EXPERIMENT 1

NAME	**Shreya Shetty**
UID	**2019140059**
CLASS	**TE IT**
BATCH	**A**
SUBJECT	**Big Data Analytics Lab**

AIM: Study of any latest research paper on memory efficient data structure for big data

TOPIC: Bloom Filter

RESEARCH PAPER TITLE: IDS based fake content detection on Social Network Using Bloom Filtering

AUTHORS: P. Manju Bala; S. Usharani; M. Aswin

DOI: 10.1109/ICSCAN49426.2020.9262360

PUBLISHED IN: 2020 International Conference on System, Computation, Automation and Networking (ICSCAN)

INTRODUCTION:

Spam delivery is the most common issue, in today's Online Social Networks. Numerous long range interpersonal communication clients experience the social phony substance in explicit structure. Those spammers can be robotized through spamming botnets, a genuine individual or a phony record. The social spammers will utilize breaking stories to plant the vindictive substance through the web joins with the problematic substance. The Spammers send notes to the any regular gatherings or fan pages on informal community from fake records. Such notes may incorporate implanted the site connects to oppressive substance or any item based locales (Amazon). Most of the modern spam-filtering Techniques are deployed in the Receiver Side only. They are good at filtering spam for end users, but spam messages still keep wasting Internet bandwidth and the storage space of servers. The intrusion detection system to monitor the SMTP sessions in a university campus, and track the number and the uniqueness of the recipients' online social Networks addresses in the outgoing messages from each individual internal host as the features for detecting spamming bots. Due to the huge number of Spams observed in the SMTP sessions, it can be stored and managed efficiently in the Bloom filters.

BLOOM FILTER:

A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. For example, checking availability of username is set membership problem, where the set is the list of all registered username. The price we pay for efficiency is that it is probabilistic in nature that means, there might be some False Positive results. False positive means, it might tell that given username is already taken but actually it's not.

PROPERTIES OF BLOOM FILTERS:

1. Unlike a standard hash table, a Bloom filter of a fixed size can represent a set with an arbitrarily large number of elements.
2. Adding an element never fails. However, the false positive rate increases steadily as elements are added until all bits in the filter are set to 1, at which point all queries yield a positive result.
3. Bloom filters never generate false negative result, i.e., telling you that a username doesn't exist when it actually exists.
4. Deleting elements from filter is not possible because, if we delete a single element by clearing bits at indices generated by k hash functions, it might cause deletion of few other elements.

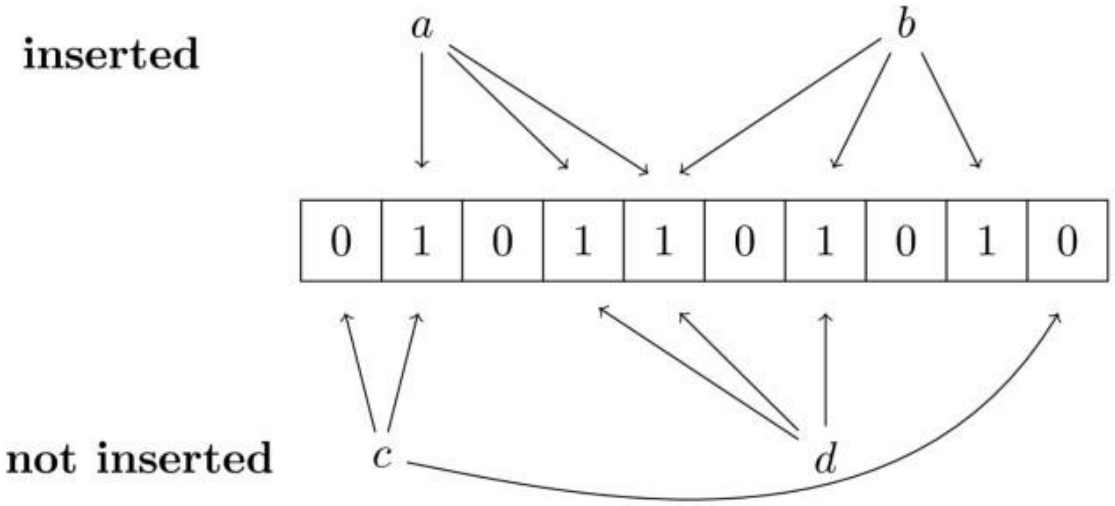
SPACE EFFICIENCY:

Large list of items in a set for purpose of set membership can be stored in hashmap, tries or simple array or linked list. All these methods require storing item itself, which is not very memory efficient. For example, if we want to store "college" in hashmap we have to store actual string "college" as a key value pair {some_key: "college"}. Bloom filters do not store the data item at all. They use bit array which allow hash collision. Without hash collision, it would not be compact.

BLOOM FILTER ALGORITHM:

- The Bloom filter is a probabilistic data structure supporting dynamic set membership queries with false positives.
- It allows us to identify in an extremely compact way all *k*-mers that are present more than once in a data set, while allowing a low rate of false positives.
- The Bloom filter is a bit array B, initialized to be 0 at every position. We also define a set of d hash functions, h_1, ..., h_d, where each hash function maps a given k-mer x to a location in B.
- In order to insert a *k*-mer *x* into the Bloom filter, we set all of the *d* corresponding locations in *B* to be 1; that is, we set $B[h_i(x)] = 1$ for $i = 1, ..., d$.
- Then, to determine whether a *k*-mer *y* has been inserted, we simply check whether each of the corresponding hash positions is 1: i.e., whether $B[h_i(y)]$ are all set to 1 for $i = 1, ..., d$. If this is the case, then we infer that *y* has probably been seen before.

- By construction, this procedure correctly identifies every *k*-mer that is present more than once in the data; however, the cost of very efficient memory usage is that we accept a low rate of false positives in which we infer that *y* has been seen previously, but in fact it has not.
- The Bloom filter has a tradeoff between memory usage (i.e., the number of bits used) and the false positive rate.
- When storing *n* *k*-mers in a Bloom filter of *m* bits, and using *d* hash functions, the false positive rate is approximately $(1 - e^{-d\frac{n}{m}})^d$. Given *n* and *m*, the optimal number of hash functions that minimizes the false positive ratio is $d \approx \frac{m}{n} \ln 2$[19].
- In practice we may have a rough idea in advance about *n*, the number of *k*-mers, and we can select *m* as a fixed multiple of *n*. For example using $m = 8 \cdot n$ (which corresponds to storing one byte per *k*-mer), and $d = 5$ gives a false positive ratio of 2.16%.
- Consider an example of a Bloom filter in the below figure with three hash functions. The k-mers a and b have been inserted, but c and d have not. The three hash functions are represented with arrows, and the bits corresponding to the hashes for a and b have been set to 1. The Bloom filter indicates correctly that k-mer c has not been inserted since not all of its bits are set to 1. However, k-mer d is an example of a false positive: it has not been inserted, but since its bits were set to 1 by the insertion of a and b, the Bloom filter falsely reports that d has been seen already.



