Future Generation Computer Systems 82 (2018) 440–449

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Bloom filter based optimization scheme for massive data handling in IoT environment

Amritpal Singh [a], Sahil Garg [a,*], Shalini Batra [a], Neeraj Kumar [a], Joel J.P.C. Rodrigues [b,c,d,e]

[a] Computer Science & Engineering Department, Thapar Institute of Engineering and Technology (Deemed University), Patiala (Punjab), India
[b] National Institute of Telecommunications (Inatel), Brazil
[c] Instituto de Telecomunicações, Portugal
[d] ITMO University, Russia
[e] University of Fortaleza (UNIFOR), Brazil

ARTICLE INFO

Article history:
Received 30 June 2017
Received in revised form 22 September 2017
Accepted 14 December 2017
Available online 21 December 2017

Keywords:
Internet of Things
Big data analytics
Probabilistic data structures
Bloom filter
In-stream data processing

ABSTRACT

With the widespread popularity of big data usage across various applications, need for efficient storage, processing, and retrieval of massive datasets generated from different applications has become inevitable. Further, handling of these datasets has become one of the biggest challenges for the research community due to the involved heterogeneity in their formats. This can be attributed to their diverse sources of generation ranging from sensors to on-line transactions data and social media access. In this direction, probabilistic data structures (PDS) are suitable for large-scale data processing, approximate predictions, fast retrieval and unstructured data storage. In conventional databases, entire data needs to be stored in memory for efficient processing, but applications involving real time in-stream data demand time-bound query output in a single pass. Hence, this paper proposes Accommodative Bloom filter (ABF), a variant of scalable bloom filter, where insertion of bulk data is done using the addition of new filters vertically. Array of m bits is divided into b buckets of l bits each and new filters of size 'm/k' are added to each bucket to accommodate the incoming data. Data generated from various sensors has been considered for experimental purposes where query processing is done at two levels to improve the accuracy and reduce the search time. It has been found that insertion and search time complexity of ABF does not increase with increase in number of elements. Further, results indicate that ABF outperforms the existing variants of Bloom filters in terms of false positive rates and query complexity, especially when dealing with in-stream data.

1. Introduction

We have entered into massive data processing age where enormous amount of data is being generated dynamically by different geographically separated devices. Usage of Smart phones, GPS devices, laptops, Facebook, Twitter, *etc.*, produce a huge amount of data. Machines which include Internet-enabled instruments and sensors, capture rapidly growing volumes of data [1].

The next generation of computing will be based on Internet of Things (IoT) which collaborate with each other without any human intervention and in future it is expected to perform intelligent decision making too [2]. The basic idea behind this concept is to connect all physical devices in home, at work place, in moving vehicles, *etc.*, through the internet and then infer the useful information from the collected data and make decisions accordingly [3,4]. Using latest analytical techniques such as-data mining, machine learning, predictive analysis, statistical analysis, *etc.*, one can analyze data independently for better and faster decision making [5,6].

Presently available hardware and software tools are not capable enough to handle the emerging paradigm of Big data, where complexity is beyond the ability of present computational techniques [7]. Data is generated from various sources (Variety) which may be structured or unstructured (such as sensors, social media, surveillance, video and image archives, Internet texts and documents, web logs, and medical records). Further the amount of data generated may vary from terabytes to zeta-bytes (Volume) where data arrives in batches or streams (Velocity). Another essential component of Big data is high value (Value) associated with it and trust generated for business decision making (Veracity) [8]. The need for Big data analytics is clearly demonstrated in Fig. 1.

Efficient measures are required for turning "big data" into "big insights". Interrelated data analysis provides true insight to the

* Corresponding author.
E-mail addresses: amritpal.singh203@gmail.com (A. Singh), garg.sahil1990@gmail.com (S. Garg), sbatra@thapar.edu (S. Batra), neeraj.kumar@thapar.edu (N. Kumar), joeljr@ieee.org (J.J.P.C. Rodrigues).

