COMS 535

Programming Assignment 1 – Report

Specifications of all classes

# Class: BaseBloomFilter

This class is used as base or super class for all filter. It provides common functionality such as initializing filter, adding in filter, and checking presence in filter etc.

**Class Constructor:** BaseBloomFilter(int setSize, int bitsPerElement)

It takes two parameters (i) size of the set and (ii) number of bits per elements. Using these two parameters, it calculates filter size and number of hashes.

**Methods:**

* **initFilter(int k):** This function will initialize (parameter of this function) number of filters.
* **add(String s):** This method adds a given string in the filter. To do so, it computes hash for the string based on BaseHashGenerator (which is initialize different concrete implementation of hash generators based on bloom filter type) and stores in the filter.
* **appears(String s):** This method checks if a given string is present in bloom filter. Similar to add method, it computes hash first, then checks if all hk is set to true. If not, then it returns false, otherwise true.
* **filterSize:**​ This method returns the size of the bloom filter used.
* **dataSize:**​ This method returns the total number of elements added.
* **numHashes:**​ This method returns the number of hash functions.

# Class Name: BaseHashGenerator

This is a abstract class which is used as base or super class for all hash generator classes. It specifies a template to be implemented by concrete hash generators.

**Methods:**

* **init(String s):** This method initializes hash generator for the given string.
* **getHash(int iteration):** This method returns the hash value, hiteration(s), where s is the string hash function was initialized with.

# Class Name: BloomFilterFNV

This class extends BaseBloomFilter class, thereby inheriting all the properties and functionalities from super class.

**Class Constructor:** BloomFilterFNV(int setSize, int bitsPerElement)

It first calls super constructor with given parameters to initialize common attributes of the filter such as numHashes, filterSize etc. Then, it initializes BaseHashGenerator object with an instance of FNVHashGenerator. It also initializes one filter or table of size filterSize.

# Class Name: BloomFilterRan

This class extends BaseBloomFilter class, thereby inheriting all the properties and functionalities from super class.

**Class Constructor:** BloomFilterRan(int setSize, int bitsPerElement)

It first calls super constructor with given parameters to initialize common attributes of the filter such as numHashes, filterSize etc. Then, it initializes BaseHashGenerator object with an instance of RandomHashGenerator. It also initializes one filter or table of size filterSize.

# Class Name: MultiMultiBloomFilter

This class extends BaseBloomFilter class, thereby inheriting all the properties and functionalities from super class.

**Class Constructor:** MultiMultiBloomFilter (int setSize, int bitsPerElement)

It first calls super constructor with given parameters to initialize common attributes of the filter such as numHashes, filterSize etc. Then, it initializes BaseHashGenerator object with an instance of RandomHashGenerator. It also initializes k (numHash) filters or tables, each of size filterSize.

# Class Name: NaiveBloomFilter

This class extends BaseBloomFilter class, thereby inheriting all the properties and functionalities from super class.

**Class Constructor:** NaiveBloomFilter (int setSize, int bitsPerElement)

It first calls super constructor with given parameters to initialize common attributes of the filter such as numHashes, filterSize etc. It overrides numHashes to one for this bloom filter. Then, it initializes BaseHashGenerator object with an instance of RandomHashGenerator. It also initializes one filter or table, each of size filterSize.

# Class Name: FNVHashGenerator

This class extends BaseHashGeneratorclass, thereby inheriting all the properties and functionalities from super class.

# Methods:

* **hash64:** This function calculates 64-bit FNV hash for a given string. We have used implementation for this function which is available online at <http://www.java2s.com/Code/Java/Development-Class/FNVHash.htm>

# Class: RandomHashGenerator

This class extends BaseHashGeneratorclass, thereby inheriting all the properties and functionalities from super class.

**Class Constructor:** RandomHashGenerator (int numHashes, int filterSize)

It takes two parameters (i) number of hashes and (ii) size of filter. Using these two parameters, it calculates numHashes pairs of (a,b) and a prime greater than filter size.

**Methods:**

* **getCoefficient(Random random):** This function calculates a random number which is 0<c<PRIME.
* **isPrime(int n):** This method tests whether a given integer is prime or not.
* **nextPrime(int N):** This method returns next prime number > N.

# Class: FalsePositives

**Class Constructor:** FalsePositives (int setSize, int bitsPerElement)

It initializes four instances of bloom filters (FNV, Random, Multi, and Naïve).

