

BLOOM FILTERS AND HYPER LOGLOG

SEMINAR REPORT

Submitted by

RITESH BUCHA [RA1811003010914]

DEBADITYA SEN [RA1811003010897]

In partial fulfilment for the award of the degree

of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

of

FACULTY OF ENGINEERING AND TECHNOLOGY



S. R. M. Nagar, Kattankulathur, Kancheepuram District

DECEMBER 2020


SRM INSTITUTE OF SCIENCE AND

TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that the Industrial training report titled “**Bloom Filters and Hyper loglog**” is the bonafide work of “**Ritesh Bucha [RA1811003010914]** and **Debaditya Sen [RA1811003030897]**” submitted for the course 18CSP103L Seminar – I. This report is a record of successful completion of the specified course evaluated based on literature reviews and the supervisor. No part of the Seminar Report has been submitted for any degree, diploma, title, or recognition before.


27/11/2020

SIGNATURE

Mrs.B.Sowmiya
Assistant Professor
Dept. of Computer Science &
Engineering

SIGNATURE

Dr. S. Ganesh Kumar
Academic Advisor
Dept. of Computer Science
& Engineering

TABLE OF CONTENTS

SNo.	Title	Page No.
1	Abstract	4
2	Acknowledgment	5
3	Terminology	6
4	Introduction	7
5	Objective	9
6	Advantages and Disadvantages	11
6	Literature survey	12
8	Methodology	15
9	Module Description	19
12	Novelty	23
13	Conclusion and limitation	24
14	Future Enhancement	25
15	References/Bibliography	26

ABSTRACT

The constant frequency and variety of high-speed data have resulted in the growth of large data sets in the last two decades. Storage of this data in memory is not possible due to storage constraints. As a result, today's streaming algorithms need to work with one pass of data. Many streaming algorithms address different problems with data streams. Data stream problems though originated around the '70s have gained popularity only in that last 15 years due to the exponential growth of data from a variety of sources. Today's researchers aim at providing solutions to specific problems that are found across many industries. Most of the problems addressed in this paper are the ones researchers have considered as oft faced problems in the industry like heavy-hitters/top k elements or frequency count or membership check or cardinality. Algorithms addressing these problems like **Bloom Filter** and **Hyper Loglog** and such are being continuously improved for faster and effective results. Hashing is a technique used by many algorithms to resolve massive data problems affecting today's applications. Hashing enables streaming data to be able to store and retrieve data resourcefully. In this paper, we analyse both the problems in streaming data and the various algorithms used to resolve them.

Undoubtedly, dealing with security issues is one of the most important and complex tasks various networks face today. A large number of security algorithms have been proposed to enhance security in various types of networks. Many of these solutions are either directly or indirectly based on Bloom filter (BF), a space- and time-efficient probabilistic data structure introduced by Burton Bloom in 1970.

IP networks are constantly targeted by new techniques of denial of service attacks (SYN flooding, port scan, UDP flooding, etc), causing service disruption and considerable financial damage. The on-line detection of DoS attacks in the current high-bit rate IP traffic is a big challenge. We propose in this paper an on-line algorithm for port scan detection. It is composed of two complementary parts: First, a probabilistic counting part, where the number of distinct destination ports is estimated by adapting a method called 'sliding Hyper Loglog' to the context of port scan in IP traffic. The algorithm uses a very small total memory of less than 22 kb and has a very good accuracy on the estimation of the number of destination ports (a relative error of about 3.25%), which is in agreement with the theoretical bounds provided by the sliding Hyper Loglog algorithm.

ACKNOWLEDGEMENTS

The support of the resources available online on the web is to be acknowledged here. The project is grateful to all those techno-curious and tech savvy internet users, who raise a query about whether a software do a miracle, or if yes, how to develop that software. Various older models were present on the web, doing somewhat the same thing, but because the title of the project was to develop something with the minimum complexity, a bit of research was necessary. As we know, no project is robust enough to solve all the outer-world problem with 100% accuracy and precision, so is this project. Most importantly, I would acknowledge here the contribution of my Faculty-in-charge, Ms. S UshaSukanya (101591), who constantly supported me towards the end of the project, guiding me what to do every week for the project. At last, thanks to all the tutorial website on the internet, which helped to solve any glitches which occurred during the execution of the program.