https://doi.org/10.1016/j.future.2017.12.016
0167-739X/

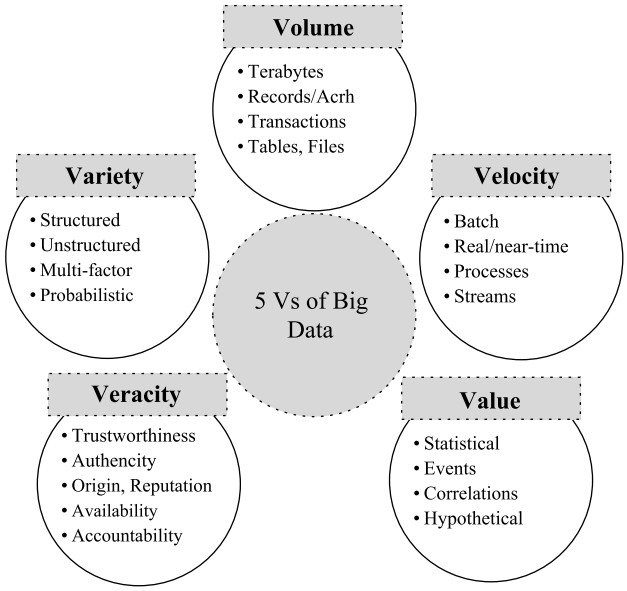

Fig. 1. Need for Big data analytics [9].

correlations and hidden principles for various applications such as-health analysis, business predictions, terrorist activity detection, industrial research, *etc.* Innovation in analytics, applications, and visualization tools is critical for the widespread adoption of Big data [10,11]. With the manifold increase in recent data generation methods especially for streaming data, the major challenge is how to store and query such huge amount of data in single pass [12,13]. These challenges are manifested in all aspect, *i.e.*, technology, techniques, methods, processing, *etc.* [14]. One of the most important and major challenge is to handle such a huge amount of data, generated at a high pace, especially when data is available only for short duration of time. One of the alternatives for handling in-stream data is through the usage of probabilistic data structures (PDS) as they can handle scalable and dynamic data efficiently [15–17].

The Bloom Filter (BF) [18] consists of a set $S = \{s_1, s_2, \ldots, s_n\}$ of n elements from a universe U. Initially a m bits array is considered where all elements are set to 0. Then k independent hash functions $h_1, h_2, \ldots, h_k$ with their value ranging between $1, 2, \ldots, m$ are considered. Corresponding to every element $x \in S$, the bits $BF[h_i(x)]$ are set to 1 for $1 \leq i \leq k$. For a given query y, membership is examined by probing the complete BF to check whether the bits at positions $h_1(y), h_2(y), \ldots, h_k(y)$ are set to 1. If all $h_i(y)(1 \leq i \leq k)$ bits are 1, the query result is TRUE. If even a single bit returns zero, then y is definitely not a member of S. False positives in BF can be defined by the user in accordance with the application being considered. Mathematically false positive (fp) in BF is:

$$fp = (1 - e^{-kn/m})^k. \quad (1)$$

BF proposed by Burton Bloom [19] had limited scope in dealing with dynamic datasets. As the size of incoming data increases, static BFs need increase in array size, otherwise the number of collisions will increases exponentially [20]. Dynamic and scalable BFs deal with the scalability problem by adding extra filters. In dynamic BFs, an array of size same as that of initial array, *i.e.*, an array of m bits is added repeatedly to accommodate the ever rising data [21]. In scalable BFs, slices are added at each repetition [22,23]. One of the shortcoming of dynamic BF is how to control the compound error rate of filter as the new filters are added since the size of new filter depends on the size of previous filter.

1.1. Organization

The plan of this paper is as follows: Section 2 provides the literature study on bloom filters and its variants. The concept formulation (motivation) for the proposed scheme is also discussed in this section. In Section 3, the proposed approach, *i.e.*, ABF is discussed in detail. Section 4 provides experimental results and comparison with the existing approaches. Finally, Section 5 concludes the paper with discussion in Section 6.

2. Related work

With the advent of Big data and IoT, the relevance of data processing techniques has increased significantly, which further attracted researchers to contribute in this field [24,25]. To have a broader view of this domain, some of the existing techniques are discussed in this section.

Cai et al. [26] identified the challenges and opportunities associated with IoT big data storage systems in cloud computing. The authors suggested new data processing patterns for massive datasets which can cope with the demerits of existing approaches and support the complex operations such as-acquisition, management, and processing along with the unstructured data in an efficient manner. Zhou [27] proposed a novel data-driven delay estimation model for a practical cloud media with heavy traffic. The method proposed by the author tend to achieve the balance between the computational complexity and estimation accuracy while computing the delay in a small dataset. However, its complexity may increase inevitably when the number of servers is huge, *i.e.*, in the case of large-scale media cloud. Similarly, in an another work by Zhou [28], state-dependent delay announcement scheme for cloud mobile media has been proposed in order to provide the users a quality of experience (QoE). In this paper, author considered two classical scenarios for validating the delay announcement in cloud mobile media where the first scenario deals when cloud servers have enough computational capacity and second scenario considers a case when cloud servers do not have enough capacity. However, the first case is an ideal case where the delay announcement scheme reduce the users' apprehension from the delay uncertainty whereas second case deals with a less idealized scenario where delay announcement scheme aims to reduce the average waiting by advising some users to leave the waiting queue.

