Calvin Kranig

COM S 435

TTH 0930 – 1050

PA1

Introduction

This report will detail the methodology behind my work in PA1 as well as report upon the results of various tests.

Methodology in Creating Bloom Filters

BloomFilterFNV

In order to generate k hash functions for my BloomFilterFNV I did the following:

a. For each String the first hash began at the first letter and hashed every letter in the string as dscribed in the FNV hashing algorithm
b. For every ith hash the hashing began at the ith character in the string, and proceeded in the end of the string. Once the end of the string was reached, the hashing continued starting at the beginning of the string and finishing at the ith character.

To give a visual example of what the algorithm is doing consider the string “abc”. Let us say that the number of hashes is 3. Then the following would occur

1. h(“abc”) is entered into the bloom filter
2. h(“bca”) is entered into the bloom filter
3. h(“cab”) is entered into the bloom filter

This is akin to shifting bits in a bit string. I selected this process with the knowledge that bit shifting works for regular hash functions so it should work for this use case. I then tested my new filter and found that the false positive rate was within a small difference from the expected rate.

BloomFilterRan

For BloomFilterRan I started off by randomly generating k pairs of random integers a and b between 0 and p (p = smallest prime greater than the bloom filter size). I then computed each hash with (ax +b)%p.

In order to get x I got the hash code for each String inputted by using the hashCode() method of class String.

False Positive Experiment

Methodology

In order to test each filter my FalsePositives class was designed in the following way:

FalsePositives(int setSize, int bitsPerElement, int testStringNum)

> This method created a new false positives test which would create bloom filters with the given setSize and bitsPerElement. It would then randomly generate setSize strings and add them to

each one of the filters, It also keeps track of all the strings in the filters with a HashSet (This comes into play later).

Then when fp.filterTest() was called on the created FalsePositives instance 3 tests would be run, one each for the three different types of filters. Before running the tests the method will randomly generate testNumStrings such that they are not found in the filters (using the HashSet to see if they had been put in the filters). After randomly generating the strings not found in the filters each bloom filter would be tested against the set of strings and the number of falsepostives would be incremented by 1 if the string appeared inside the bloomfilter. After all the strings had been iterated over the method outputs the ratio of falsepositves to strings tested.

By constructing the test in this way many calls to filterTest can be ran on a set of bloomFilters without having to regenerate the filters. Another bonus of the design is that FalsePositives uses the Filter interface within all its internal methods which allows for easy modification in order for new types of bloom filters to be tested.

FalsePositive Test Results

Filter Test with setSize = 1000 and 1000 strings tested

```
Filter Test
Testing Filters of size 1000 with bits per element 4
BloomFilterFNV:
Ratio: 0.147 Expected: 0.14586594177599999
DynamicFilter:
Ratio: 0.151 Expected: 0.14586594177599999
BloomFilterRan:
Ratio: 0.142 Expected: 0.14586594177599999

Testing Filters of size 1000 with bits per element 8
BloomFilterFNV:
Ratio: 0.015 Expected: 0.02127687297019942
DynamicFilter:
Ratio: 0.017 Expected: 0.02127687297019942
BloomFilterRan:
Ratio: 0.017 Expected: 0.02127687297019942

Testing Filters of size 1000 with bits per element 10
BloomFilterFNV:
Ratio: 0.02 Expected: 0.02127687297019942
DynamicFilter:
Ratio: 0.024 Expected: 0.02127687297019942
BloomFilterRan:
Ratio: 0.017 Expected: 0.02127687297019942
```

Filter Test with setSize = 1000000 and 1000000 strings tested

```
Filter Test
Testing Filters of size 1000000 with bits per element 4
BloomFilterFNV:
 Ratio: 0.155461 Expected: 0.14586594177599999
DynamicFilter:
 Ratio: 0.991266 Expected: 0.14586594177599999
BloomFilterRan:
 Ratio: 0.155129 Expected: 0.14586594177599999

Testing Filters of size 1000000 with bits per element 8
BloomFilterFNV:
 Ratio: 0.024939 Expected: 0.02127687297019942
DynamicFilter:
 Ratio: 0.83975 Expected: 0.02127687297019942
BloomFilterRan:
 Ratio: 0.022355 Expected: 0.02127687297019942

Testing Filters of size 1000000 with bits per element 10
BloomFilterFNV:
 Ratio: 0.02508 Expected: 0.02127687297019942
DynamicFilter:
 Ratio: 0.84149 Expected: 0.02127687297019942
BloomFilterRan:
 Ratio: 0.022062 Expected: 0.02127687297019942
```

The test results were able to show me that my implementation of the filters was producing reasonable results. Out of all the filters the BloomFilterRan consistently had the smallest ratio of false positives. As bits per element increases the size of the Filters increase and the number of false positives decrease. The only outlier between the 3 filters is the dynamic filter. The dynamic filter is implemented as follows:

1. It starts off with a collection of BloomFilterRan's consisting of 1 BloomFilterRan of set size 1000.
2. The dynamic filter adds elements to the last filter in its filter collection until that filter contains n elements (where n = the set size the filter was constructed with).
    a. Once this occurs the dynamic filter resizes by creating a new BloomFilterRan that doubles the overall size of the dynamic filter and appends it to the list of BloomFilterRan's.
3. When checking for elements every BloomFilterRan in the list of BloomFilterRan's is checked to see if it contains the element.
    a. This results in the false positive rate being closer to $x(0.618)^{bitsPerElement}$ where x is the number of filters in the list of BloomFilterRan's.