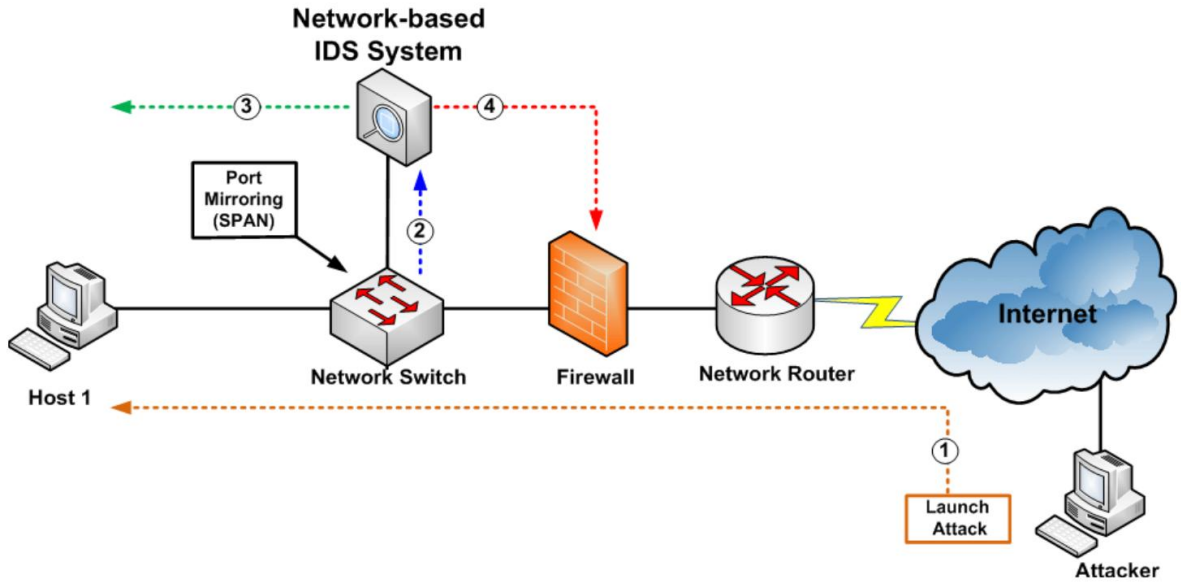
- To count all non-unique k-mers a Bloom filter B and a simple hash table T to store k-mers is used. The Bloom filter keeps track of k-mers encountered so far and acts as a "staging area", while the hash table stores all the k-mers seen at least twice so far.
- The idea is to use the memory-efficient Bloom filter to store implicitly all k-mers seen so far, while only inserting non-unique k-mers into the hash table.
- Initially both the Bloom filter and the hash table are empty. All k-mers are generated sequentially from the sequencing reads.

- In most applications we do not need to distinguish between a k-mer and its reverse complement sequence.
- Thus, as we read in each k-mer we also consider the reverse complement of that k-mer and then work with whichever of the two versions is lexicographically smaller (we refer to the smaller sequence as the "canonical k-mer").
- For each k-mer, x, we check if x is in the Bloom filter B. If it is not in B then we update the appropriate bits in B to indicate that it has now been observed. If x is in B, then we check if it is in T, and if not, we add it to T.
- This scheme guarantees that all k-mers with a coverage of 2 or more are inserted into T. However a small proportion of unique k-mers will be inserted into T due to false positive queries to B.
- After the first pass through the sequence data, one can re-iterate over the sequence data to obtain exact counts of the k-mers in T and then simply delete all unique k-mers.
- The time spent on the second round is at most 50% of the total time, and tends to be less since hash table lookups are generally faster than insertions. A detailed pseudocode is given in the below figure.

Algorithm 1 Bloom filter k-mer counting algorithm

```
1: B ← empty Bloom filter of size m
2: T ← hash table
3: for all reads s do
4:    for all k-mers x in s do
5:       x_rep ← min(x, revcomp(x)) //x_rep is the canonical k-mer for x
6:       if x_rep ∈ B then
7:          if x_rep ∉ T then
8:             T[x_rep] ← 0
9:       else
10:         add x_rep to B
11: for all reads s do
12:    for all k-mers x in s do
13:       x_rep ← min(x, revcomp(x))
14:       if x_rep ∈ T then
15:          T[x_rep] ← T[x_rep] + 1
16: for all x ∈ T do
17:    if T[x] = 1 then
18:       remove x from T
```

INTRUSION DETECTION SYSTEM:



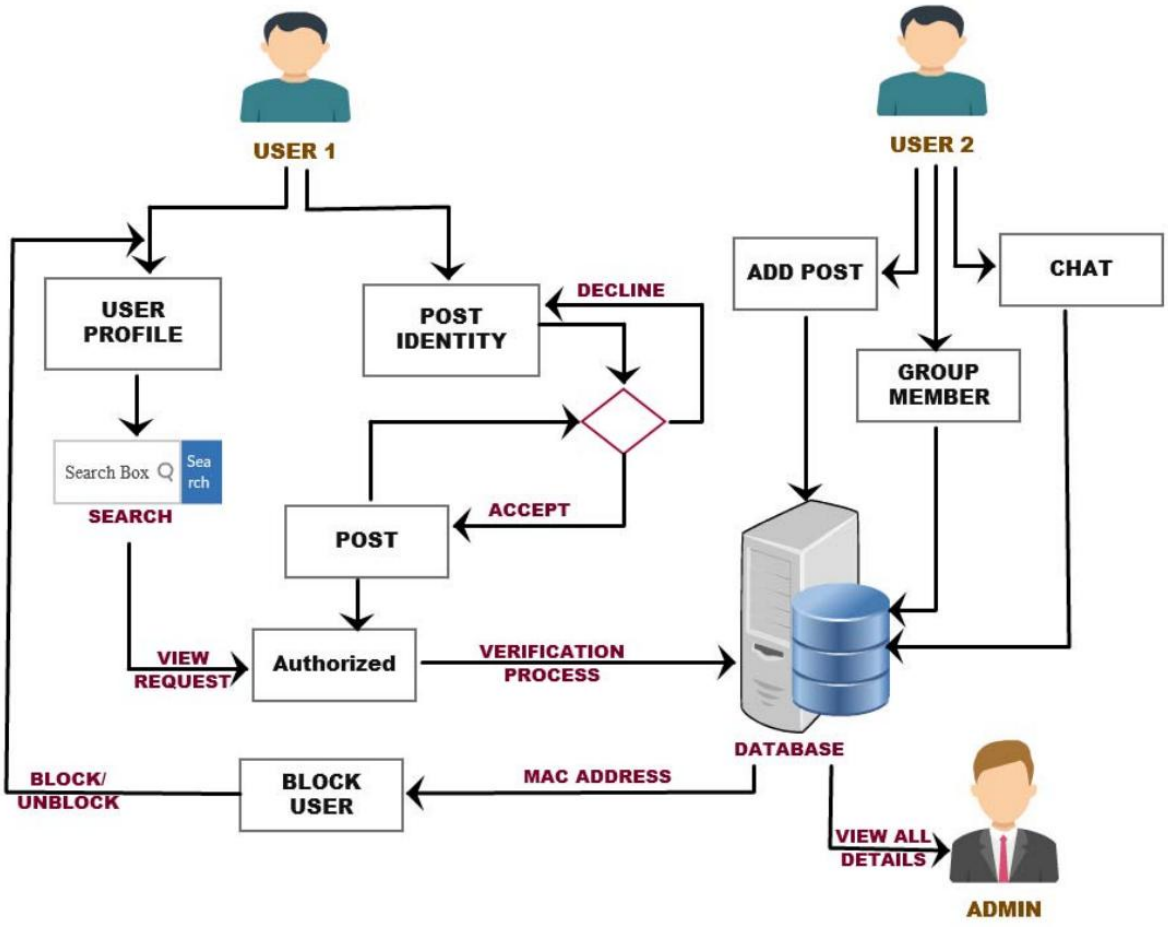
- Intrusion detection system distinguishes vindictive traffic on a system. It needs stable system access to break down all the traffic happened while information transmission. Any malignant movement or infringement is commonly revealed or gathered midway utilizing a security data and occasion the executives framework.
- An IDS attempts to recognize malevolent movement.
- Intrusion Detection System provides the following functions to security professionals:
    - Checking the activity of switches, firewalls, key administration servers and documents that are required by other security controls planned for distinguishing, forestalling or recouping from cyberattacks;
    - Giving directors an approach to tune, compose and comprehend pertinent working framework review trails and different logs that are regularly in any case hard to follow or parse;
    - Giving an easy to understand interface so non-master staff individuals can help with overseeing framework security;
    - Counting a broad assault signature database against which data from the framework can be coordinated;
    - Perceiving and announcing when the IDS distinguishes that information documents have been modified;
    - Producing a caution and notifying that security has been broken;
    - Responding to interlopers by blocking them or hindering the server.
- Intrusion Detection System can enable the undertaking to accomplish administrative consistence.
- Intrusion Detection System can improve security reaction.