Probabilistic data structures (PDS) are extremely useful for such applications where delay can occur while performing membership queries. They use hash functions to randomize the items. Different variants of PDS exists in literature such as-Bloom filters (BF) [19], Quotient filter (QF) [29], Count Min Sketch (CMS) [30], HyperLogLog (HLL) [31], etc. However, BF is the most widely used PDS for performing complex queries in a time efficient manner. BF is a hashing based probabilistic data structure which was initially used to represent words in a dictionary. Earlier BF was used in limited domains especially in networks and security fields, but as the amount of incoming data is increasing enormously, probabilistic data structures have been considered as effective alternate to deterministic data structures by both academia and industry. Some of the domain where BF has been used include accounting, monitoring, load balancing, policy enforcement, routing, filtering, and security [32,33]. In networks, BFs have been successfully used in peer-to-peer (P2P) networks for content summarize for global collaboration [34,35], in various algorithms for routing and locating resources [36], and web cache information [34] communication.

Till date several techniques have been proposed in the literature which suggest the role of BFs in IoT environment [37]. Kalmar et al. [38] proposed a Context-Aware Addressing and Routing scheme (CAEsAR) where BFs are used to represent context information inside the IoT domain. In their paper, they integrated the

BFs along the Routing Protocol for Low Power and Lossy Networks (RPL) tree, through a lightweight operation that spares the resource constrained IoT nodes. Alrawais et al. [39] propounded a fog based novel approach for the distribution of certificate revocation information among IoT devices. In their scheme, BF had been used for the reduction of revocation list size with acceptable overhead.

Munoz and Leone [40] designed a new architecture of Fragmented-Iterated BFs in order to route events in a network of constrained sensing devices. Their proposed approach consists of two types of BFs, *i.e.*, Fragmented BFs (FBF) and Iterated BFs (IBF) where FBF deals with the conjunctive and disjunctive set of events and IBF discards all the single events that do not match to the subscribed list. Zhang et al. [41] proposed a scheme for unknown tag identification in large-scale IoT systems. Here, the authors used BF to reduce the data transmission during the tag identification process. Amoretti et al. [42] designed a distributed naming service approach for IoT which used BFs to create compact names from node descriptions. Further, message propagation strategies have been used to publish and discover information within the network whereas distributed caches to store names within the network. Li et al. [43] proposed a Software-defined Networking (SDN) based OpenFlow control channel for investigating the potential threats of Man-in-the-Middle (MitM) attacks on IoT-Fog Networks. In this approach, the authors used BFs to enhance the security of the OpenFlow channel by detecting MitM attacks such as-packet modification, modified flow paths, etc. Further, Table 1 summarizes several application domains where BF has been applied to solve different problems.

2.1. Dynamic bloom filter (DBF)

Guo and Wu proposed dynamic BF, a variant of standard BF to deal with streaming data [21]. The basic idea of DBF is to extend the m bit BF into $r \times n$ array by adding r slices of m bits [23].

Fig. 2 shows insertion of elements in DBF where DBF_0 is starting array and as size of data grows $DBF_{(1,2\ldots n)}$ arrays of same size are added in the initial array DBF_0. To represent N elements, the size of bit array required is multiple of m (initial size of bit array). Only variation in DBF, when compared to simple BF, is that a new filter of size m is added whenever false positive rate approaches the defined threshold. N_0 (the elements under consideration for each BF) depends on the upper bound of defined false positive rate f_0. In DBF, parameter $r = \lfloor (N/N_0) \rfloor + 1$ is used to define the number of times array of m bits is added to the original array. With N elements, total size of BF required is:

$$M_{DBF} = r \times m = (\lfloor (N/N_0) \rfloor + 1) \times m. \tag{2}$$

For N elements with k hash functions, m bit initial filter and t elements in the last filter, false positive rate f_{DBF} is:

$$F_{DBF} = 1 - \{(1 - (1 - e^{-kN_0/m})^k)^{(r-1)} \times (1 - (1 - e^{-kt/m})^k)\}. \tag{3}$$

Time complexity of adding an element is $O(k)$ same as that of simple BF but in querying an element from DBF, search starts from rth bit array till first array in reverse order.

