COMS 535 PA1

Xiaoyun Fu | Raj Gaurav Ballabh Kumar

1st October, 2018

Q1)

BloomFilterFNV Class:

Methods in the class are as follows:
1) BloomFilterFNV(int setSize, int bitsPerElement):
- Constructor for this class
- Creates a Bloom filter using the FNV function that can store a set S of cardinality setSize.
- The size of the filter is approximately (setSize * bitsPerElement).
- The k-FNV functions to be used in the Bloom Filter are generated using the kFNV Class.
- Basic Idea:

```
Use generateKOffsetBasisValues() to generate k OffsetBasis Values
Store each of these k OffsetBasis Values in an array: kOffsetBasisValues[]
For each element in kOffsetBasisValues[]
    Create an object O of kFNV Class
    For O, set instance variables a and b to be used in:
    $h_i(x) = (ax + b)\%p$
    Store the object O in an ArrayList kFNVS<kFNV>
```

Each object O now contains an OffsetBasisValue, FNVPrime value (fixed for each object), and a value of a and b to calculate the final value of the index that will be returned:
$indexReturned = (a * (FNVHashValue) + b)\%p$

2) public void add(String s):
- Adds the string s to the BloomFilter. Method is case insensitive.
- For a string s, the add method calculates the index using each of the k FNVs stores in the ArrayList. It sets each of these indices to 1.
- Also, everytime we add an element to the Filter, we increment the counter numOfElmntsAdded which is later returned as the dataSize of the Filter.

3) public boolean appears(String s):
- Returns true if the String s appears in the Bloom Filter. The method is case insensitive.

4) public int filterSize():
- Returns the size of the filter. The size of the filter is the next prime which is immediately greater than the number of elements to be stored in the Filter i.e. setSize.

5) public int dataSize():
- Returns the value of the numOfElmntsAdded counter which is the number of elements added to the filter.

6) public int numHashes():
- Returns the number of hash functions used. The number of hash functions ahs been chosen to be equal to the value that is mentioned in the PA, i.e. $ln\ 2 * (bitsPerElement)$

7) private void generateKOffsetBasisValues():
- Generates k integers that are used as kOffsetBasis values for generating hash from FNV.
- We can change this value because as is mentioned in the IETF specifications:

> In the general case, almost any $offsetBasis$ will serve so long as it is non-zero. The choice of a non-standard $offsetBasis$ may be beneficial in defending against some attacks that try to induce hash collisions.

- Repeats the following process k times:

```
    Generate a random integer i
    Find the prime that is immediately greater than i
    Store the prime in kOffsetBasisValues[ ]
```

Rationale:
- The basic idea is to try and use Universal Hashing where instead of using a single Hash Function, we try and use a family of hash functions.
- Random values of a and b in $h_i(x) = (ax + b)\%p$ can give some spread but the problem with this method is that if the hash value for string s output from the FNVHash function is same for 2 strings, then all of the k functions that we have defined are going to collide because the input x is going to be the same for all of them. Thus there seemed no practical advantage of using the k hash functions.
- So we needed a way to change the value x being generated for different strings. One way was to mutate the strings s1 and s2 in different ways (for eg. appending different characters to each string), but the implementation was a bit tedious and keeping track of all the different ways in which the string was originally mutated was cumbersome at the least.
- So instead we went with the third and more obvious approach of changing the parameters input to the FNVHash function itself to some random values. The IETF doc mentions that:

> In the general case, almost any $offsetBasis$ will serve so long as it is non-zero. The choice of a non-standard $offsetBasis$ may be

beneficial in defending against some attacks that try to induce hash collisions.

- One possible disadvantage that we could foresee to this was that the offsetBasis values and FNVPrime values probably have been selected to be those exact values only after extensive testing and changing any of those values could only lead to a higher collision count.
- However, once we had set up the experiment and observed the results over multiple iterations for 10^3 to 10^7 setSize values, we concluded that the number of false positives were always around the theoretical limit. Hence any qualms about there being a higher number of collisions were baseless and the method of generating the k hash functions worked perfectly in practice.
- This was the rationale behind using the k different $offsetBasis$ values to generate the k hash functions.

