2017

Bloom Filter

COMS 535 – PROGRAMMING ASSIGNMENT 1

SHRUTI SAHU (106202120) & HUNG PHAN(799388622)

I. BLOOMFILTER

1. FUNCTIONALITY

BloomFilter is an abstract class that has 2 parameterized constructors.

1. **BloomFilter(int bitsPerElement)**

   This constructor is for the DynamicFilter which extends BloomFilterRan, a child class of BloomFilter.

2. **BloomFilter(int setSize, int bitsPerElement)**

   This constructor initializes the setSize, bitsPerElement to the values passed while creating object of this class. It also sets the filterSize to (setSize * bitsPerElement), numHashes to (**int**) (Math.*log*(2) * bitsPerElement) and initializes the filter array to a size of the filteSize.

2. INHERITED MEMBERS

None

3. ABSTRACT METHODS

1. **hashFunction(String s)**

   The classes that extend this BloomFilter class implement their own hashFunction(), which basically returns an array of hash values depending on the type of hash function they use. It takes the String s as the argument. This is the string whose hash values are computed and returned.

4. PUBLIC METHODS

1. **add(String s)**

   This function adds a string to the bloom filter. It computes the hash values as per the implementation of the hashFunction().

2. **appears(String s)**

   This function checks if the string s appears in the bloom filter or not. It computes the hash values of the string s and then checks if the bit is set for the position values returned by the hash functions.

3. **filterSize()**

   It returns the filter size.

4. **dataSize()**

   It returns the size of the data.

5. **numHashes()**

   It returns the number of hash functions used. It is computed once the object of BloomFilterRan, BloomFilterFNV, etc (all the child classes of the abstract class BloomFilter) type is created.

5. PRIVATE MEMBERS

1. **filterSize –** Represents the filter size
2. **numHashes** - Represents the number of hashes.
3. **filter –** Represents the filter.
4. **setSize –** Represents the set size
5. **bitsPerElements** – Represents the bits per element.
6. **dataSize –** Represents the size of the data added to the filter.

II. BLOOMFILTERFNV

1. FUNCTIONALITY

Create a Fowler–Noll–Vo hash (FNV) function. it takes an input as a string a and return h(a). To have k hash functions based on FNV, from string a we add the **index of hash function** to produce a1, a2, … , ak. For example, string "hello" with 2 hash functions will have "01hello" and "02hello" as input of FNV function. because the BitSet has the length n, the output h(a1),…., h(a2) will be moduled by n to get n final hash results.

2. INHERITED MEMBERS

1. filterSize – Represents the filter size
2. numHashes - Represents the number of hashes.
3. filter – Represents the filter.
4. setSize – Represents the set size
5. bitsPerElements – Represents the bits per element.
6. dataSize – Represents the size of the data added to the filter.

3. INHERITED METHODS

1. add(String s)
2. appears(String s)
3. filterSize()
4. dataSize()
5. numHashes()

4. OVERRIDEN METHODS

1. hashFunction(String s).
2. public void add(String s)
3. public boolean appears(String s)
4. public int filterSize()
5. public int numHashes()

5. PRIVATE METHODS

1. private long hash64(byte[] data) – get FNV hash 64 from byte array
2. private long hash64(byte[] data, int length) – get FNV hash from byte array with specific length
3. private long hash64(final String k) – get FNV hash for string s.

6. PRIVATE MEMBERS

1. private static final long FNV_64_INIT = 0xcbf29ce484222325L;
2. private static final long FNV_64_PRIME = 0x100000001b3L;

7. FUNCTIONALITY

The k hash functions are generated by modifying the input to k inputs by adding index of hash functions, before get k FNV hashes.

III. BLOOMFILTERRAN

1. FUNCTIONALITY

Creates a bloom filter with random hash function. It picks a prime number greater than tN (where N is the setSize and t is bits per element) and sets it to the size of the filter.

2. INHERITED MEMBERS

7. filterSize – Represents the filter size
8. numHashes - Represents the number of hashes.
9. filter – Represents the filter.
10. setSize – Represents the set size
11. bitsPerElements – Represents the bits per element.
12. dataSize – Represents the size of the data added to the filter.

3. INHERITED METHODS

6. add(String s)
7. appears(String s)
8. filterSize()
9. dataSize()
10. numHashes()

4. OVERRIDEN METHODS

6. hashFunction(String s)

5. PRIVATE METHODS

4. pickPrime() – Picks a prime number that is at least t*N, where t is the bits per elements and N is the set size.

6. PRIVATE MEMBERS

1. a – the randomly picked number from 0 to p-1
2. b – the randomly picked number from 0 to p-1

7. RATIONALE

The rationale behind generating k hash functions is that, k hash functions are generated in the following manner:

```
int h = Math.abs(this.a[i]*s.hashCode() + this.b[i]);
                    hashFnVals[i] =  h % filterSize;
```

Here, for every k hash function the values of a and b picked randomly, i.e. for every i^{th} hash function the value of a and b is randomly picked.