2.2. Scalable bloom filter (SBF)

The scalable bloom filter (SBF) [22,23] represents dynamic datasets efficiently and solves the scalability problem of DBF effectively. With the increase in the size of dataset, the SBF controls the increase in false positive rate by adding a new slice with size more than that of the existing one. Like DBF, SBF is also made up of series of one or more BFs but of different sizes, while assuring a minimum false positive probability P_0. Fig. 3 shows insertion of elements in SBF. In addition to the initial bit size m and target P_0, SBF includes the expected growth rate s and the error probability tightening ratio ρ. The SBF starts with one filter with m_0 size and error probability P_0. When this filter exceeds its filling capacity, a new filter of m_1 size is added and $P_1 = P_0\rho$ is error probability for new filter, where ρ is the tightening ratio with $0 < \rho < 1$. At a given time, for total i filters, error probabilities are given as:

$$P_0, P_0\rho, P_0\rho^2, \ldots, P_0\rho^{i-1}. \tag{4}$$

The compounded error probability for the SBF is:

$$P_{SBF} = 1 - \prod_{x=1}^{i} (1 - P_0\rho^{x-1}). \tag{5}$$

Each successive BF is created with a tighter minimum error probability on a geometric progression, so that the compounded probability over the whole series converges to P_0.

When collision exceeds the defined threshold, a new filter of m_1 slices is added. The size of m_1 is given as:

$$m_1 = m_0 + s^i \tag{6}$$

$$s^i = \frac{m}{k}. \tag{7}$$

Another important aspect covered in SBF is that the size of new slice m_i is decided by considering maximum threshold for false positives.

$$m_i = \log_2 P_i^{-1} = m_0 + i \times log_2\rho^{-1}. \tag{8}$$

For flexible growth in SBF size, exponential growth factor s is added, where size of new slice added in SBF is increased by factor s with respect to the initial filter size, *i.e.*, m_0. So size of i filters in SBF respectively are: $m_0, m_0s, m_0s^2, .., m_0s^{i-1}$. When the threshold fill ratio f for one filter is achieved another filter is added to it, with a well defined growth parameter s.

The value of s is decided on experimental results and the requirement of application. As shown in [22], the values $s = 2$ for slow growing data, $s = 4$ for fast growing data and selection of r within a range of $0.8 - 0.9$ will result in better average space usage for wide ranges of growth.

False positive rate (FPR) for SBF depends on the individual false positive rate of $SBF_j (0 \leq j \leq i)$. FPR of ith SBF is:

$$f_i^{SBF} = (1 - e^{-kNo/m_i})^k. \tag{9}$$

FPR for the recently added slice, where size of the slice is m_ℓ is given by:

$$f_\ell^{SBF} = (1 - e^{-kt/m_\ell})^k. \tag{10}$$

Based on Eqs. (9) and (10), compound false positive rate (F_{SBF}) of SBF is:

$$F_{SBF} = 1 - \{\prod_{j=0}^{j=i-1} (1 - f_j^{SBF})\} \times (1 - f_\ell^{SBF}). \tag{11}$$

Approach used to search the elements in SBF is similar to that in DBF, *i.e.*, by checking the presence of element in active filter followed by the previous filters in decreasing order. In worst case, when the element is present in first filter, querying an element in SBF requires a lookup of all the $(i + 1)$ SBF filters. It has been experimentally verified that computational overhead of SBF surges as the size of SBF grows.

2.3. Motivation

With the recent rise in popularity of social media, on-line transactions, recommender systems, *etc.* extraction of useful information from such massive data, particularly from analytic point of

Table 1
Overview of few existing Bloom filter based techniques.