TERMINOLOGY

Probabilistic Data Structure

A **NO** is a **NO**, but a yes might not be a yes.

Probabilistic data structures are a group of data structures that are extremely useful for big data and streaming applications. These data structures use hash functions to randomize and compactly represent a set of items

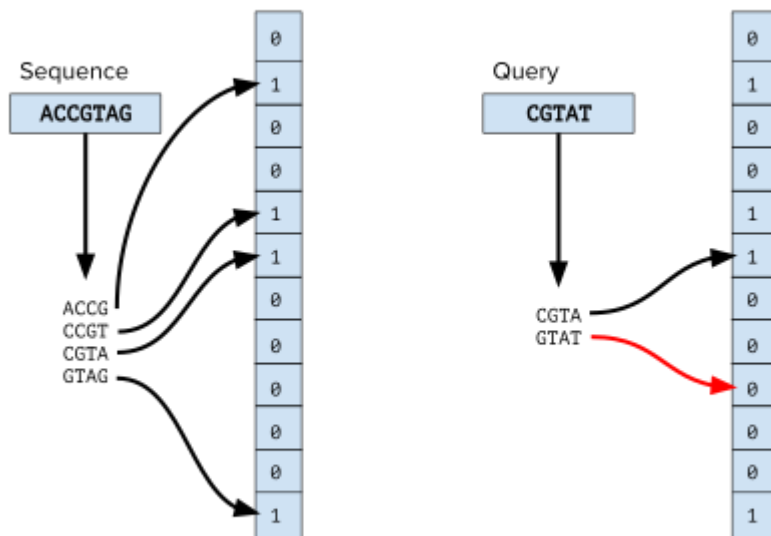
Cardinality

The term "cardinality" is used to mean the number of distinct elements in a data stream with repeated elements. However, in the theory of multisets the term refers to the sum of multiplicities of each member of a multiset.

INTRODUCTION

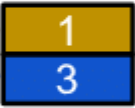
BLOOM FILTER:

- ✗ Bloom Filter is a probabilistic data structure which is used to search an element within a large set of elements in constant time that is $O(K)$ where K is the number of hash functions being used in Bloom Filter. This is useful in cases where:
- ✗ The data to be searched is large
- ✗ The memory available on the system is limited/ low
- ✗ Bloom Filter is memory efficient than a Hash Map with the same performance. The only thing to note is that this is a probabilistic data structure so for a small number of cases, it may give wrong results (which can be limited).



HYPER LOGLOG:

- ✗ Hyper Loglog is an algorithm for the count-distinct problem, approximating the number of distinct elements in a multiset. Calculating the exact cardinality of a multiset requires an amount of memory proportional to the cardinality, which is impractical for very large data sets.
- ✗ The basis of the Hyper Loglog algorithm is the observation that the cardinality of a multiset of uniformly distributed random numbers can be estimated by calculating the maximum number of leading zeros in the binary representation of each number in the set. If the maximum number of leading zeros observed is n , an estimate for the number of distinct elements in the set is 2^{n+1} .

Value	Hash	Hyper-LogLog
hut	1000	
hat	0100	
has	0011	
Hütte	1111	
Hut	1011	
hat	0100	

OBJECTIVE

Bloom Filter:

Security has always been a major concern for networked systems administrators and users. Many approaches have been proposed to achieve the various security goals. In these approaches, a variety of techniques and data structures have been used to address the security concerns in an efficient manner. On the other hand, there are typically umpteen numbers of data items that need to be stored, queried and updated in the network environment. Therefore, the fact is concluded that space and time are two important factors that should be taken into consideration by the security approaches, especially in the specific networks, such as sensor networks, which suffer from severe limitations. A probabilistic data structure that has been widely utilized in this field is Bloom filter (BF), which was introduced by Burton Bloom in 1970. BF is a simple, memory- and time-efficient randomized data structure for succinctly representing a set of elements and supporting set membership queries. These properties of BF make it very attractive to be utilized for many security applications. In recent years, BFs and their variants have been widely used in networking applications, such as resource routing, security, and web caching. This paper provides a survey on the applications of BFs in the field of network security. Note that we only focus on the idea behind the approaches without discussing implementation details. It should be noted, however, that our goal of making this survey is not providing an exact classification of security attacks for different networks. But we intend to review where BFs and their variants have been used to improve the efficiency of the different security schemes.