IV. BLOOMFILTERMURMUR

1. FUNCTIONALITY

Creates a bloom filter using murmur hash function.

2. INHERITED MEMBERS

1. filterSize – Represents the filter size
2. numHashes - Represents the number of hashes.
3. filter – Represents the filter.

4. setSize – Represents the set size
5. bitsPerElements – Represents the bits per element.
6. dataSize – Represents the size of the data added to the filter.

3. INHERITED METHODS

1. add(String s)
2. appears(String s)
3. filterSize()
4. dataSize()
5. numHashes()

4. PROTECTED METHODS

None

5. OVERRIDDEN METHODS

1. hashFunction(String s)

6. PRIVATE MEMBERS

None

7. RATIONALE

The k hash functions are generated by XORing the seed with the ith hash function number. 0<=i<k.

V. DYNAMICFILTER

1. FUNCTIONALITY

Creates a dynamic bloom. The data size is initially set to 1000. Once 1000 number of data items have been added to the filter, the filter size is doubled.

2. INHERITED MEMBERS

1. a array
2. b array
3. filterSize – Represents the filter size
4. numHashes - Represents the number of hashes.
5. filter – Represents the filter.
6. setSize – Represents the set size
7. bitsPerElements – Represents the bits per element.
8. dataSize – Represents the size of the data added to the filter.

3. INHERITED METHODS

1. pickPrime()
2. hashFunctions()

4. PRIVATE METHODS

NONE

5. PRIVATE MEMBERS

1. dynamicFilters()

6. OVERRIDDEN METHODS

1. add()
2. appears()

7. RATIONALE

The dynamic filter uses random hash functions created in the BloomFilterRan. As the filter size is doubled, the values of a and b are picked again for the hash functions.

VI. RESULTS

You can run the main function of FalsePositives.java to get this result.

```
Bits per element: 4
Theoretical FP= 0.14586594177599999
FNV FP: 0.155116
Murmur FP: 0.155232
Random FP: 0.19792
Dynamic FP: 0.262678

Bits per element: 8
Theoretical FP= 0.02127687297019942
FNV FP: 0.021822
Murmur FP: 0.021522
Random FP: 0.022124
Dynamic FP: 0.196352

Bits per element: 16
Theoretical FP= 4.527053233900027E-4
FNV FP: 5.4E-4
Murmur FP: 5.0E-4
Random FP: 6.12E-4
Dynamic FP: 0.18918
```

Time analysis

To get time analysis, we get average 10 times for running each configuration. The result is shown in this table.

Analysis for make BloomFilter of 500000 (milliseconds)

Tyoe of hash	bitPerE=4	bitPerE=8	bitPerE=16
FNV	989.7	2936.1	7883.5
MurMur	344.8	729.3	1601.2
Random	162.4	243.9	431.0
DynamicFilter	1212.5	2537.5	5497.2

Analysis for querying 500000 (milliseconds)

Tyoe of hash	bitPerE=4	bitPerE=8	bitPerE=16
FNV	830.9	2735.6	11241.2
MurMur	404.4	748.4	1732.2
Random	206.4	241.7	352.7
DynamicFilter	202.1	250.5	572.0

In this results, we can see that the time on FNV and MurMur are higher, due to the fact that the time to generate hash value is longer than Random and DynamicFilter.

VII. FALSE POSTIVE RATIONALE

To setup this experiment, we created the data file and query file by Random string. The data file (data_2.txt) has **500000** lines while query file (q_2.txt) has **500000** lines. The intersect between query and data is 250000 lines.

The FalsePositive.java case is the case that the string in query doesn't appear in the data, but the appear() function of the Bloom Filter return true. We get the false positive rate by count those cases and divide it to the number of queries.

You can run FalsePositive.main() to get results of 4 hash functions.

VIII. BLOOM JOIN

Rationale. We implemented the BloomJoin according to the requirement stated in the assignment. The output will be created at R3.txt. To validate the result, we have another function **doValidate()** that stored data of R1 and R2 by HashMap, and get validate result in **R3Validate.txt**.

Result. R3 and R3Validate are actually matched, means there are no False Positive cases. There are over 190000 lines in the result. We used FNV for the hash function.

You can run BloomJoin.main() to get the result.

PUBLIC Methods.

1. validateAction(String r3Validate) – create R3Validate that used Join by Java HashMap to get actual results.
2. doAction(BloomFilter bloom) – Make a bloom filter from data.txt
3. join(String r3) – join and output R3.txt.

IX. DYNAMIC AND RANDOM BLOOM FILTER COMPARISON

The random bloom filter gives a false positive value very close to the theoretical false positive. The dynamic filter however gives a comparatively very higher false positive.

X. REFERENCE

1 The Dynamic Bloom Filter, http://ieeexplore.ieee.org/document/4796196/

2 http://d3s.mff.cuni.cz/~holub/sw/javamurmurhash/MurmurHash.java