Contributor	Filter used	Focus of the research	Performance evaluation
Quan et al. [44]	Adaptive Prefix Bloom Filter	A scalable name lookup solution, *i.e.*, Scalable Name Lookup engine with Adaptive Prefix Bloom filter (NLAPB) approach was proposed for deploying Named Data Networking (NDN) at large scale. Here, each NDN name/prefix was split into B-prefix followed by T-suffix where the length of B-prefixes were dynamically throttled based on their popularity in order to accelerate the lookup.	For experimentation, 1–10 million prefixes were selected randomly to compare the NLAPB approach with the three existing solutions, *i.e.*, Components-Trie (CT), Bloom-Hash (BH) and Hash Table (HT). The results indicated that NLAPB lowers the false positive rate, decreases the memory requirements and reduces processing time when compared to other three approaches.
Alexander et al. [45]	Dynamic Bloom Filter (DBF)	A Dynamic Interest-Tagged Filtered Bloom Filters (DITFBFs) based cache sharing system was introduced to reduce inter-proxy network overhead that occurs in cooperative web caching.	Evaluations on Bank-Search dataset (collection of Web documents) was conducted to measure latency, cache hit ratio, byte hit ratio, and number of cache miss protocol messages.
Lee et al. [46]	Trie based Bloom Filter	A hashing-based name prefix trie for name prefix matching using BF was proposed. Here, an off-chip hash table storing the nodes of the name prefix trie is accessed using BF.	Due to a hierarchical structure of URL names, performance evaluation was carried over ALEXA dataset by extracting 4 sets with sizes of 10,000, 50,000, 100,000, and 300,000 names.
Liu et al. [47]	Vector Bloom Filter (VBF)	Data streaming method for detection of superpoints (infected hosts) was proposed. Here, the overlapping of hash bit strings was evaluated to obtain the information of infected hosts.	Theoretical analysis was conducted which showed that the proposed method detects super-points more precisely and efficiently when compared to existing approaches
Mun et al. [48]	Cache management based Bloom Filter	A cache management scheme for effective content distribution and high cache utilization with neighboring routers in Named Data Networking (NDN) was proposed. Here BF was used to define a summary packet.	Evaluation of the proposed scheme was done over ndnSIM, an NS-3 based Named Data Networking simulator for evaluating the measures like content diversity, cache hit ratio, and average content delivery time.
Xiong et al. [49]	Probabilistic Bloom Filter (PBF)	Classical Bloom filter was extended into probabilistic direction for the identification of frequent traffic flows. Further, the capacity and tradeoffs of PBF was investigated to calibrate this data structure's parameters.	The proposed PBF was evaluated (in terms of computational time complexity, memory overhead and trade-off) on GPUs by considering the real network traces (dynamic data-streams) fetched from a backbone router.

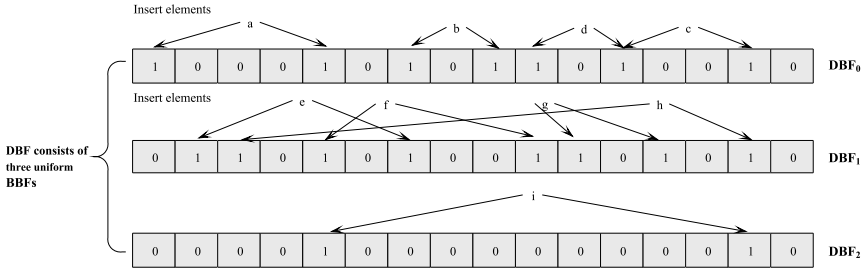

Fig. 2. Dynamic Bloom filter [23].

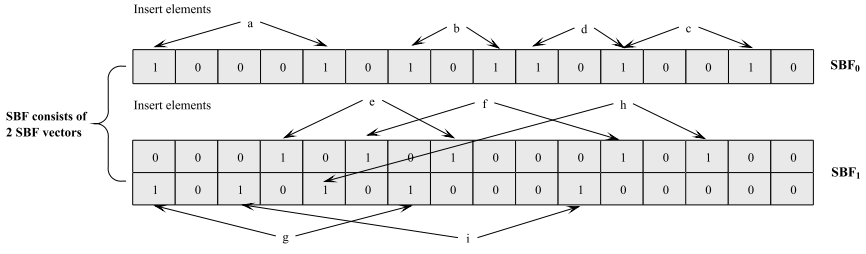

Fig. 3. Scalable Bloom filter [23].

view has become a daunting task [50]. In order to support data applications running in different domains, data storage must be efficient and automated as much as possible. To provide bulk storage with minimum memory requirements, various probabilistic alternatives can be considered, with some bearable error. Thus, this paper tries to accommodate the massive datasets using a variant of BF, a probabilistic data structure which can store bulk data using two set of hash functions, leading to efficient storage and membership query.