**Methods:**

* **generateString(int stringLength):** This method generates a random string with given length.
* **expectedRate(int bitsPerElement):** This method returns expected false positive rate for given bits.
* **createFilter(BaseBloomFilter bloomFilter):** It adds setSize number of randomly generated strings in given bloom filter. It also tests if false negative exists in the given filter. After adding a string, it checks whether item there in bloom filter to determine false negative.
* **findFalsePositives(BaseBloomFilter bloomFilter):** It first finds a random string not inserted in given bloom filter already. Then, checks if that string is present in the filter to calculate false positive. It repeats the process numTrials (10000) times.
* **main(String [] args):** Main method of the class which run false positive experiment for different configuration (set size and bits).

# Class: BaseDifferential

This class acts as base or super class for NaiveDifferential and BloomDifferential class.

**Methods:**

* **retrieveRecord(String key):** This method returns a record by consulting differential.txt and database.txt as described in assignment. In case of BloomDifferential, it first checks if the key is present in filter. If not, it doesn’t access differential.txt unlike NaiveDifferential.
* **getType():** This method returns type of the differential. It’s overridden by concrete classes to provide type of bloom filter being used or to indicate it it’s a naïve differential.

# Class: BloomDifferential

This class extends BaseDifferential.

**Methods:**

* **retrieveRecord(String key):** It first checks if the key is present in filter. If not, it doesn’t access differential.txt unlike NaiveDifferential.
* **createFilter(BloomFilterType filterType):** Given a type of bloom filter, it reads keys from differential.txt and create a filter by adding them.

# Class: NaiveDifferential

This class extends BaseDifferential.

**4. Class Name:** ​**FalsePositives** No Constructor **Public Methods:**

**Main method:**

Calculates falsepositive rates for all the three bloom filters using falsepositiverate function with different values of bits per element and set size parameters and outputs them in the console.

**Private Methods:**

**FalsePositiveRate:**

Check each filter for false positives using random data and count the number of false positives for each of the filter. Return type is array of long which has falsepositives for each filter.

**createFilterAndStorage:**

create new random strings and add them to all the three filters. Return type is String array which has newly generated strings.

**generateString:**

Returns a new random string

**expectedRate:**

Returns expected value of false positive rate using bits per element parameter

# 5. Class Name: Bloom Differential

Default Constructor

**Public methods:**

**createFilter:** ​Checks whether a filter already exists, if not, it would create respective filters by adding each word from the file.Return type is BloomFilter.

**retreiveRecord:** Checks in bloom filter if DiffFile contains the record, if yes, it would retrieve it from DiffFile, else it would directly go to database file. False positives are handled here.

**NOTE:**​ We have three different methods for different filters for the above two methods

**naiveRetrieval:**​ Returns a String from file if present

**getStringFromLine**:​ Gets the first string from the line

**Private methods:** ​None

1. **Class Name:**​ ​**NaiveDifferential default Constructor**​.

**Public methods:**

**retreiveRecord:** Checks for String in DiffFile, if it is present, it returns it , else checks in Database file, returns if present or returns null.

**naiveRetrieval:**​ Returns a String from file if present

**getStringFromLine**​: Gets the first string from the line

**Private methods:**  Not used.

1. **Class Name:**​ FNVHash

**Class Constructor:** ​Default constructor

# Public methods: *public static long hash64(final String k)*

**Private methods**​: None

*Note*​: This function is available online [(](http://www.avanderw.co.za/fnv-hash-in-java/)​[http://www.avanderw.co.za/fnv-hash-in-java/)](http://www.avanderw.co.za/fnv-hash-in-java/)​ , we have not created this function.

**8. Class Name:**​ MurmurHash

**Constructor**​: Default constructor.

**Public methods:**

*public static long hash64( final byte[] data, int length, int seed)*

*public static long hash64( final byte[] data, int length)*

*public static long hash64( final String text)*

# *public static long hash64( final String text, int from, int length)*

**Private methods:**​ None

*Note*​: This function is available online, also given as reference in the HW instruction file ([http://d3s.mff.cuni.cz/~holub/sw/javamurmurhash/MurmurHash.jav](http://d3s.mff.cuni.cz/~holub/sw/javamurmurhash/MurmurHash.java)​ [a](http://d3s.mff.cuni.cz/~holub/sw/javamurmurhash/MurmurHash.java))​ , we have not created these functions.

1. **Class Name:**​ GenerateDetHashFunction

**Constructor**​: Default constructor

**Public methods:**  **initMurmur:** gets hash values using MurmurHash and stores the values of bits in low and high variables. **initFNV**​: gets hash values using FNVHash and stores the values of bits in low and high variables. **getHash:**​ Hashes low and high along with the index provided by the bloomfilter class.