Hyper Loglog:

Denial of service (DoS) attacks are one of the most important issues in network security. They aim to make a server resource unavailable by either damaging data or software or flooding the network with a huge amount of traffic. Thus, the server becomes unreachable by legitimate users, causing a significant financial loss in some cases. Port scan is a particular DoS attack that aims to discover available services on the targeted system. It essentially consists of sending an IP packet to each port and analysing the response to the connection attempts. Definitions found in the literature are unable to provide an absolute quantitative definition of port scan. The attack is rather defined by a comparison to the standard behaviour. The attacker can discover not only available ports (or services) but also more relevant information about the victim such as its operating system, services owners, and the authentication method. Once the system vulnerabilities are identified, a future attack can be launched, engendering important damages. Various port scanning techniques have been developed and are very simple to install in order to launch serious port scan attacks. Nmap [2] is the most known port scan method. It was proposed by Fyodor in 2009. Zmap is a faster scanning method developed by Durumeric et al. in 2013. It can scan the IPv4 address space in less than 45 min using a single machine.

The memory size required by this kind of approach is proportional to the number of flows, which is clearly unscalable and not adapted to the current very high-bit traffic carried by very high-speed links. To overcome this problem, it is necessary to dispense accurate statistics and to generate estimates which require less memory and are based on a faster processing. In this context, some recent probabilistic methods based on Bloom filters have been proposed. A Bloom filter is an efficient data structure of a limited size that guarantees fast processing, thanks to the use of hash functions.

ADVANTAGES

Bloom Filter:

- ✗ The Time Complexity associated with Bloom Filter Data Structure is $O(k)$ during Insertion and Search Operation where k is the number of hash function that have been implemented.
- ✗ The Space Complexity associated with Bloom Filter Data Structure is $O(m)$ where m is the Size of the Array.
- ✗ While a hash table uses only one hash function, Bloom Filter uses multiple Hash Functions to avoid collisions.

Hyper Loglog:

- ✗ It allows us to estimate unique items within a large dataset by recording the longest sequence of zeroes within that set.
- ✗ This ends up creating an incredible advantage over keeping track of each and every element in the set. It is an incredibly efficient way to count unique values, with relatively high accuracy

DISADVANTAGES

Bloom Filter:

- ✗ Unlike other Data Structures, Deletion Operations is not possible in the Bloom Filter because if we delete one element, we might end up deleting some other elements also, since hashing is used to clear the bits at the indexes where the elements are stored
- ✗ Besides the obvious false positive potential, the bloom filter can only report yes or no. It can't suggest alternatives for items that might be close to being spelled correctly. A bloom filter has no memory of which bits were set by which items so a yes or no answer is the best we can get with even a yes answer not being correct in some circumstances.

Hyper Loglog:

- ✗ Only provides estimated count.

LITERATURE SURVEY

Sl. No	Title	Author	Algorithm	Limitation	Parameters Considered
1.	Network applications of Bloom Filters : a survey.	BRODERY, A. & MITZENMACHERZ, M. (2002)	Counting Bloom Filter	False Positive Inconsistency	Time Complexity, Accuracy
2.	Bloom-based filters for hierarchical data	KOLONIARI, G. & PITOURA, E. (2003)	Multi level Bloom Filter	Frequent Memory Leaks	Infrastructure Cost, Efficiency

3.	Hardware support for a hash-based IP traceback	SANCHEZ, L., MILLIKEN, W., SNOEREN, A., TCHAKOUNTIO, F., JONES, C., KENT, S., PARTRIDGE, C., & STRAYER, W. (2001)	Depth Bloom Filters	Passive Traceback is cost intensive	Capex Oriented Implementation for enterprises
4.	Counting distinct elements in a data stream	Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan	MinCount	Less accuracy	-
5.	An improved data stream summary: the count-min sketch and its applications	G. Cormode and S. Muthukrishnan	Count Min Sketch	Average Optimality	Accuracy

6.	Two sharp inequalities for power mean, geometric mean, and harmonic mean	2. Y.-M. Chu and W.-F. Xia	Harmonic Mean Correction	Buffer requirement much more than needed for use case	Cost Efficiency
----	--	----------------------------	--------------------------	---	-----------------

