Com S 435/535: Large Scale Dataset

Report: Bloom Filter And Count Min Sketch

Authors: Yijia Huang and Mohammad Wardat

---

1. Class Specifications

1.1) BloomFilterFNV.java

This class contains the following methods:

```
public BloomFilterFNV(int setSize, int bitsPerElement)
```

Constructor for BloomFilterFNV class. The filter size is setSize * bitsPerElement. The number of hash functions should be ln(2) * filterSize / setSize.

```
public void add(String s)
```

Adds the string s to the filter.

```
public boolean appears(String s)
```

Return true if s appears in the filter.

```
public int filterSize()
```

Returns the size of the filter.

```
public int dataSize()
```

Returns the number of elements added to the filter.

```
public int numHashes()
```

Returns the number of the hash functions.

```
public void clearFilter()
```

Clear all values in the filter.

```
private int fnvHashCode(String s, int seed)
```

Return the hashed code be the input string s and seed. We use a deterministic function(FNV) for hash code. For different seed, we modified the initial hash value to:

$$H = (seed \ \& \ 0xffffffffl) \ \hat{} \ this.FNV\_64NIT$$

To preserve the bit pattern generated by each hash function "& 0xffffffffl" will mask off those bits. Then XOR FNV_64INT to use as much information of the bits as possible and get a good distribution of hash-values. Then we do regular FNV hashing:

for *i* in the range [0,…,s.length() -1] do

h = h XOR s[*i*];

h = (h * FNV64PRIME);

1.2) BloomFilterMurmur.ava

```
public BloomFilterMurmur(int setSize, int bitsPerElement)
```

Constructor for BloomFilterMurmur class. The filter size is setSize * bitsPerElement. The number of hash functions should be ln(2) * filterSize / setSize.

```
private int murmurHashCode(final byte[] data, int length, int seed)
```

Return the hashed code by Murmur hash function based on http://d3s.mff.cuni.cz/~holub/sw/javamurmurhash/MurmurHash.java.

1.3) BloomFilterRan.java

```
public BloomFilterRan(int setSize, int bitsPerElement)
```

Constructor for BloomFilterRan class. The filter size is setSize * bitsPerElement. The number of hash functions should be ln(2) * filterSize / setSize.

```
public int ranHashCode(String s, int[] pair)
```

Return the hashed code by random hash function. We first pick a prime *p* that is bigger than the filter size. Then randomly pick k tuples <*a, b*> where *a, b* are in {0,1,…,*p* -1}. For each hash function, we define that as follows: $h(x) = (ax + b)\%p$.

1.4) FalsePositives.java

```
public static void main(String[] args)
```

Use Helper.getRandomString() to generate 30000 random strings and store them in each filter. Then, generate another 30000 random strings to test their false positive rates. For each bloom filter run the test 10 times and output the average false positive rate.

```
public static void experiment(BloomFilter bf, int bitsPerElement)
```

Create an experiment for each bloom filter to test its positive rate 10 times and output an average rate.

```
public static double falsePositiveRate(BloomFilter bf, int setSize, int
testSize)
```

Create a test on the given bloom filter by storing 30000 random strings and calulate its false positive rate on another 30000 random strings. Then, return the result.

1.5) Application-Differential Files

1.5.1) Differential.java

```
public static int countLines(File file)
```

Returns the number of lines in the file.

```
public static String getKey(String line)
```

Returns the key of the string by taking the fourth words.

```
Public static String retrieveRecordFrom(String key, File file)
```

Returns the target record by giving the key and file.

1.5.2) BloomDifferential.java

```
Public BloomFilter createFilter()
```

Returns a Bloom Filter corresponding to the records in the differential file. We use random bloom filter for our differential comparison.

```
Public String retrieveRecord(String key)
```

Return the corresponding record based on the input key. It will check the filter first. If the key is in the filter, then search for it. If it cannot find the key in the filter, then go to search database.txt file.

1.5.3) NaiveDifferential.java

```
Public String retrieveRecord(String key)
```

Return the corresponding record based on the input key. This will directly search the DiffFile.txt first. If there is no such a key, then search database.txt file.

1.5.4) EmpiricalComparison.java

```
Public static void main(String[] args)
```

We first count the number of keys in the Grams.txt file. Then randomly pick the size / 20000 keys as our test data. For each key, calculate the running times of two differentials and InDiff and InFilter statuses.

```
Public static ArrayList<String> compareDiffs(Differential naive,
Differential bloom, List<String> keys)
```

For each key, calculate the running times of two differentials and InDiff and InFilter statuses.

```
Private static long estimateDifferential(Differential diff, String key,
int[] falsePositive)
```

Calculate the running time and false positive status by searching the key with the differential.

```
Private static List<String> pickRandomKeys(int numKeys)
```

Randomly pick a list of keys(size / 20000) based on the total number of keys in the Grams.txt file.

