Bloom Filters

Dylan Finazzo, Kyler Kopacz, and Jack Fergus

What is a bloom filter?

Bloom filters are probabilistic data structures that are used to determine if an element is contained in a given set. Although this can be achieved by many different data structures, bloom filters have the advantage of being extremely space efficient compared to other data structures that accomplish the same task (tries, hash tables, balanced trees, linked lists, etc.). This is because bloom filters don't have to store the elements inside of themselves. Bloom filters instead use bit sets and hashing to get a representation of the data in a much more compact form. The trade off for such an improvement in space efficiency is the chance that we get a false-positive when querying the data. With bloom filters there is never a chance for a false negative, that is, a bloom filter will never return false when an element is actually contained in the set added to the filter. However, there is a chance of a false positive. There are cases in which a bloom filter will return true even when an element was not part of the set added to the bloom filter. The probability of a false positive varies based on several factors that will be discussed shortly. Given the properties of bloom filters, they are most likely to be useful for tasks that require fast lookups on large data, as the greatest advantage of a bloom filter is its ability to minimize the memory used to implement it. Further, bloom filters are best fit for tasks in which a false positive is not a huge danger to the program, as there is always a chance of a bloom filter returning true when the element that was looked uo is not actually in the dataset.

Applications

There are many applications for bloom filters. One of the more commonly used ones is spell-check. A standard dictionary can be hashed into the bit array, and if a word is spelled incorrectly, the membership test will return negative. Another common use is checking availability of usernames. When a new username is created it is hashed into the bit array, and when someone tries to create a new username it is tested against the array to see if it has already been taken. Google Chrome uses bloom filters to check if a URL could be harmful.

How they work

Bloom filters are data structures composed of an array of bits and a number of hash functions. Instead of storing the elements of a dataset like a hash table does, a bloom filter only stores bit values (0 or 1), allowing this data structure to store information about data while taking up minimal space. There are two operations associated with bloom filters, insertion and query. Data that is inserted into a bloom filter is hashed by any number of hash functions so that later on we can lookup whether or not a data element is contained in the set passed into the bloom filter.

Hashing into a bit vector

Hash functions are the most important aspect of a bloom filter. These functions take in a data element and output an int that refers to an index of the bit array. Any hash functions that maps inputs to integers within the range of the array can be used in bloom filters, however hash functions that are fast and independently and identically distributed are best. Some commonly used and efficient hash functions are murmur, the fnv series of hashes, and HashMix. All of these hash functions are both simple and fast enough to be used in bloom filters while satisfying the iid property necessary for the bloom filter to be efficient.

For each hash function, a data element will be associated with one slot in the bit array. This means that in a bloom filter with three hash functions, a data element will cause up to three bits to be set to 1. However, there is a chance, although unlikely for large bit arrays and efficient hash functions, that different hash functions can output the same index for the same data element. That is, it is possible that $h_1(x)$ and $h_2(x)$ both output a while $h_3(x)$ outputs b, so element x is only associated with two bits in the bit array, A[a] and A[b]. Of course, it is also possible that hash functions will output the same numbers for different elements occasionally, and this is where we run into the problem of false positives.

```
insert(x):
    for i = 1, … , k
        Set $A[h_i(x)] = 1$
```

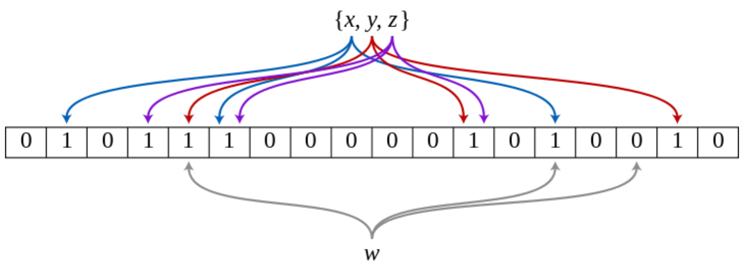
*k = number of hash functions and A is the bit array

Test for membership

To test for membership in a bloom filter, the element in question is put through the bloom filter's hash functions and the associated outputs are returned. If the bits at the indexes of all the outputs are all equal to 1, then it is said that the element is in fact in the set. If any of the bits are 0, then it is said that the element is not in the set. False positives occur when the outputs of all hash functions of an element not included in the set correspond to indexes in the bit array that are set to 1, which happens by chance.

```
lookup(x):
    for i = 1, ... , k
        if $A[h_i(x)] = 0$
            Return false
    Otherwise return true
```

Example



In the above example, we have a bloom filter with bit set size (m) = 18, with the number of hash functions (k) = 3. The elements that we want to add to the set is {x, y, z}. We take each element and run them through the three hash functions to acquire indices that correspond to each element. We then go to those indices in the bit set, and set the value in each index to 1. After doing that for all 3 elements, we have completed the insertions.

Next, we can use the bloom filter to query if an element is in the set or not. In this case, we are querying for the element *w*, which is not contained in our set. We run the element through the 3 hash functions and acquire 3 indices that correspond to that element. Next, we check those elements in the bit set. If we check an index and see that there is a 0 there, we know that we have never seen the element before, because all indices for hashed elements are set to 1. We can see that the third index that we check is set to 0, so we know that *w* is absolutely not in the set of hashed elements. We would return false, indicating that we have not seen *w*.

Parameter Decisions

We can use the following to calculate ideal parameters for a bloom filter:

- *m* = size of bit array
- *k* = number of hash function
- *n* = expected number of insertions

- *P* = probability of false-positive

To find P, we can use:

$$P = (1 - [1 - \frac{1}{m}]^{kn})^k$$

If we know the expected number of inputs and our desired false positive probability, we can deduce the necessary size of our bit array by the following formula:

$$m = -\frac{n * ln(P)}{ln(2)^2}$$

Finally, if we now know the size of our bit array, and we know the expected number of inputs, we can find the optimal number of hash functions to use by the following formula:

$$k = \frac{m}{n} * ln(2)$$

Performance Analysis

Bloom filters have a run time complexity of O(*k*), where k is the number of hash functions used. This is due to elements being ran through *k* amount of hash functions, and *k* number of bits being set in the upon each insertion. When querying for an element, the runtime is also O(*k*) because the algorithm checks at most *k* bits. Additionally, bloom filters have a constant space complexity because the bit set does not grow as the number of inserted elements increases.

The probability of returning a false-positive can be calculated by choosing a different number of hash functions and/or changing the amount of bits in the bit set. However, a bloom filter with 1% error and an optimal value of k requires only about 9.6 bits per element, regardless of the size of the elements. This means elements such as strings, which can be quite large, can be represented in a much more compact manner using a good bloom filter.

Research questions we want to study

There are two main questions we are interested in. Our goal is to determine how changing the number of hash functions affects the rate of false-positives, and how varying the size of the bit set affects the rate of false-positives. We will do this by simulating Bloom filter insertion and membership testing for several different values of k (number of hash functions) and n (size of A, the bit array), and count the number of false positives for each case.

Works Cited

- Wikipedia contributors. (2019, March 15). Bloom filter. In *Wikipedia, The Free Encyclopedia*. Retrieved 17:28, April 26, 2019, from https://en.wikipedia.org/w/index.php?title=Bloom_filter&oldid=887852703
- https://www.coursera.org/learn/algorithms-graphs-data-structures?authMode=signup
- https://llimllib.github.io/bloomfilter-tutorial/
- https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/