METHODOLOGY

BLOOM FILTER:

Information representation and query processing are two core problems of many computer applications, and are often associated with each other. Representation means organizing information according to some formats and mechanisms, and making information operable by the corresponding method. Query processing means making a decision about whether an element with a given attribute value belongs to a given set. For this purpose, BF can be an optimal candidate. A Bloom filter, conceived by Burton Howard Bloom in 1970, is a simple space-efficient randomized data structure for representing a set in order to support membership queries [1]. BFs may yield a small rate of false positives in membership queries; that is, an element might be incorrectly recognized as member of the set. Although Bloom filters allow false positives, for many applications the space savings and locating time constantly outweigh this drawback when the probability of false positive can be made sufficiently small. Initially, BF was applied to database applications, spell checkers and file operations [2–4]. In recent years, BFs have received a great deal of attention in networking applications, such as peer-to-peer applications, resource routing, security, and web caching [5,6]. A survey on the applications of Bloom filters in distributed systems can be found in [7]. BFs are also being used in practice. For instance, Google Chrome uses a Bloom filter to represent a blacklist of dangerous URLs. The idea of standard BF is to allocate vector A of m bits, initially all set to 0, for representing a set $S = \{x_1, x_2, \dots, x_n\}$ of n elements. The BF uses k independent hash functions h_1, h_2, \dots, h_k , each with range $\{0, \dots, m-1\}$. A BF is constructed in two phases: programming phase and querying phase [1,5]. In the programming phase, each element $x \in S$ is hashed by k independent hash functions. Then, all the bits at positions $A[h_i(x)]$ in A are set to 1 for $(1 \leq i \leq k)$. Fig. 1 depicts the pseudocode for insertion of n elements. A particular position in the vector might be set to 1 multiple times, but only the first time has an effect. In the querying phase, to query for an element y , we check the bits at positions $h_i(y)$. If any of the bits at these positions are 0, the element is definitely not in the set. Otherwise, either the element is in the set, or the bits have by chance been set to 1 during the insertion of other elements, resulting in a false positive. Fig. 2 depicts the pseudocode for querying an element. The more elements that are added to the set, the larger the probability of false positives. The percentage of false positive of a Bloom filter can be minimized by tuning the three parameters: (i) number of elements (n) added to generate the Bloom filter. In most cases, this parameter is defined by the application and, thus, cannot be controlled. (ii) Number of bits used in a Bloom filter (m). m can be used in order to minimize false positives but obviously the larger the value of m the less compact representation. (iii) Number of hash functions (k) used to create the Bloom filter. The larger k the higher processing overhead (CPU usage) especially if hash functions perform complex operations. Fig. 3 depicts the mentioned process. In Fig. 3, three elements x_1, x_2 , and x_3 are separately hashed by 3 hash functions and then the corresponding bits in A are set to 1. To check if the element y_1 is in the set approximated by A , we check whether all $A[h_i(y_1)]$ are 1. As depicted in Fig. 3, because the bit position 8 is not 1, we surely conclude that y_1 is not a member of the set. Since all the three-bit positions related to y_2 are set to 1, we conclude

that y_2 is a member, although this may be wrong due to the false positive probability. There is a trade-off between the probability of false positive and the length m of the BF array [1,5]. It has been proven that the probability of false positive (fp) is equal to:

Fig. 2. Pseudocode for querying phase.

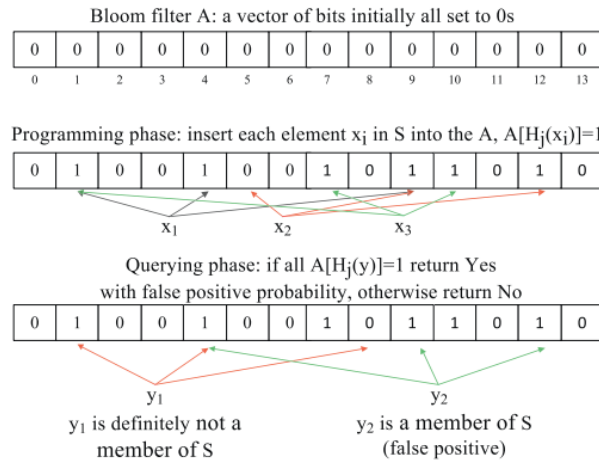
$$fp = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx (1 - e^{-kn/m})^k \quad (1)$$

Input: Set of n elements;
Output: Bloom filter A ;
 A = allocate m bits initialized to 0;
for each x_i **in the set do**
 for $j \leftarrow 1$ **to** k **do**
 $A[h_j(x_i)] \leftarrow 1$;

Fig. 1. Pseudocode for programming phase.