Private methods: None

1. **Class Name**​: RandomHashFunction **Class Constructor**​:

• Sets the values of a and b which can be later used to calculate hash value. note that a and b are random values.

**Public methods:**

**hash**​ - Returns the value ​*(a\*x+b)%PRIME*​ where ​*x*​ is the input integer.

**Private methods:**

None

1. ​**Class Name**​: LongBitSet

**Class Constructor**:​ Default

**Public methods:**

*public void set( final long index, final boolean value )*

*public boolean get( final long index )*

**Private methods:**

*private long getSetIndex( final long index )*

*private int getPos( final long index )*

# *private BitSet bitSet( final long index )*

*Note*​: This function is available online

(java-performance.info/bit-sets/​[java-performance.info/bit-sets/](http://java-performance.info/bit-sets/))​ , we have not created these functions and we have used only the above functions.

1. ​**Class Name**:​ **EmpericalComparison**​

**Constructor**​: Default constructor

**Public methods:** ​None

**Main Method-** ​Experiment to compare retrieval times of different filters and naive retrieval. Random keys are selected from grams file and sent as queries to bloomdifferential and naivedifferential. It would output the results in console.

**timeTaken** - time taken for particular filter to retrieve the record. (4 different methods for different filters and naive retrieval)

**getRandomKeys -** Returns keys from grams file based on the inputted indices​

**Private methods:**

None

1. ​**Class Name**​: CMS

**Class Constructor**​:

Intialize all varialbes like rows , columns, datasize, actualdata, data and hashfunctions using the input parameters epsilon,delta and words array.

**Public methods:**  **add** - Adds a string to CMS using random hash functions. Increments the position mentioned by hashfunction in each row by 1

**approximateFrequency** - calculates approximate frequency of a particular word using CMS. the word is hashed and that particular index is looked. Minimum of all those values is returned as approximate frequency.

**approximateHeavyHitter** - Returns a list of words whose approximate frequency/datasize is greater than input q and not less than input r.

**Private methods:**

None

**14.** **Class Name**​ :​ HeavyHitterTest

**Class Constructor**​: Default

**Public methods:** ​None

**Main Method**​- A CMS is constructed using shakespear.txt and a list of approximate heavy hitters are given using the CMS class.

# The process via k-hash values are generated for BloomFilterFNV and BloomFilterMurmur

For both FNV and Murmur hash functions, we have followed the same approach of separating the 64 bit hash value into two 32 bit hash values and multiply the high 32 bits with the index of the row of bloom filter. Then we add the low and modified high bits to get a new hash value. In this way, we get k different hash values for all the k rows in bloom filter.

*Temp\_hashvalue = FNV(String) Hashvalue = lowerbits(Temp\_hashvalue) + row\_number\*(higherbits(Temp\_hashvalue))*

Rationale: This would work because all the k values would be different for a given string.

# The process via k-hash values are generated for class BloomFilterRan

We used the randomhashfunction class and initialized it k times for k hashfunctions and set the values of a and b for all the k hashfunctions and calculate ​*(ax+b)%PRIME*​ where ​*x*​ is the integer representation of the string.

Rationale: We would have different hashvalues for different strings with high probability if we follow the above process

# Experiment to compute False Positives

Design: We have created random strings using different set size and bits per element created filters using this data . Then, we generated random strings and made sure that they are not in the random data previously generated. Then, if the bloom filter gives it as true for this string, it is counted as falsepositive.

Rationale behind the Design: This idea works because there may be chance of collision and the string which is not in the data may come out as positive because of collisions

# Performance of BloomFilterRan, BloomFilterFNV, BloomFilterMurmur when bitsPerElement =4, 8 ,10

This is the output of our program when bitsperElement is 4, 8 and 10.

