Bloom Filters

1 Introduction

A Bloom filter is a probabilistic data structure that tells you whether or not an element is a member of a set. It is termed as probabilistic because it provides you with an approximate answer, not a precise one. In the case of a Bloom filter it will either tell you that an element is possibly in the set, or definitely not in the set. In other words, a query can return false positives but it cannot return false negatives. The probability of false positives can be determined before the Bloom filter is created.

1.1 Applications

The main application of a Bloom filter is to allow someone to quickly check if an element is in a data set. If the Bloom filter says that an element is not in the set then no more work has to be done, this is definitely true, however if it says that an element is possibly in the set then a disk read will have to be made to determine if this is correct. It does however make the search process a lot quicker than simply doing a disk read straight away.

Elements can also be added to a Bloom filter after it has been created, but this will increase the chance of getting false positives when querying an element.

2 Analysis

Bloom filters have a number of advantages over other data structures that have the same purpose, such as binary search trees and hash tables.

2.1 Space advantage

Since Bloom filters don't store the elements themselves, the number of bits required for each element is dependant only on the probability of false positives (p), not on the size of the elements. So, for example, a Bloom filter with a 1% chance of error requires only about 9.6 bits per element. This is a huge advantage for sets with large elements, but it does mean that a separate solution must be provided for the storage of the set.

2.2 Time advantage

The time required to either add an element to the Bloom filter or to query an element takes a fixed constant time, O(k). It does not depend on the number of items in the set. This is because both when adding or querying an element all the program has to do is call the k hash functions on the element.

3 IMPLEMENTATION

For my implementation I decided to use the C programming language. I chose an imperative language over Haskell because it would make it easier to implement all the different algorithms associated with Bloom filters, such as adding elements, querying elements or implementing the hash function. I also chose C over Java because C programs are a lot faster to run without all the runtime checks and type checks that Java uses.

3.1 CREATING A BLOOM FILTER

To create the Bloom filter, I first initialised it to be an array of *m* bits, all set to 0.

I also had to define a set of *k* hash functions. Each hash function has to take an element as the input and map it to one of the *m* array positions in the Bloom filter. Since I didn't know how big the value of *k* would be I implemented one hash function that takes an integer (between 0 and k-1) to be the initial hash value.

Each of the elements in the data set is then passed into each of the *k* hash functions (or the one hash function *k* times in this case) and the resulting array positions in the Bloom filter are set to 1. The program then writes the Bloom filter to the disk as a binary file.

3.2 QUERYING THE BLOOM FILTER

After the Bloom filter has been created it can be read back into the program from the disk, and then the querying of an element can be done on it.

To check if an element is in the set, the element is passed into each of the *k* hash functions to get *k* array positions, and if any of the bits at the array positions are set to 0 then the array is definitely not in the data set, because they would have all been set to 1 when it was added to the Bloom filter. If all the array positions are set to 1 then the element is possibly in the array, but there is the possibility that all the array positions were set to 1 by chance when other elements were added, resulting in a false positive.

3.3 ADDING AN ELEMENT

Adding an element to the Bloom filter is done in the same way as when it is first created. Firstly, the Bloom filter has to be read from the disk, then the element to be added is passed into each of the *k* hash functions and the resulting array positions in the Bloom filter are set to 1. The Bloom filter can then be written back to the disk.

4 EMPIRICAL EVALUATION

To test my program, I used the artist data set provided in the lab sessions. The only difference I made was to remove the ID column so that it only had one column – the artist name, this way it was easier to test the membership of an artist in the set.

4.1 CALCULATING OPTIMAL VALUES

Now, knowing the size of the data set (1,848,276 items) I could calculate the optimal values for *m* and *k*. To do this I had to first decide on a value of *p* (chance of false positives), which I decided to set to 0.01.

Here are the equations for calculating the optimal values of *m* and *k*.

$$m = -\frac{n \ln p}{(\ln 2)^2} = -\frac{1848276 \ln 0.01}{(\ln 2)^2} = 17715833.36$$

$$k = \frac{m}{n} \ln 2 = \frac{17715833.36}{1848276} \ln 2 = 6.64$$

Since they have to be integers the values I used in my program were:

m = 17715833

k = 7

4.2 RUN TIME

Considering the size of the data set, the program ran relatively quickly. I timed how long each function took to run on the school Unix servers.

Querying an element and adding an element to the Bloom filter both take constant time, O(k), and so were very quick to execute. Querying took around 0.003 seconds and adding an element took around 0.03 seconds.

Creating the Bloom filter in the first place is what takes up the most time, but in the real world this process would only have to take place once for any data set, so it is not so important how long it takes. For the artist data set it took around 2.3 seconds to create the Bloom filter, which is still pretty quick, but for larger data sets with billions of items this process would take noticeably longer. The time complexity for creating the Bloom filter is O(*n*), where *n* is the number of items in the data set.