Input: An element y ;
Output: True/False;
for $j \leftarrow 1$ **to** k **do**
 if $A[h_j(y)] = 0$ **return** False;
return True;

Fig. 3. Insert and query operations in standard Bloom filter



HYPER LOGLOG:

The insertion of a data element into a HyperLogLog data structure requires the calculation of a $(p + q)$ -bit hash value. The leading p bits of the hash value are used to select one of the 2^p registers. Among the next following q bits, the position of the first 1-bit is determined which is a value in the range $[1, q + 1]$. The value $q + 1$ is used, if all q bits are zeros. If the position of the first 1-bit exceeds the current value of the selected register, the register value is replaced. Algorithm 1 shows the update procedure for inserting a data element into the Hyper Loglog sketch. Algorithm 1 Insertion of a data element D into a Hyper Loglog data structure that consists of $m = 2^p$ registers. All registers $K = (K_1, \dots, K_m)$ start from zero.

procedure InsertElement(D)
 $h \leftarrow \text{hash}(D)$
 $i \leftarrow 1 + \text{first_one_position}(h)$
 if $i > K_i$ then $K_i \leftarrow i$ end if
end procedure

The described element insertion algorithm makes use of what is known as stochastic averaging [16]. Instead of updating each of all m registers using m independent hash values, which would be an $O(m)$ operation, only one register is selected and updated, which requires only a single hash function and reduces the complexity to $O(1)$.

3 A HyperLogLog sketch can be characterized by a parameter pair (p, q) . The precision parameter p controls the relative error while the second parameter defines the possible value range of a register. A register can take all values starting from 0 to $q + 1$, inclusively. The sum $p + q$ corresponds to the number of consumed hash value bits and defines the maximum cardinality that can be tracked. Obviously, if the cardinality reaches values in the order of 2^{p+q} , hash collisions will become more apparent and the estimation error will increase drastically. Algorithm 1 has some properties which are especially useful for distributed data streams. First, the insertion order of elements has no influence on the final HyperLogLog sketch. Furthermore, any two HyperLogLog sketches with same parameters (p, q) representing two different sets can be easily merged. The HyperLogLog sketch that represents the union of both sets can be easily constructed by taking the register-wise maximum values as demonstrated by Algorithm 2.