**For set size = 20000 And bits = 4**

Expected Rate :0.14586595

BloomFilter Random: 0.0227

BloomFilter FNV: 0.1116

BloomFilter Murmur: 0.1092

**For set size = 30000 And bits = 4**

Expected Rate :0.14586595

BloomFilter Random: 0.0225

BloomFilter FNV: 0.1093

BloomFilter Murmur: 0.106

**For set size = 40000 And bits = 4**

Expected Rate :0.14586595

BloomFilter Random: 0.0234 BloomFilter FNV: 0.103

BloomFilter Murmur: 0.1086

**For set size = 50000 And bits = 4**

Expected Rate :0.14586595

BloomFilter Random: 0.0248 BloomFilter FNV: 0.1073

BloomFilter Murmur: 0.1054

**For set size = 20000 And bits = 8**

Expected Rate :0.021276873

BloomFilter Random: 7.0E-4 BloomFilter FNV: 0.0119

BloomFilter Murmur: 0.0141

**For set size = 30000 And bits = 8**

Expected Rate :0.021276873

BloomFilter Random: 7.0E-4 BloomFilter FNV: 0.0134

BloomFilter Murmur: 0.0135

**For set size = 40000 And bits = 8**

Expected Rate :0.021276873

BloomFilter Random: 4.0E-4 BloomFilter FNV: 0.0112

BloomFilter Murmur: 0.0125

**For set size = 50000 And bits = 8**

Expected Rate :0.021276873

BloomFilter Random: 6.0E-4 BloomFilter FNV: 0.0119

BloomFilter Murmur: 0.0128

**For set size = 20000 And bits = 10**

Expected Rate :0.008126148

BloomFilter Random: 1.0E-4 BloomFilter FNV: 0.0035

BloomFilter Murmur: 0.0043

**For set size = 30000 And bits = 10**

Expected Rate :0.008126148

BloomFilter Random: 3.0E-4 BloomFilter FNV: 0.0033

BloomFilter Murmur: 0.0048

**For set size = 40000 And bits = 10**

Expected Rate :0.008126148

BloomFilter Random: 1.0E-4 BloomFilter FNV: 0.0036

BloomFilter Murmur: 0.0032

**For set size = 50000 And bits = 10**

Expected Rate :0.008126148

BloomFilter Random: 1.0E-4 BloomFilter FNV: 0.0033

BloomFilter Murmur: 0.0038

Here are the answers of the given questions:

* **How do false positives depend on bitsPerElement?**

False Positives reduce with increase in number of bits per element. It happened the same with experiment too.

* **Which filter has smaller false positives?**

BloomFilterRan has smaller false positives according to the experiment

* **If there is a considerable difference between the false positives, can you explain the difference?**

BloomFilterRan has very less false positives compared to other two and expected false positives. This might be due to the fact that there are k different hash functions and all of them may not point to same location for two different strings.

* **How far away are the false positives from the theoretical predictions?**

For BloomFilterMurmur and BloomFilterFNV , they are close to the expected rate but BloomFilterRan has very less number of false positives than the expected value.

**Explanation of how EmpericalComparison is comparing the performances of BloomDifferential and NaiveDifferential:**

We have created the bloom filters using the bloomdifferential class and pick random words from grams file used to build bloom filters. Then we have checked the time of retrieval for each of filters and for the one without bloom filters i.e, searching directly in the ​*database.txt*​ file.

**Rationale:** Time is a good factor to measure performance as it is the case of search. If it finds in the bloomfilter , then it searches in *DiffFile*​ or else it would search in database file. If bloom filter gives false as output, then, it is definitely not present in ​*DiffFile* and we remove a overhead of searching in ​*DiffFile*​.

**CMS:**

We created CMS data structure by adding words from the given file *shakespear.txt.*​ We need to add only words whose length is at least 3 and do not add the words “the" or “The" to the data structure. We took ε = 1/100 and δ= 2​-20​.

Using the above built CMS data structure, we need to compute a set L that is ​⟨​0.04, 0.03​⟩ approximate heavy hitter. We need to report the following :

* Number of 0.04 heavy hitters that are in L.

**None (sometimes “and” comes as a result as approximate frequency is around 0.04)**

* Number of 0.025 heavy hitters that are in L.

**1 (“and” for the given database)**

* Number of items in L that are not 0.04 heavy hitters. **All the items in the database**

* Total number of strings that are added to the data structure, and the total number of distinct strings that are added to the data structure.

​**Total No of items :**​​**698521**

**Total No of distinct items : 28040**

* An estimate of total memory used to store the CMS data structure.

**20 rows and 200 columns of size long**

**4000 \* size of long = 4000\*8 bytes = 32KB**

**Acknowledgement:**

We discussed with Ritam Ganguly for this assignment who has been also enrolled in COMS 535 this semester.

**References:**

1. FNV hash : ​<http://www.avanderw.co.za/fnv-hash-in-java/>

1. Murmur Hash : ​<http://d3s.m.cuni.cz/holub/sw/javamurmurhash/MurmurHash.java>

1. Long Bitset :​ [java-performance.info/bit-sets/](http://java-performance.info/bit-sets/)