Implementation of Bloom Filter

Nikhil Balwani (1641070), B.Tech ICT

***Abstract*—The main requirement of big data systems is the ability to deal with datasets for which traditional algorithms with polynomial run-times are not feasible. In this paper, we present a systematic study of the Bloom filter, a space-efficient data structure for testing set membership. The Bloom Filter in itself uses a data structure called bit-vector, with a given number of bits. Its implementation also includes a set of independent and uniform hash functions. First, we present the working principle of the filter and establish the data structures and the algorithms necessary for its implementation. Next, we discuss the details of its implementation. Finally, we present the false positive error rate as a function of the number of bits in the bit vector, and the number of independent hash functions employed in the algorithm and give the execution time of our implementation.**

***Index Terms*—Bloom filters, Streaming algorithms, Membership Testing.**

I. Introduction

The Bloom filter has emerged as one of the promising solutions for big data applications and can provide a scalable and space-efficient membership testing. It was devised by Burton Howard Bloom in 1970 [1]. It has found applications in Web Cache [2], Navigation Systems [3], Parallel and Distributed Computing [4], and Network Security [5].

The Bloom filter is capable of responding to a query in $\mathcal{O}(1)$ time. [1] However, the time and space efficiency of the Bloom filter comes with its tendency to predict false positives. In essence, the element being tested for membership may be predicted as a member of the set when the element is not a member in actuality.

This paper seeks to explore the equations that govern the false positive error rate of this algorithm, with the nature of performance when different parameters of the algorithm are varied.

The rest of this paper is organized as follows. First, Section II presents the working principle of the Bloom filter. Section III describes the implementation details of this algorithm. The experimental results are analysed and discussed in Section IV. Finally, Section V draws the main conclusions from the study.

II. Working Principle of the Bloom Filter

The Bloom filter uses a data structure called bit-vector. It contains a specified number of contiguous bits. These bits are capable of storing binary data where a bit can be either set or reset. When a bit in the bit vector is set, it takes the value 1, and whenever it is reset, it takes the value 0. Fig. 1 shows a 16-bit vector.

It also uses a set of k-wise independent and uniform hash functions, where k is the number of hash functions that this algorithm employs. Essentially, the value that each hash function produces should be independent from any other hash function being used with this algorithm.

Algorithm 1 Insertion

```
1: function INSERT(element, bit_vector)
2:     for j ← 1 to k do
3:         h_k ← (k)^th independent hash function
4:         index ← h_k(element)            ▷ Apply the hash function
5:         bit_vector[index] ← 1  ▷ Set the bit vector at the index}
6:     end for
7:     return bit_vector
8: end function
```

Algorithm 2 Membership Test

```
1: function TEST MEMBERSHIP(element, bit_vector)
2:     for j ← 1 to k do
3:         h_k ← (k)^th independent hash function
4:         index ← h_k(element)            ▷ Apply the hash function
5:         if bit_vector[index] is not 1 then   ▷ Check if index is 0
6:             return False
7:         end if
8:     end for
9:     return True
10: end function
```

Algorithm 1 illustrates how an element is inserted into a set that implements Bloom filter. Whenever we see an element, we apply the k different hash functions described in the above paragraph, and we set the indices in the bit-vector that correspond to the values generated by these hash functions.

The procedure for membership test has been described in Algorithm 2. We apply the same k hash functions to the element to be tested for membership, and if the indices in the bit-vector that correspond to values generated are not set to 1, we conclude that the element does not belong the set. This algorithm shows some susceptibility to false positive errors albeit.

The false positive rate of this algorithm is given by equation (1) and equation (2) gives the optimal number of hash functions k for the filter with given input set size and bit vector length.

$$p_f = (1 - e^{-kn/m})^k \tag{1}$$

$$k = \frac{m}{n} \ln 2 \tag{2}$$

where k is the number of hash functions, n is the number of elements inserted and m is the size of bit vector.

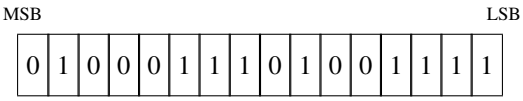

Figure 1: A 16-bit vector.