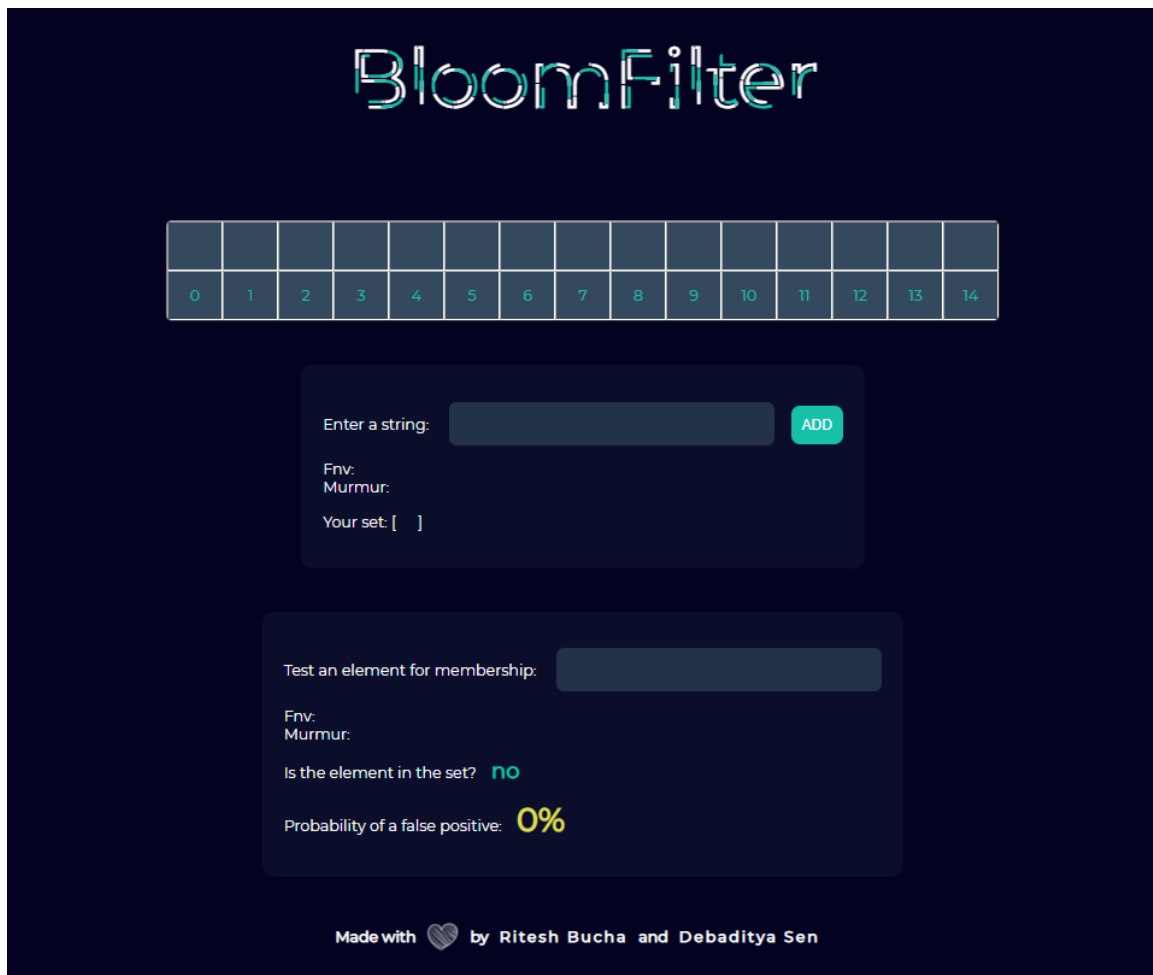
Algorithm 2 Merge operation for two HyperLogLog sketches with register values $K_1 = (K_{11}, \dots, K_{1m})$ and $K_2 = (K_{21}, \dots, K_{2m})$ representing sets S_1 and S_2 , respectively, to obtain the register values $K_0 = (K_{01}, \dots, K_{0m})$ of a HyperLogLog sketch representing $S_1 \cup S_2$.

function Merge(K_1, K_2)
 for $i \leftarrow 1$ to m do
 $K_{0i} \leftarrow \max(K_{1i}, K_{2i})$
 end for
 return K_0
end function

At any time a (p, q) -HyperLogLog sketch can be reduced to a (p_0, q_0) -HyperLogLog data structure, if $p_0 \leq p$ and $p_0 + q_0 \leq p + q$ is satisfied (see Algorithm 3). This transformation is lossless in a sense that the resulting HyperLogLog sketch is the same as if all elements would have been recorded by a (p_0, q_0) -HyperLogLog sketch right from the beginning. A $(p, 0)$ -HyperLogLog sketch corresponds to a bit array as used by linear counting [17]. Each register value can be stored by a single bit in this case. Hence, linear counting can be regarded as a special case of the Hyper Loglog algorithm for which $q = 0$.

MODULE DESCRIPTION

1. Bloom Filter { <https://bloomfilter.bucharitesh.in> }



The screenshot shows the BloomFilter web application interface. At the top, the title "BloomFilter" is displayed in a stylized font. Below the title is a grid of 15 empty slots, numbered 0 to 14. The main interface consists of two sections. The first section, titled "Enter a string:", has a text input field and an "ADD" button. Below this, there are labels for "Fnv:", "Murmur:", and "Your set: []". The second section, titled "Test an element for membership:", has a text input field. Below this, there are labels for "Fnv:", "Murmur:", and "Is the element in the set?". The result "no" is displayed in red. At the bottom, the text "Probability of a false positive: 0%" is shown in red. The footer text reads "Made with ❤ by Ritesh Bucha and Debaditya Sen".