SPAMMING BOTNETS:

- The term botnet is got from the words robot and network. Users are often unaware of a botnet infecting their system.

- Botnet is a collection of internet-connected devices, which may include personal computers (PCs), servers, mobile devices and internet of things (IoT) devices that are infected and controlled by a common type of malware.
- Contaminated gadgets are controlled remotely by risk on-screen characters, frequently cybercriminals, and are utilized for explicit capacities, so the malignant tasks remain covered up to the client.
- Botnets are ordinarily used to send email spam, take part in click extortion battles and create pernicious traffic for distributed disavowal of-administration (DDoS) assaults.
- The botnet malware commonly searches for helpless gadgets over the web, instead of focusing on explicit people, organizations or enterprises.
- The goal for making a botnet is to taint whatever number associated gadgets as could be expected under the circumstances and to utilize the assets of those gadgets for mechanized undertakings.
- Botnets will also send the encrypted disruptive files.
- Existing system IDS will not able to find the encrypted packets which are sent by the botnets, which leads to the storage of large collection spams and uses the more internet bandwidth.

ARCHITECTURE DIAGRAM & METHODOLGY:



- Bloom Filtering spam technique is used at the sender side itself, due to huge number of spams stored in the SMTP session and the usage of large number internet bandwidth.
- It checks all the information provided at the central repository client server.
- It checks the encrypted spam files shared by the spammers or botnets to the users.
- List of spamming bots is reported to network administrators in the computer center.
- Memory usage is reduced by neglecting the unnecessary spams from the server and improves the internet bandwidth.

The following are the main 6 modules implemented -

1. *Registration & Login*
   User can create an account on this site by executing registration process. After registration, user can login by entering the correct credentials and then server allows going to inside the websites or else user name or password alert is generated by server.
2. *TimeLine Add*
   User can post some image contents to share within friends lists. This post will be displayed on the timeline of their friends list.
3. *Friend Request*
   User enters some of the string into the search bar and then sent this string as request to the server. The server automatically checks the possibility of results and then responds to the requested user. If user wants to friend any member, they can send friend request.
4. *Profile Matching*
   This module is executed by server whenever user makes friends requests. The server gets another user name and profile information from database and collects profile details of requested users. The server then matches both the profiles by using profile matching algorithm and generates a single value based on parameter matching. Based on this user might accept the request or reject it.
5. *Secure Profile View*
   Users can view profiles of people from their friend lists by getting the profile key from the profile owner and then view the profile information.
6. *Group* Actions
   Users are able to create group for sharing information with in specified users, the server automatically create group key used to perform group actions.

CONCLUSION:

There were many spam filtering techniques introduced with specific functionality constraints. By using Bloom Filter algorithm, we can improve the usage of internet bandwidth and memory efficiency through reducing huge number of encrypted spams in the SMTP Sessions by tracking the number of unique users or botnets. It will reduce the network traffic and creates smooth way of data transmission between the users.

REFERENCES:

1. P. M. Bala, S. Usharani and M. Aswin, "IDS based fake content detection on Social Network Using Bloom Filtering," 2020 International Conference on System, Computation, Automation and Networking (ICSCAN), 2020, pp. 1-6, doi: 10.1109/ICSCAN49426.2020.9262360.
2. https://en.wikipedia.org/wiki/Bloom_filter
3. https://blog.medium.com/what-are-bloom-filters-1ec2a50c68ff
4. https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/