Theorem II.1 (Optimality formulas)**.** *The optimal values of bit-vector size m and number of hash functions k for a given false positive rate p_f and input set size n are given by*

$$k = -\log_2 p_f$$

$$m = -n\frac{\log_2 p_f}{\ln 2}$$

Proof. Putting equation (1) in equation (2), we have

$$p_f = \left(1 - e^{-(\frac{m}{n}\ln 2)\frac{n}{m}}\right)^{\frac{m}{n}\ln 2}$$

which can again be simplified to

$$\ln p_f = -\frac{m}{n}(\ln 2)^2$$

Again,

$$\boxed{m = -n\frac{\log_2 p_f}{\ln 2}} \quad (3)$$

Putting equation (2) in (3),

$$\boxed{k = -\log_2 p_f} \quad (4)$$

□

The next section describes the implementation details of the Bloom filter.

III. Implementation Details

The algorithm was implemented in Python 3.6 programming language using Jupyter Notebook plugin [6]. MurMurHash3 (mmh3) [7] was used to generate k-wise independent hash functions. BitVector [8] was employed to use the bit vector data structure. UUID [9] was used to produce unique strings for insertion and testing of the Bloom filter. Python's inbuilt math library was also employed.

Given the values of n and p_f, we used equations (3) and (4) to calculate the optimal values of m and k, and designed the filter with these values of m and k.

We then did a simulation of the false positive rate on the implementation by adding n unique elements to the set and then by testing another 10,000 unique elements (different from the ones already added to the set). The simulated value of p_f is calculated as the fraction of these values that the filter predicted as positives.

Table I gives the optimal values of the filter calculated by the program and the simulated p_f. The simulated false positive rate matches with the desired value in each case.

Section IV shows the results of our work.

Table I: Optimal values of m and k for different values of input false positive rate p_f and input-set size n

n	Desired p_f	m	k	Simulated p_f
10^6	0.0100	9585059	7	0.0104
10^6	0.0500	6235225	5	0.0510
10^4	0.0100	95851	7	0.0107
10^6	0.0500	62353	5	0.0510

IV. Results

Using the algorithms illustrated in Section II (Algorithms 1, 2), insertions and membership-tests were performed. Using equation (1), the simulation results were validated.

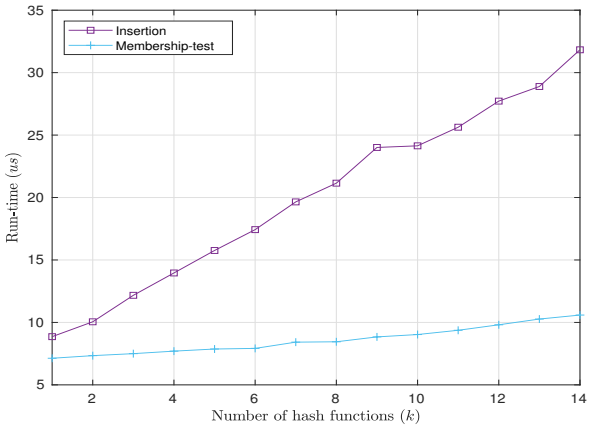

Figure 2: Insertion and Membership-test run-times for different values of k, for $n = 10^5$, $m = 10^6$.

Table II: Insertion and Membership-test run-times for different values of k, for $n = 10^5$, $m = 10^6$.

k (number of hash functions)	**Insertion-time (μs)**	**Membership-test run-time (μs)**
1	8.86	7.13
2	10.05	7.34
3	12.17	7.50
4	13.96	7.70
5	15.76	7.87
6	17.43	7.92
7	19.65	8.42
8	21.15	8.45
9	24.01	8.84
10	24.14	9.03
11	25.63	9.37
12	27.72	9.81
13	28.99	10.27
14	31.83	10.59

A. Execution Time of the implementation

The execution-time of this implementation is expected to increase with increase in the number of hash functions. With each hash function added to the implementation, the algorithm needs to perform more computation.

Fig. 2 and table II show the execution-times of our implementation for different values of k. The

membership-test operation is computationally cheaper than the insertion operation, due to the fact that insertion does not involve updation of the bit-vector in the filter. It can be seen from fig. 2 and table II that the average execution time of membership-test does not increase as rapidly as the insertion operation.

B. Performance Analysis

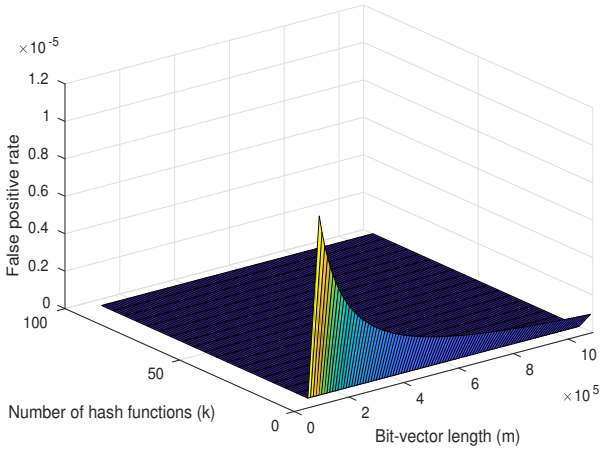

Figure 3: False positive rates for different values of k and m, for $n = 10^2$.