The only alternative to this method is to keep a secondary list of all the objects entered into the bloomfilter. When resizing you create a new filter and re-add each of these elements to it. Having this list defeats the purpose of the bloom filter itself since it takes up a considerable portion of memory.

The other two filters are typically within a few percentage points of the theoretical predictions. As the size of the tests increase this difference tends to have the created filters reporting slightly more false

positives than the theoretical predictions. This stems from the difficulty in hashing of selecting truly random numbers and receiving an evenly distributed set of hashes.

Statistics Evaluation

100 Runs of Statistics Test on Bloom Filter of set size 100000 with 8 bits Per Element `Statistics Test`

```
Intersect Size Estimate was 0.001813 percent off from real set size
Set Size Estimate was 0.000745 percent off from real set size
```

100 Runs of Statistics Test on Bloom Filter of set size 1000 with 8 bits Per Element `Statistics Test`

```
Intersect Size Estimate was 0.013812 percent off from real set size
Set Size Estimate was 0.006979 percent off from real set size
```

The set size estimates are close enough to their true sizes that the difference is marginal. As the set size decreases the percent difference in estimate increases.

This test was run in the following manner:

1. The estimate was calculated on a random BloomFilterFNV of a certain size
    a. The difference between the estimate and the actual set size was calculated and incremented to a total
2. Step 1 was repeated n times and then the result was calculated to be the total difference/n

BloomDifferential vs. NaiveDifferential

I tested the implementations of BloomDifferential and NaiveDifferential using a stopwatch to see their performance given an evenly distributed set of keys. The results can be seen below:

Testing with Database size of 100,000 and bits per element of 8 with 10% of keys in Difference File

```
Testing BloomDifferential key in DiffFile
 100 keys were found in 638 milliseconds
Testing BloomDifferential key in Database
 900 keys were found in 47305 milliseconds
Testing NaiveDifferential key in DiffFile
 100 keys were found in 441 milliseconds
Testing NaiveDifferential key in Database
 900 keys were found in 53686 milliseconds

Overall Results:
 BloomDifferential Total Time: 47943 milliseconds
 NaiveDifferential Total Time: 54127 milliseconds
```

Ask you can see on my machine the NaiveDifferential is much slower than the BloomDifferential despite having a faster lookup when keys are in the Difference File. Let us now consider the experiment mentioned in the PA description. Here are the details of the experiment:

- Accessing the files in secondary memory takes 1000 milliseconds
- Accessing the files in main memory takes 1 millisecond

For each of file look up above the following will occur for each implementation:

BloomDifferential

- Entry in database or file in difference file
    - 1001 milliseconds to access main memory and secondary memory

NaiveDifferential

- Entry in difference file
    - 1000 milliseconds to access main memory
- Entry in database
    - 2000 milliseconds to access main memory twice.

Looking at the test we ran above where we had a 10% chance of the key being the difference file we can calculate some new times for the look up of 1000 entries.

BloomDifferential: 47943 + 100*1001 +900*1001 = 1,048,943 milliseconds = 1,048.943 seconds

NaiveDifferential: 54127 + 100*1000 + 900*2000 = 1,954,127 milliseconds = 1,954.127 seconds

Even for smaller database sizes using a bloom filter results in considerable speedup:

Testing with Database size of 1,000 and bits per element of 8 with 10% of keys in Difference File

```
Testing BloomDifferential key in DiffFile
100 keys were found in 47 milliseconds
Testing BloomDifferential key in Database
900 keys were found in 734 milliseconds
Testing NaiveDifferential key in DiffFile
100 keys were found in 16 milliseconds
Testing NaiveDifferential key in Database
900 keys were found in 719 milliseconds

Overall Results:
       BloomDifferential Total Time: 781 milliseconds
       NaiveDifferential Total Time: 735 milliseconds
```

Totals with Calculation

BloomDifferential: 1,001,781 milliseconds

NaiveDifferential: 1,900,735 milliseconds

Distributed Join

Let us consider the following experiment. Each string in the table is of at most length s (s bytes). Each line will then have 2*s bytes. Let the sizes of R1, R2, and R3 be denoted by |R1|, |R2|, and |R3|. Let p

be a constant representing the percentage of elements in R2 that will be put into R3 and sent back to server 1. We have |R2|p = |R3|.

Let a be the percentage of unique keys in |R1|.

We can calculate the costs of using BloomJoin vs. a Naïve implementation.

BloomJoin

sending over the bloom filter costs at most: a*|R1|*bitsPerElement

sending over R3 = 2*s*|R2|*p

Total cost = a*|R1|*bitsPerElement + 2*s*|R2|*p

Naïve

sending over R1: 2*s*|R1|

In order for our BloomJoin to be cost effective the following must hold

a*|R1|*bitsPerElement + 2*s*|R2|*p <= 2*s*|R1|

Let bitsPerElement = 8 = 1 byte

a*|R1|+2*s*|R2|*p <= 2*s*|R1|

2*s*|R2|*p <= (2*s – a)|R1|

2*s*|R3| <= (2*s – a)|R1|

|R3| <= |R1| - $\frac{a}{2*s}$

Recall that a can be at most 1.0. So:

|R1| - $\frac{a}{2*s}$≈|R1|

|R3| <= |R1|

This means that as long as the table sent back to server 1 by server 2 is smaller than |R1| using the BloomJoin method is more efficient. Recall that |R3| = |R2|*p where p is percentage of elements of |R2| put into |R3|. Since p <= 1 we have |R3| <= |R2|. If |R2|≈|R1| then using BloomJoin will always have an equal or higher efficiency then the naïve solution.