8) private int findPrime(int n):
- Finds the next prime that is greater than n

9) private boolean isPrime(int n):
- Checks if the input number n is indeed prime or not.

kFNV Class:

1) public kFNV(int offba, int size):
- Constructor for this class
- Function has been described above.

2) public int hashV(String str):
- returns the index that will be set to 1 in the Bloom Filter by the input string str
- Algorithm:

```
hash = offsetBasis
for each octetOfData to be hashed
        hash = hash xor octetOfData
        hash = hash * FNVPrime
        index = (a * (hash) + b) % filterSize)
return index
```

BloomFilterMurmur Class

The methods in this class are pretty much the same as in the BloomFilterFNV. The same design ideas have been followed.

BloomFilterMurmur(int setSize, int bitsPerElement) is the Constructor
Creates k random primes that serve as the seed values to the Murmur Hash Function
For each of the k primes, it creates an object of the kMurmur class with instance variables a and b that are finally used to calculate and return the index of the bit to be set in the filter

Rationale:

- There are 2 64 bit implementations of the Murmur Hash Function on the link provided. Along with the String in a Byte array as input, one of them also requires us to provide the seed to the function. The other uses a default seed value to calculate the hash value. Thus it is clear that we can generate different hash values for the same string by simply changing the seed value.
- Using the same logic as the one in the kFNV functions, we went ahead with changing the seed values for the k different Murmur Hash Functions in addition to using random values if a and b for $h(x) = (ax + b)\%p$.
- The experiments confirmed that the method works and gives us the expected number of falsePositives.

Q4) The experiment designed to compute false positives bad your rationale behind the design of the experiment:

FalsePositives Class
The class uses the following methods:

1) public FalsePositives(int k):
- Constructor
- Takes as input k, the size of the Set S to to be created. This is the set that will be added to the Bloom Filter. - Generates k random strings:
Algorithm:
Initialize BloomFilter F
Initialize 2 variables min, max. These represent the minimum and maximum length of the String to be generated.
Initialize counter c that will increment everytime we add a string to the filter F
Initialize a HashSet H that will keep track of unique Strings generated by the Random process
Initialize an Array List A that will contain only those elements that do not occur in set S

```
Populate the Sets F and A: While (F.size() < k)
    Generate a random integer between min and max, say i
    Generate a random String str of length i
    if( ! H.contains(str))
        Add str to H
        if (c < 10)
            add c to F
            c++
        else add c to A
        set c to 0
```

- This method will ensure that the string that is added in A will not be added to F. Also, since for every 10 strings in F, 1 string is added to A, this will give us a very probable values of sizes of set S and set A as might be in practice.

2) public void startExpKFNV()|public void startExpKMurmur()| public void startExpKRan()

- Here F is created using the method being tested. For eg. if startExpKFNV() was called then the Filter was created using the FNV Hash Function and so on.
- For each String in A, say str, we now check if str appears in F.
- We know that any positive that we get is going to be a false positive.
- We count the number of the positives, say falsePositives and divide it by the total number of elements in the Set A. This gives us the approximate probability of getting a false positive.

The rationale is to check the probability for a large size of set F, say something around 10^7. This would mean that the set A would contain 10^6 elements. With this big a set size, we should get a very close estimate of the actual probability.

Q 5) For all the Bloom filter classes report the false probabilities when bitsPerElement are 4, 8 and 10. How do false positives depend on bitsPerElement? Which filter has smaller false positives? If there is a considerable difference between the false positives, can you explain the difference? How far away are the false positives from the theoretical predictions?

- For all experiments Length of strings between 6 and 12 . BloomFilter.size(): 10000000
SetA.size(): 999999

For Bit Size = 4
Theoretical Probability: $0.618^4 = 0.145865941776$
kFNV RESULTS: 0.1545891545891546
k MURMUR RESULTS: 0.15543315543315545
kRAN RESULTS: 0.15536415536415538

For Bit Size = 8
Theoretical Probability: $0.618^8 = 0.02127687297019942203417 6$
kFNV RESULTS: 0.021658021658021658
k MURMUR RESULTS: 0.021684021684021684
kRAN RESULTS: 0.02195002195002195