2.4. Contributions

This paper presents a variant of existing scalable and dynamic BF with the prime focus on the timely query of in-stream data by using partition hashing. The proposed BF named as Accommodative Bloom Filter (ABF) accommodates the incoming data by adding new filters of size 'm/b' in the limited buckets selected by initial bits of hash functions. Since there is no overhead of adding new filter of size greater then or equal to the size of original filter, as in DBF and SBF, it offers better performance and scalability with reduced query time. It has been experimentally proved that the proposed filter outperforms the existing BF when dealing with in-stream data.

3. The proposed approach: accommodative bloom filter (ABF)

ABF consists of b buckets of l bits each and corresponding to each bucket a BF of size $\hat{m} = \frac{m}{b}$ is attached. Incoming elements are hashed to p bits and two set of hash functions are considered: first q bits are used for hashing the bucket index and remaining $(r = p-q)$ bits are used for hashing the BF attached with each bucket. At first level q bits are hashed using k hash functions, which helps to determine the bucket number for a data instance. At second level, r bits data is hashed using k_1 hash functions and inserted in BF. Whenever certain BF exceeds its fill ratio, a new filter is added to accommodate incoming elements. To perform membership query in ABF, checks are performed at two levels, *i.e.*, bucket level and BF level. This two-tier checking reduces the membership query time and enhances the accuracy of the proposed approach. A detailed layout of the proposed ABF approach has been shown in Fig. 4.

3.1. Insertion in accommodative bloom filter

- Since ABF consists of an array of b buckets with each bucket having filter of size $\hat{m}$, total size of ABF (S_{ABF}) is:

$$S_{ABF} = b(l + \hat{m}). \tag{12}$$

- Every input element x_i is hashed to p bits. First q bits are used to determine bucket and values corresponding to selected buckets are set HIGH. Remaining $r = (p - q)$ bits set the value of the respective $\hat{m}$ bit array HIGH corresponding to the buckets set high by r bits. k hash functions $\overset{k}{\underset{i=1}{}}H(x^q)$ are used to map q bits from the input x to the buckets. Mathematically, the insertion process is:

$$\begin{cases} (x_i) \mapsto (p\ Bits) \mapsto \begin{cases} x_i^q & q\ Bits \\ x_i^r & r\ Bits \end{cases} \\ Hashing\ in\ Buckets \begin{cases} \overset{k}{\underset{j=1}{}}[H((x_i)^q) \mapsto Bu_j] \\ \overset{k}{\underset{j=1}{}}Bu_j = HIGH \end{cases} \\ Hashing\ in\ Bloom\ Filter \begin{cases} \overset{k}{\underset{j=1}{}}[G((x_i)^r) \mapsto g_j] \\ \overset{k}{\underset{j=1}{}}(BF^{Bu_j}[g_j] = HIGH) \end{cases} \end{cases}.$$

- Second set of hash functions, *i.e.*, $\overset{k_1}{\underset{i=1}{}}G(x^q)$ contain k_1 hash functions used to hash remaining r bits to the array associated with each bucket. Here partitioning hashing is used to update the bits corresponding to hash indexes $\overset{k_1}{\underset{i=1}{}}(g(x))$, *i.e.*, BF is divided in k_1 parts(p_i) and one hash function is used to update each part. Mathematically, the operations performed are:

$$\begin{cases} \overset{k_1}{\underset{j=1}{}}(p_j = \frac{m}{(k_1)b}) \\ Hashing\ in\ Bloom\ Filter \begin{cases} \overset{k_1}{\underset{j=1}{}}[G((x_i)^r) \mapsto g_j] \\ \overset{k_1}{\underset{j=1}{}}[g_j \mapsto p_j] \\ \overset{k}{\underset{j=1}{}}(BF^{Bu_j}[p_j] = HIGH) \end{cases} \end{cases}.$$

- As the array associated with bucket Bu_i exceeds its defined threshold, a slice m_1 is added with size same as that of SBF. This will help to accommodate more number of elements. The size of m_1 is given as:

$$m_1 = m_0 + s^x \tag{13}$$
$$s^x = \frac{\hat{m}}{k_1}.$$

- Whenever a slice is added to BF, counter of corresponding bucket is incremented by one. Incrementing the counter helps in determining the number of slices added to a particular bucket.

The insertion complexity of proposed ABF is depended on the three factors: hashing process (H_{ef}) which includes mapping of the input to f_q and f_r, number of buckets (B_n) used and number of hash functions (k) used in BF associated with each bucket along with number of slices (x) added in it. The insertion complexity is given by:

$$O\big(B_n(H_{ef} + xk)\big). \tag{14}$$

The entire insertion process has been demonstrated in Algorithm 1.