The filter was tested on a range of values of m and k for fixed $n = 10^2$. Fig. 3 shows the false positive rate as a function of the input parameters m and k. It can be inferred that with an increase in both, the bit-vector size m and the number of hash functions k, f_p goes down, in essence, the performance improves.

Another inference is that after reaching close to the zero false positive rate, the graph flattens. Therefore, after this point in the curve, an increase in m and k does not lead to much improvement in performance. An increase in the values of m and k after this limit will only lead to overheads in the implementation in terms of time and space complexity. For instance, a very high value of m would mean a very big bit-vector, which will add to space-complexity. On the other hand, a very high value of k means a decline in time-efficacy of the implementation.

V. Conclusion

In this paper, we saw the working principle of a Bloom filter and the algorithms and equations that govern its behaviour. We established the equations for optimal values of the parameters bit-vector length m and number of hash functions k for a given false positive rate f_p and input-set size n. The obtained results demonstrate that after a limit, an increase in the values of m and k does not add much to the performance, but can significantly add to the time and space complexity of the filter. We also saw that the insertion operation in a Bloom filter is more expensive than testing an element for set-membership.

References

[1] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970. [Online]. Available: http://doi.acm.org/10.1145/362686.362692

[2] C. Jing, "Application and research on weighted bloom filter and bloom filter in web cache," in *2009 Second Pacific-Asia Conference on Web Mining and Web-based Application*, June 2009, pp. 187–191.

[3] C. Jing, N. Zhengang, L. Liying, and Y. Fei, "Research and application on bloom filter in routing planning for indoor robot navigation system," in *2009 Pacific-Asia Conference on Circuits, Communications and Systems*, May 2009, pp. 244–247.

[4] Y. Li, "Memory efficient parallel bloom filters for string matching," in *2009 International Conference on Networks Security, Wireless Communications and Trusted Computing*, vol. 1, April 2009, pp. 485–488.

[5] K. Saravanan, A. Senthilkumar, and P. Chacko, "Modified whirlpool hash based bloom filter for networking and security applications," in *2014 2nd International Conference on Devices, Circuits and Systems (ICDCS)*, March 2014, pp. 1–6.

[6] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay *et al.*, "Jupyter notebooks-a publishing format for reproducible computational workflows." in *ELPUB*, 2016, pp. 87–90.

[7] A. Appleby, "Murmurhash 2.0," 2008.

[8] A. K. Purdue University, "Bitvector 1.5.1," 2008. [Online]. Available: https://engineering.purdue.edu/kak/dist/BitVector-1.5.1.html

[9] "Uuid 2.5," 2008. [Online]. Available: https://docs.python.org/2/library/uuid.html

Bloom Filter

March 27, 2019

1 Bloom Filter Implementation in Python

1.1 Required Libraries

The required libraries are:

1. MurMurHash3 library for generating k-wise independent hash functions
2. BitVector library for Bit manipulation in python
3. In-built math library
4. UUID library for generating unique identifiers (strings)

```
In [83]: import mmh3
         import BitVector as bv
         import math
         import uuid
         import time
```

2 Implementation of bloom filter

The following equation was used to calcualted the analytical false positive rate.