For Bit Size = 10
Theoretical Probability: $0.618^{10} = 0.0081261484322704440609806 34624$
kFNV RESULTS: 0.008461008461008461
k MURMUR RESULTS: 0.008359008359008359
kRAN RESULTS: 0.008593008593008593

- False Positives exponentially decrease with increase in bitsPerElement.
- There is no one filter that has the lowest probability of error. Although the highest in all 3 experiments was the kRAN filter.
- The difference in the results between (kFNV + Murmur) and (kRAN) is that if the hash values for any 2 strings collides for 1 out of the k hash functions then it means that all of the k hash functions will also end up in collision. hence the false positives are slightly higher in the case of kRAN.

- The number of false positives are surprisingly close to the theoretical prediction.

Q6) Describe how the EmpericalComparison is comparing the performances of BloomDifferential and NaiveDifferential. Explain the rationale behind your design.

BloomDifferential Class
- This class has the following methods:

1) public BloomFilterFNV createFilter():
- We use the FNV Hash Function for this BloomFilter.
- We iterate through each line in the file and then add the Key on each line to the BloomFilter F.

2) public String retrieveRecord(String key):
- Algorithm:

```
if(F.appears(key))
        if DiffFile.contains(key)
        return line
        else Search for key in database
        return line
else Search for key in database
return line
```

3) private String retrieveRecordfromDbase(String myKey):
- Implements the Search for key in database method.

NaiveDifferential Class
- This class has the following methods:

1) public String retrieveRecord(String key):
Algorithm:

```
if DiffFile.contains(key)
        return line
else Search for key in database
        return line
```

2) private String retrieveRecordfromDbase(String myKey):
- Implements the Search for key in database method.

EmpiricalComparison Class

Basic Idea of experiment:

This is what my experiment is going to be:
3 Modes in the experiment

Mode 1: Generate a random Key from the database.txt : grams.size (12632196)
Mode 2: Generate a random key from the DiffFile.txt (1262147)
Mode 3: Generate a key that is not in Diff but is in database.txt i.e.subset.txt: (722)

Note: I have computed subset.txt separatetly for this experiment. The file is attached to the zip folder in case you want to look at it. It contains 720 entries.

Choose a number r randomly between 0 to size of file being considered (mode dependent)
Start a loop to read the file until you reach line number r.
Use counter for counting the number of lines
The key value on your line r is the key value that you are going to search for.
We use the BloomDifferential and find the key. Check time taken.
We use the NaiveDiffernetial and find the key. Check the time taken.
Repeat these steps, say 20 times. Check time difference. Output results.

As expected, the biggest advantage in using BloomFilter is seen when we are in Mode3. Since the Bloom Filter only gives false positives with a very low probability, it is much more likely that the BloomDiffernetial will just skip the DiffFile.txt ans straight away go to the database.txt to search the key. The NaiveDifferential on the other hand has to scan through the entire DiffFile.txt and then through the Database.txt to get the correct record. This time difference becomes more glaring if the key does not occur in the the database.txt at all.

Mode 1 gives somewhat better results compared to Mode 2 because there is a 1/10 chance that the key actually belongs to Diff.txt. This would mean that for those entries, the Bloom and Naive would take the same amount of time. Although the results for Mode 1 are most surprising. When all of the 10 keys are in Database, the Bloom filter takes only 1.9s. But when we allow the chance for some of them to be in DiffFile, the time taken suddenly increases by a very large amount. The time taken by the Naive also increases and still is greater than the Bloom which means our dataStructure is working as expected. It might have something to do with how Windows actually accesses the files in the memory.

Mode 2 is the absolute worst with both methods taking the same amount of time because the key is in the Diff file itself. However we can be pretty sure that when users will actually search for a key, it is very very unlikely that all of the requests are going to be for elements in the diffFile.txt itself.

The same can be seen from the below numbers as well. 10 entries were searched for in each mode. The time taken is in milliseconds.

Mode 1:
Total Bloom Time: 45221
Total Naive Time: 58976

Mode 2:
Total Bloom Time: 9561

Total Naive Time: 9556

Mode3:
Total Bloom Time: 1953 Total Naive Time: 17861