2. Count Min Sketch

2.1) CMS.java

```
Public CMS(double epsilon, double delta, ArrayList<String> s)
```

Store a list of string s to CMS data structure with random hash function. We only added words with the following conditions:

1. The length of word is at least 3.
2. Not including “the” or “The”.
3. Case-insensitive.

```
Public int approximateFrequency(String x)
```

Return the approximate value of the frequency of the given string.

```
Public ArrayList<String> approximateHeavyHitter(double q, double r)
```

Return an array list of strings with <q, r> heavy hitter.

2.2) CMSTest.java

```
Public static void main(String[] args)
```

Generate a CMS data structure based on the words in the shakespear.txt file. And analysis the data structure by following the instructions of the Programming Assignment.

```
Public static ArrayList<String> getWords()
```

Return a list of words in the shakespear.txt file.

3. Result Report

3.1) False Positive Results

Based on the theory, the false positive probability of non-dynamic filters is around (0.618)^(bitsPerElement). And the results show as below in the table. As we can see, our results are close to the theoretical ones. Overall, the random bloom filter has the best performance since it has the lowest false positive rate. For our observations, the random bloom filter has the domination of the lowest FP rate as the number of bits per element increasing.

Bits Per Element	Theoretical FP Rate	BloomFilterFNV	BloomFilterMurmur	BloomFilterRan
4	14.58%	15.22%	14.62%	14.74%
8	2.128%	2.351%	2.152%	2.117%
10	0.813%	0.855%	0.813%	0.806%

Table 1: Comparing between BloomDifferential and NaiveDifferential

3.2) Application-Differential Files: Result and Data Analysis

First, we selected 631 keys from grams.txt file and run the queries into two programs, then we get the following result:

File	BloomDifferential	NaiveDifferential	# Queries
Differential	6915.537906	8213.272563	64
Database	657.234375	774.75	554

Table 2: Comparing between BloomDifferential and NaiveDifferential

Note: The time for each of these queries in nanoseconds was recorded.
We have 631 keys, we found that the number of keys in the Differential file equal to 64 keys, and from the result in the table above, we can see that the average of time for BloomDifferential is faster than NaiveDifferential if the key is available in differential file.

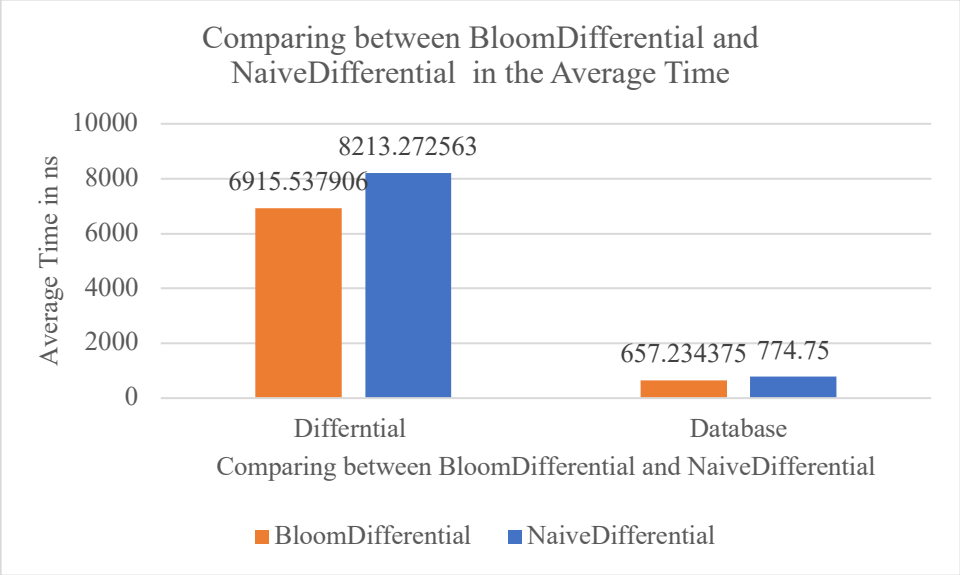

Fig 1: Comparing between BloomDifferential and NaiveDifferential in Average Time

In other side, if we see the result when the key is available in database file, we can investigate also, the BloomDifferential is faster than NaiveDifferential, in this case we took 554 queries and we didn't include the False positive queries. At the end we can see that BloomDifferential is faster in both cases.

Now, how many time the false positive occurred in our experiment, we found that the number of false positive that the BloomDifferential returned true and we didn't find the record in differential file, it was 13 times, then the probability of false positive in our experiment was 2.29%. Since the we are using 8 bits per element, then theoretical false positive rate is 2.13%.

From Fig 1 and Fig 2, we can see the different between BloomDifferential and NaiveDifferential, and how the BloomDifferential is faster than NaiveDifferential in both cases, we select 20 queries for each case and show the result in line graph as we can see below.