$p_f = (1 - e^{-kn/m})^k$

```
In [84]: class BloomFilter:
             def __init__(self, m, k):

                 # Bit vector length 'm'
                 self.m = m

                 # Number of hash functions
                 self.k = k

                 self.n = 0

                 # Create a bit vector of size 'm'
                 self.bit_vector = bv.BitVector(size=self.m)

                 # Initialize all elements to zero
                 for i in self.bit_vector:
```

```
            self.bit_vector[i] = 0


    def add_element(self, key):
        for i in range(1, 1 + self.k):
            # Implement hash function and find the index of bit to set
            index = ( hash(key) + i * mmh3.hash(key) ) % self.m

            self.bit_vector[index] = 1

        self.n += 1

    def has_element(self, key):
        for i in range(1, 1 + self.k):
            # Implement hash function and find the index of bit to check for
            index = ( i * mmh3.hash(key) ) % self.m

            # For a positive, each of the bits should be set
            if self.bit_vector[index] != 1:
                return False

        return True

    def get_no_of_elements(self):
        return self.n

    def get_bit_vector_length(self):
        return self.m

    def get_no_of_hash_functions(self):
        return self.k

    def get_analytical_fp_rate(self):
        return math.pow( ( float(1) - math.exp(-(self.k * self.n) / self.m) ), self.k)

    def get_simulated_fp_rate(self):

        false_positives = 0
        fp_count = 10 ** 5

        sum = 0.0

        for i in range(fp_count):

            a = time.time()
            if self.has_element(str(uuid.uuid4())):
                false_positives += 1
            b = time.time()
```

```
                sum += float(b - a)

            sum /= float(fp_count)

            print("The average membership-test time is", sum)

            return float(false_positives) / float(fp_count)
```

2.1 Function to test the bloom filter

We run a simulation to check the value of p_f.

```
In [85]: def test_bloom_filter(n, m, k):
             bloom_filter = BloomFilter(m, k)

             sum = 0.0

             for i in range(n):

                 # Adding a unique random element

                 a = time.time()
                 bloom_filter.add_element(str(uuid.uuid4()))
                 b = time.time()

                 sum += float(b - a)

             sum /= float(i)

             print("The actual false positive error rate is %.9f" % (bloom_filter.get_simulated_
             print("The analytical false positive error rate is %.9f" % (bloom_filter.get_analyt
             print("The average insertion time is", sum)

             return bloom_filter


In [86]: test_bloom_filter(n=100000, m=1000000, k=10)

The average membership-test time is 9.432196617126465e-06
The actual false positive error rate is 0.009870000
The analytical false positive error rate is 0.010185894
The average insertion time is 2.436190886795953e-05


Out[86]: <__main__.BloomFilter at 0x7f66dde00ac8>
```

2.2 Optimal Bloom filter

The following function automatically generates the values of k and m when n and pf are provided.