✕ **Aim/Objective:** To test an element for membership from a given set of elements or a continuous data string that it is present in the set or not.



✖ **Input:** To test an element for membership from a given set of elements or a continuous data string that it is present in the set or not.

✖ Algorithm:

input: Input file *inputFile*

input: Number of estimator buckets N

data: Vector of N buckets $\vec{M} = (M_0, \dots, M_{N-1})$

output: Estimation E of number of different items in the input file

$M_{\text{index}} = 0, 0 \leq \text{index} < N$

while not at end of file *inputFile* **do**

dataChunk = readChunk(*inputFile*)

foreach *item* in *dataChunk* **do**

hash = hashFunction(*item*)

 [*remainder*, *index*] = splitHash(*hash*)

nleadingZeros = countLeadingZeros(*remainder*)

$M_{\text{index}} = \max(M_{\text{index}}, n\text{leadingZeros})$

end

end

$$E = \text{Alpha} \times N^2 \times \left(\sum_{i=0}^{N-1} 2^{-M_i} \right)^{-1}$$

× Output:

Tests the dataset for membership and gives a probabilistic yes or a definite no (in accordance with the false positive rate)

Test an element for membership:

Fnv: 10
Murmur: 4

Is the element in the set? maybe!

Probability of a false positive: 75%

Test an element for membership:

Fnv: 3
Murmur: 7

Is the element in the set? no

Probability of a false positive: 75%

NOVELTY

Bloom Filter:

- ✗ Does not store the actual items. In this way it's space efficient. It's just an array of integers
- ✗ Saves expensive data scanning across several servers depending on the use case
- ✗ Error typically error $\leq 2\%$

Hyper Loglog:

- ✗ Estimate cardinalities well beyond 10^9 with a relative accuracy (standard error) of 2% while only using 1.5kb of memory
- ✗ Can be distributed and paralleled
- ✗ Fast ($O(1)$)

CONCLUSION & LIMITATIONS

Bloom Filter:

A Bloom filter is a simple space-efficient representation of a set or a list that handles membership queries. As we have seen in this survey, there are numerous networking problems where such a data structure is required. Especially when space is an issue, a Bloom filter may be an excellent alternative to keeping an explicit list. The drawback of using a Bloom filter is that it allows false positives. Their effect must be carefully considered for each specific application to determine whether the impact of false positives is acceptable. This leads us back to: The Bloom filter principle: Wherever a list or set is used, and space is at a premium, consider using a Bloom filter if the effect of false positives can be mitigated. There seems to be plenty of room to develop variants or extensions of Bloom filters for specific applications. For example, we have seen that the counting Bloom filter allows for approximate representations of multi-sets or dynamic sets that change over time through both insertions and deletions.

Hyper Loglog:

The Hyper loglog algorithm gives us the possibility of having **really small structures capable of calculating the cardinality of really big sets** for the price of a minimal error. As we have said before, each different problem will tell us if we need the exact data or if we can tolerate this low error rate to take advantage of the low size that this structure provides.

FUTURE ENHANCEMENT

Bloom Filter:

Bloom filter has the advantage that space efficiency and query time efficiency are much better than ordinary algorithms. A hash table can be used to determine whether an element is in a collection or not, and the retrieval is very efficient. Nevertheless, Bloom filter does the same thing with only one fourth or one eighth or even lower of the space complexity of the hash table approach. Bloom filter can insert elements, but non-existing elements can be deleted. Bloom filter is impossible to have a false negative, as long as the elements exist in Bloom filter. However, Bloom filter has one major shortcoming that it can produce false positive, and the more similar elements there are, the greater the false positive rate will be.

Hyper Loglog:

HyperLogLogPlusPlus is an enhancement of Hyper Loglog.

- ✗ Uses 64-bit integers rather than 32-bit
- ✗ Sparse data structure rather than one huge array
- ✗ Better bias correction algorithm at lower cardinalities

REFERENCES

- × <https://www.geeksforgeeks.org/>
- × <https://stackoverflow.com/>
- × <https://www.github.com/>
- × <https://www.srmist.edu.in/>
- × <https://www.researchgate.net/>
- × <https://www.arxiv.org/>
- × <https://scholar.google.com/>
- × <https://google.bucharitesh.in/>