Algorithm 1 Insertion in ABF

```
procedure
    S_ABF = b(l + m̂)
    s_x = m̂ / k_1
    for x_i ∈ D do
        x_i ↦ x_i^p Binary conversion
        x_i^p ↦ x_i^q Bucket hashing Input
        x_i^p ↦ x_i^r Bloom filter hashing Input
        for (j = 1, j ≤ k, k++) do
            Bu_j ← H_j(x_i^q)
            Bu_j = HIGH
            for (c = 1, c ≤ k_1, k_1++) do
                ∀Bu_j do:
                g_c ← G_c(x_i^r)
                BF[g_c]^{Bu_j} = HIGH
                if (BF[g_c]^{Bu_j} > Threshold) then
                    Add slice of size s_x
                    Bu_j = Bu_j + 1
                end if
            end for
        end for
    end for
end procedure
```

3.2. Query in accommodative bloom filter

Membership query refers to determining whether an element y_i is part of filter or not. In the proposed storage scheme, query element y_i is checked at two levels, leading to more accurate membership query check.

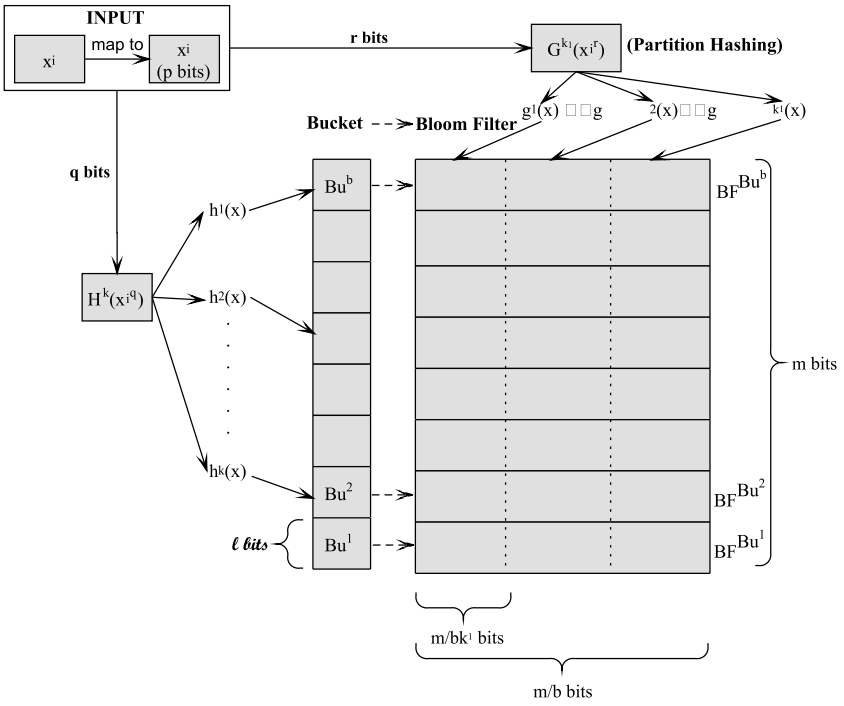

Fig. 4. A detailed layout of ABF.

- *Level I check:* Initially data is hashed using k hash function, *i.e.*, $\overset{j^k}{_{j=1}}H(y_i^q)$ to determine the bucket where the data might be residing. If all buckets corresponding to $h_j()$ are HIGH only then hash functions are calculated for next level, otherwise query is terminated with a response that data does not exist in the ABF, *i.e.*, $y_i \notin ABF$.
- *Level II check:* In second level check, (if level 1 check is true) k_1 hash functions, *i.e.*, $\overset{k_1}{_{j=1}}G(y_i^r)$ are computed for arrays corresponding to buckets set high by Level I. Here before checking in BF associated with bucket, counter of bucket is checked. If counter is $>$ 1 then searching of element starts from the latest added slice to oldest slice in reverse order. If corresponding to all buckets and all BFs associated with them, all hash positions are HIGH then only query is considered successful, otherwise query fails, *i.e.*, $y_i \notin ABF$. Mathematically,

$$\begin{cases} (y_i) \mapsto (p\ Bits) \mapsto \begin{cases} y_i^q & q\ Bits \\ y_i^r & r\ Bits \end{cases} \\ Level\ I: \\ \overset{k}{_{j=1}}[H((y_i)^q) \mapsto Bu_j]\ \{If(\forall_j(Bu_j == HIGH)) \quad \begin{cases} TRUE & Level\ II: \\ FALSE & Q = 0 \end{cases} \\ Level\ II: \\ \forall_{s_x}\{\overset{k}{_{j=1}}[G((x_i)^r) \mapsto g_j]\} \begin{cases} TRUE & Q = 1 \\ FALSE & Q = 0 \end{cases} \\ Where\ (Q = 1) \mapsto (y_i \in ABF) \\ and\ (Q = 1) \mapsto (y_i \notin ABF) \quad . \end{cases}$$

Returns TRUE if above mentioned conditions are successful.