$$k = -\log_2 p_f$$

$$m = -n\frac{\log_2 p_f}{\ln 2}$$

```
In [87]: def optimal_bloom_filter(n, pf):
             k = math.ceil(- math.log(pf, 2))
             m = math.ceil(- ( n * math.log(pf, 2) ) / math.log(2))

             bloom_filter = test_bloom_filter(n=n, m=m, k=k)

             return bloom_filter, m, k
```

```
In [88]: bloom_filter, m, k = optimal_bloom_filter(n=10**4, pf=0.05)
         print("m =", m)
         print("k =", k)
```

```
The average membership-test time is 8.649888038635254e-06
The actual false positive error rate is 0.050270000
The analytical false positive error rate is 0.051026670
The average insertion time is 1.9604759891577536e-05
m = 62353
k = 5
```

```
In [90]: for k in range(1, 15):

             bloom_filter = test_bloom_filter(n=10**4, m=10**6, k=k)
             print(bloom_filter.get_analytical_fp_rate())
```

```
The average membership-test time is 7.109525203704834e-06
The actual false positive error rate is 0.010160000
The analytical false positive error rate is 0.009950166
The average insertion time is 8.803389646826011e-06
0.009950166250831893
The average membership-test time is 7.76865005493164e-06
The actual false positive error rate is 0.000360000
The analytical false positive error rate is 0.000392093
The average insertion time is 1.1109366322984838e-05
0.00039209253881260697
The average membership-test time is 7.250192165374756e-06
The actual false positive error rate is 0.000040000
The analytical false positive error rate is 0.000025815
The average insertion time is 1.2403441042956835e-05
2.5814835993411125e-05
The average membership-test time is 7.687451839447022e-06
The actual false positive error rate is 0.000010000
The analytical false positive error rate is 0.000002364
```

```
The average insertion time is 1.547779842834137e-05
2.3638081031360614e-06
The average membership-test time is 8.07948112487793e-06
The actual false positive error rate is 0.000000000
The analytical false positive error rate is 0.000000276
The average insertion time is 1.76040610500259e-05
2.7592395203899934e-07
The average membership-test time is 7.97454833984375e-06
The actual false positive error rate is 0.000000000
The analytical false positive error rate is 0.000000039
The average insertion time is 1.9925727237640757e-05
3.900545504853698e-08
The average membership-test time is 8.208930492401124e-06
The actual false positive error rate is 0.000000000
The analytical false positive error rate is 0.000000006
The average insertion time is 2.127171087317472e-05
6.4551269222458435e-09
The average membership-test time is 8.051190376281738e-06
The actual false positive error rate is 0.000000000
The analytical false positive error rate is 0.000000001
The average insertion time is 2.3420911655984935e-05
1.2208775482886281e-09
The average membership-test time is 7.742969989776612e-06
The actual false positive error rate is 0.000000000
The analytical false positive error rate is 0.000000000
The average insertion time is 2.3401526286967075e-05
2.591865138064519e-10
The average membership-test time is 8.078160285949708e-06
The actual false positive error rate is 0.000000000
The analytical false positive error rate is 0.000000000
The average insertion time is 2.748160996977383e-05
6.09062931691357e-11
The average membership-test time is 7.612516880035401e-06
The actual false positive error rate is 0.000000000
The analytical false positive error rate is 0.000000000
The average insertion time is 2.648906953836253e-05
1.5666776835610565e-11
The average membership-test time is 7.591769695281982e-06
The actual false positive error rate is 0.000000000
The analytical false positive error rate is 0.000000000
The average insertion time is 2.826916144983639e-05
4.371288503579655e-12
The average membership-test time is 7.612566947937012e-06
The actual false positive error rate is 0.000000000
The analytical false positive error rate is 0.000000000
The average insertion time is 3.3182553713745396e-05
1.3129850680754783e-12
The average membership-test time is 7.462508678436279e-06
```

```
The actual false positive error rate is 0.000000000
The analytical false positive error rate is 0.000000000
The average insertion time is 3.213588685223503e-05
4.218407495843887e-13
```

```
In [91]: matrix = []
         k_arr = []
         m_arr = []

         for k in range(1, 100, 5):
             k_arr.append(k)
             row = []
             for m in range(10**5, 11*(10**5), 10**4):
                 if k == 1:
                     m_arr.append(m)

                 row.append(math.pow( ( (float(1) - math.exp(-( k ) * (10**2)) ) / m) , k))
                 #bloom_filter = test_bloom_filter(n=10**5, m=m, k=k)
                 #row.append(bloom_filter.get_analytical_fp_rate())
             matrix.append(row)

         print(k_arr)
         print(m_arr)
```

```
[1, 6, 11, 16, 21, 26, 31, 36, 41, 46, 51, 56, 61, 66, 71, 76, 81, 86, 91, 96]
[100000, 110000, 120000, 130000, 140000, 150000, 160000, 170000, 180000, 190000, 200000, 210000,
```

```
In [92]: len(matrix[0])
```

```
Out[92]: 100
```

```
In [93]: print("[", end='')
         for i in range(0, len(matrix)):
             print("[", end='')
             for j in range(0, len(matrix[0])):
                 print("", matrix[i][j], " ", end=''),

             print("];")
         print("];")
```

```
[[ 1e-05   9.090909090909091e-06   8.333333333333334e-06   7.692307692307692e-06   7.14285714285
[ 1.0000000000000004e-30   5.6447393005377745e-31   3.348979766803842e-31   2.071762110330033e-3
[ 1.0000000000000009e-55   3.5049389948139253e-56   1.3458798574153817e-56   5.579857714338898e-
[ 1.0000000000000013e-80   2.176291357901488e-81   5.408789293239544e-82   1.5028179131680093e-8
[ 1.0000000000000017e-105   1.3513057093103975e-106   2.1736711087157377e-107   4.04752557459475
[ 1.000000000000002e-130   8.390545288824022e-132   8.735496675330095e-133   1.0901163163848363e
[ 1.0000000000000025e-155   5.2098684819243735e-157   3.510600194239526e-158   2.936000184180245
```

```
[ 1.000000000000003e-180   3.234918430760675e-182   1.410831482381497e-183   7.907502118758621e-
[ 1.0000000000000033e-205   2.0086298320163644e-207   5.669815306638605e-209   2.129720226009814
[ 1.0000000000000037e-230   1.2472010928316896e-232   2.2785716092136891e-234   5.73595577080584
[ 1.0000000000000041e-255   7.744137526818769e-258   9.157068259764377e-260   1.5448596582229777
[ 1.0000000000000046e-280   4.808500119104365e-283   3.680020359023108e-285   4.1607562173889013
[ 1.0000000000000049e-305   2.9857002558843874e-308   1.47891764685536e-310   1.120612620595e-31
[ 0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
[ 0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
[ 0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
[ 0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
[ 0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
[ 0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
[ 0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
];
```