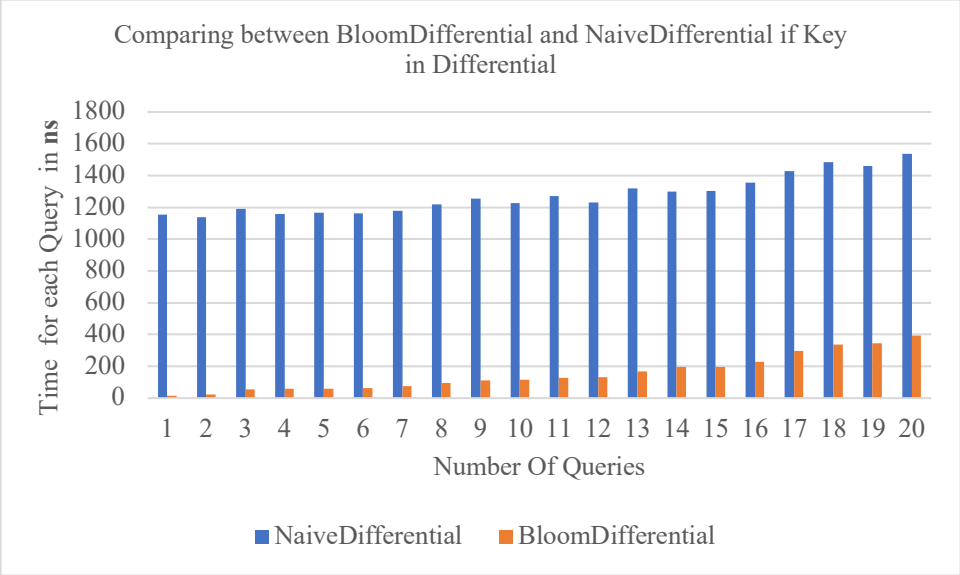

Fig 2: Comparing between BloomDifferential and NaiveDifferential if Key in Differential file

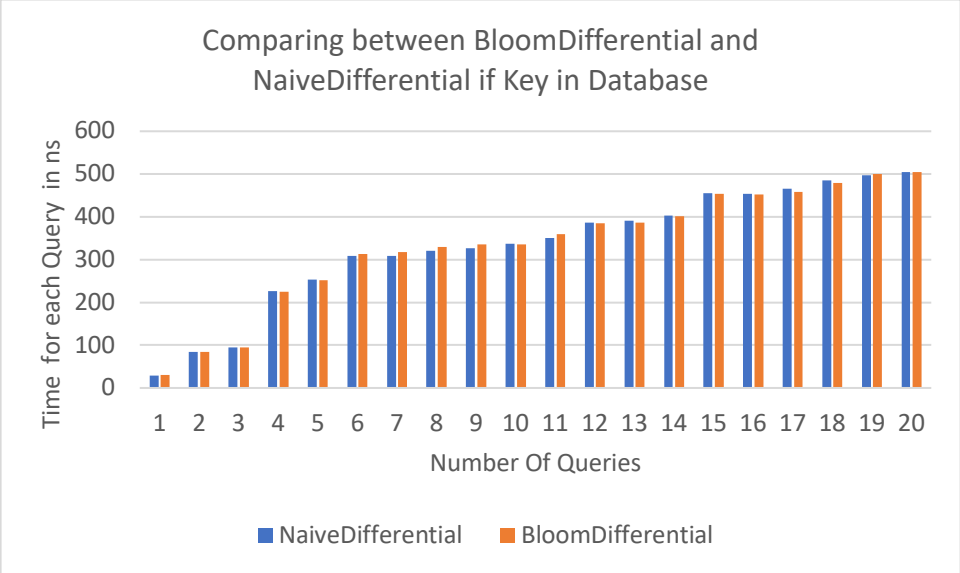

Fig 2: Comparing between BloomDifferential and NaiveDifferential if Key in Database file

3.3) Count Min Sketch

1- Number of 0.04 heavy hitters that are in L.
Our result is 0.

```
actucal and 26738
cms and 27342
a) Number of 0.04 heavy hitters: 0
b) Number of 0.025 heavy hitters: 1
c) Number if items in L that are not 0.04 heavy hitters: 1
d) Total number of strings: 682454
d) Total number of distinct strings: 23388
e) Memory used: 65MB
```

2- Number of 0.025 heavy hitters that are in L.
Our result is 1 which is “and” word. Since the total number of strings is 682454, then 0.04 * N = 27298 and 0.025 * N = 17061. Our actual frequency of “and” in s list is 0.025 * N < 26738 < 0.04 * N.

3- Number of items in L that are not 0.04 heavy hitters.
Our result is 1.

4- 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.
The total number of strings is 682454 and the total number of distinct strings is 23388.

5- An estimate of total memory used to store the CMS data structure.
Since an int has 4 bytes in java, then 4 * rows * cols / 1024 = 15.625 KB. But the measured memory used to store the CMS data structure is 65MB.