MMN13 Proposal

Introducing the Bloom filter

A Bloom filter is a set-like data structure that is highly efficient in its use of space. It only supports two operations: insertion and membership querying. Unlike a normal set data structure, a Bloom filter can give incorrect answers. If we query it to see whether an element that we have inserted is present, it will answer affirmatively. If we query for an element that we have *not* inserted, it *might* incorrectly claim that the element is present.

For many applications, a low rate of false positives is tolerable. For instance, the job of a network traffic shaper is to throttle bulk transfers (e.g. BitTorrent) so that interactive sessions (such as **ssh** sessions or games) see good response times. A traffic shaper might use a Bloom filter to determine whether a packet belonging to a particular session is bulk or interactive. If it misidentifies one in ten thousand bulk packets as interactive and fails to throttle it, nobody will notice.

The attraction of a Bloom filter is its space efficiency. If we want to build a spell checker, and have a dictionary of half a million words, a set data structure might consume 20 megabytes of space. A Bloom filter, in contrast, would consume about half a megabyte, at the cost of missing perhaps 1% of misspelled words.

Behind the scenes, a Bloom filter is remarkably simple. It consists of a bit array and a handful of hash functions. We'll use *k* for the number of hash functions. If we want to insert a value into the Bloom filter, we compute *k* hashes of the value, and turn on those bits in the bit array. If we want to see whether a value is present, we compute *k* hashes, and check all of those bits in the array to see if they are turned on.

Idea Proposed in this exercise

The Bloom Filter’s algorithm work flow is defined by turning on the bits when implementing the hash function according to the result.

Since the number of bits presented in the Bloom Filter is finite, at some point there will be parallels between the significant bits that are turned on as a result of valid inputs that are different from the values in the dictionary.

This problem is called the FALSE POSITIVE.

If we were to define the entries in the dictionary as M, the number of elements (bits) as N and the position in the array as K, we can define the probability of getting a FP:

$$\sum_t \Pr(q=t)(1-t)^k \approx (1-E[q])^k = \left(1-\left[1-\frac{1}{m}\right]^{kn}\right)^k \approx \left(1-e^{-kn/m}\right)^k.$$

In order to solve this problem, we suggest taking two sets of data, applying a bloom filter to each one and then combining them into one filter in order to find out and prevent multiplications between both sets.

The solution, though costs twice as much for the dictionaries to be implemented, however it doubles the possible password pool size and reduces by half the probability of receiving a FP.

Definition & Requirements

APPLICATION

The application shall:

- Allow selection of bloom filter parameters
    - N – size of the bit array
    - K – the number of the hash functions to be used by the filter
- Allow reset of bloom filter with the inserted parameters
- Show current status of the bloom filter
  Show number of passwords inserted
    - Show characteristics of the filter (i.e. N, K)
- Allow insertion of a new password to the filter,
    - Indicate the number of collisions, in other words, how many hash functions resulted in values that were already in the filter.
      NOTE: *if all hash values were in the filter, the result is a **false positive!***
- Provide sanity check on values of N, K (i.e. that N will not be less than K, that both are positive integers etc)

Technologies

- Language: Python

[1]https://en.wikipedia.org/wiki/Bloom_filter#Avoiding_false_positives_in_a_finite_universe