken down as follows:
Consider the problem of checking whether a query is malicious with the following circumstances:

- h: strings $\rightarrow$ [0-999]
- Given 100 queries (average 50 length), we would need a total space of 100 x 50 x 8 = 40000 bits for a complete mapping
- The key question is what can we do with 1000 bits?

2 Solution Approach

Consider a simple solution involving combining a bit map with universal hashing. For example, let's try to solve this using a hash function to map the input query into a location in a simple bit map, where if $h(q) = 1$ for a particular query, it is malicious. In order to insert a query into the mapping, we simply hash the query and set it's corresponding hash location to 1. From this we can make two observations:

1. This scheme **cannot** produce false negatives. This is true by virtue of the fact that if a query is inserted into the bit map, the hash value of that query will always be the same, and so the corresponding hash location will always be set after the insertion
2. This scheme **can** produce false positives because query A, a query that has not been inserted, could happen to hash to the same location as query B, which has already been inserted in the mapping, which would produce a false result of saying that query A is in the bit map.

We can quantify the false positive rate as follows, taking N for the number of queries entered and R for the size of the bitmap: $FP_{rate} < 1 - (1 - \frac{1}{R})^N$ (assuming complete independence between the N queries)
Proof:

- $\frac{1}{R}$: probability for one bit location being set
- $1 - \frac{1}{R}$: probability for bit location not being set
- $(1 - \frac{1}{R})^N$: probability for bit not being set for all N insertions

- $1 - (1 - \frac{1}{R})^N$: probability that bit location is set after all N insertions

Thus, the probability that we have a false positive for a given query, q, after all N insertions is at most $1 - (1 - \frac{1}{R})^N$. In the case of $n = 100, R = 1000, FP_{rate} < \frac{1}{10}$. As we can see, a false positive rate of 10% is not ideal because that means one out of every 10 queries will result in a incorrect membership check. Like we did in chaining, the way to decrease this error is to use more hashes. The solution is a Bloom Filter.

3 Bloom Filters

3.1 Basic Idea

A Bloom Filter expands on this simple approach by using the same intuition we did in chaining by adding more hashes. Bloom Filters solve the problem of an efficient membership query (or a cache) because caches still work with false positives, since you can incur an extra cost to actually check if it is positive. The guarantee of no false negatives is really helpful for caching problems because we want to ensure low memory constraint, fast access, and accuracy. The idea is as follows:

- Use K different independent hash functions (selecting seeds independently)
- Given query q, if $h_i(q) = 1 \forall i, 1 \leq i \geq K$, return 1, else return 0

3.2 False Positive Rate

In this case, what is the false positive rate?

$$Pr[\text{1 bit not set} \mid \text{N strings inserted}] = \big(1 - \frac{1}{R}\big)^{KN}$$

$$Pr[\text{1 bit set} \mid \text{N strings inserted}] = 1 - \big(1 - \frac{1}{R}\big)^{KN}$$

$$Pr[\text{all k bits set without seeing q} \mid \text{N strings inserted}] = \Big(1 - \big(1 - \frac{1}{R}\big)^{KN}\Big)^K$$

3.3 What problem does this solve?

Bloom filters solve the problem for generic set compression. This is really important when you have to deal with a set of elements that store a lot of information (strings, complex objects, etc.). A bloom filter can compress such a set, S, to a lot less memory (10 bits per element) and answer membership queries efficiently and effectively. They can also be updated dynamically on the fly if the truth of membership changes in real-time.

3.4 Handling Deletions

Deletions are hard to handle with bloom filters because we can't simply delete the bits of the various hash locations associated with the desired query to delete since those locations could correspond to other queries that were inserted as well.

3.4.1 Option 1: Two Bloom Filters

1. First bloom filter to keep track of added elements, BF_0
2. Second bloom filter to keep track of deleted elements, BF_D

Insertion stays the same, but when you want to delete an element, simply set the corresponding hash bits in BF_D. On membership check, first check BF_0, and if BF_0 returns true, check if BF_D returns false, then return true.
Q: What is the chance of false negatives?
This solution does introduce the possibility of false negatives. Take a query, q, that has actually been inserted and not deleted. On membership check of q, there is a chance of running into a false positive with checking for it in BF_D (after correctly checking that it is in BF_0 because there is still no chance of false negative for that stage), in which case it would return false for the membership check, believing that it has been deleted.
Q: What is the chance of false positives?
Compared to our previous analysis of the false positive rate, this solution has a lower chance of false positives. Take a query, q, that has not been inserted. On membership check of q, there could be a false positive on checking BF_0 but then a following false positive on checking BF_D, which leads to a correct return result of false.

3.4.2 Option 2: Counting Filter

Instead of a simple bit array, the bloom filter maps to a array of counts. On addition, you increment the associated locations that the object hashes to and decrease them on deletion. Like the previous solution, this has a chance for both false positives and false negatives.

3.5 Manipulating Bloom Filters

3.5.1 Union of Bloom Filters

Given B_1 for set S_1 and B_2 for set S_2 with the same hash function, H, what is the bloom filter of $S_1 \cup S_2$?
This can be visualized as the OR of B_1, B_2. Given $S_1 \xrightarrow{H} B_1$ and $S_2 \xrightarrow{H} B_2$, $S_1 \cup S_2 \xrightarrow{H} B_G$, and $B_G = B_1 || B_2$ because B_1 contains the information of whether any element in S_1 is present and the same for B_2 and S_2. Thus, $B_1 \cap B_2$, will give the results for whether an element is present in $S_1 \cup S_2$.
Because of this property, bloom filters can be organized in distributed data structures and aggregated using the OR operation instead of any costly communication for synchronization.

3.5.2 Changing Size

Shrink size? Yes
Use a mod function on the hash to make it map into $\frac{1}{2}$ the prior range, and apply an "OR" function on the first half with the second half of the original bit map to map it into half the size.
Double size? No
This would mean you could do this indefinitely and get no error. There is no way to map the original bit map into double the size since you have not preserved any of the original information from the insertions.

3.6 Weaknesses

1. Requires fully independent hash functions
2. The space usage is still 1.44x more than the theoretically best possible for information storage
3. Dynamically growing size is hard, and the best size depends on the false positive rate and number of insertions (which you must determine a priori)