The entire query process has been demonstrated in Algorithm 2.

Lemma 3.1. *Query complexity of ABF with b buckets using k hash functions and b BFs of size $\hat{m}$ using k_1 hash functions is given by (Q_{ABF}):*

$$Q_{ABF} = O(k) + O(k_1 x) \tag{15}$$

where x is the number of slices added in BF and initial value of $x = 1$.

Algorithm 2 Querying in ABF

```
1: procedure
2:     for y_i ∈ Q do
3:         y_i ↦ y_i^p Binary conversion
4:         y_i^p ↦ y_i^q Bucket hashing Input
5:         y_i^p ↦ y_i^r Bloom filter hashing Input
6:         for (j = 1, j ≤ k, k++) do
7:             Bu_j ← H_j(y_i^q)
8:         end for
9:         if (∀_j(Bu_j = HIGH)) then
10:            Jump Level II:
11:        else
12:            Return FALSE
13:        end if
14:        Level II:
15:        for (c = 1, c ≤ k_1, c++) do
16:            g_c ← G_c(x_i^r)
17:        end for
18:        if [(∀_{s_x})(∀_{j=1}^{j=k})(_{g_c=1}^{k_1})([BF[g_c]^{Bu_j}] = HIGH)] then
19:            Return TRUE
20:        else
21:            Return FALSE
22:        end if
23:    end for
24: end procedure
```

Proof. In DBF and SBF query for an element y_i starts with latest slot added and all slots are scanned in reverse order. For DBF and SBF with x slots, search operation is :

$$\gamma \leftarrow (\forall(Slots(1:x))(\overset{i=k}{_{i=1}}H(y) == HIGH)). \tag{16}$$

If for any slot $\gamma == TRUE$ then queried element y_i is present in BF otherwise not. The worst case query complexity of querying task is $O(xk)$.

In ABF only k buckets out of b needs to be scanned for level I checking. So query complexity of ABF, when query fails at level I is given by $O(k)$. Further, when second level check is required in ABF (assuming for all BFs x=1), query complexity is given as $O(k) + O(k_1)$. As the incoming data increases in ABF, more number of slots need to be added leading to increased query complexity.

So, query complexity for BFs with x slices added is given by:

$$Q_{ABF} = O(k) + O(k_1 x). \tag{17}$$

Hence, the two level checking provide more accurate results and average query complexity of ABF is approximately same as that of DBF and SBF.

Lemma 3.2. *Using two levels of hashing, ABF with b buckets has significantly less number of false positive (f_{ABF}) for an input dataset of N elements, as compared to DBF and SBF .*

Proof. With b buckets of l bits each and a bloom filter of size $\hat{m}$ associated with each bucket, total size of ABF is given by:

$$m = b(l + \hat{m})$$
$$\hat{m} = \hat{m} + s_x$$
$$where\ s_x = Size\ of\ slice. \tag{18}$$

In ABF buckets are represented as counting BF and all arrays are simple bloom filter. So, false positive rate at level I with b buckets, k hash functions and N number of elements accommodated, is given by:

$$f_I^{ABF} = ((1 - e^{-kN/b})^k). \tag{19}$$

At level II, BFs of b buckets of size $\hat{m}$ are checked using k_1 hash functions, where each BF can accommodate N_s elements, assuming x slices being added in each BF then false positive rate f_s^{ABF} for one slice in each bucket is:

$$f_s^{ABF} = ((1 - e^{-kN_s/m_s})^k)^b \tag{20}$$

where m_s is size of each slice and N_s is the number of elements accommodated by each slice. Hence compound false positive rate (f_{II}^{ABF}) of ABF is given as:

$$f_{II}^{ABF} = 1 - \{\prod_{s=1}^{s=i}(1 - f_s^{ABF})\}. \tag{21}$$

From Eqs. (19)-(21) compound false positives for ABF is given as

$$F_{ABF} = (1 - \{\prod_{s=1}^{s=i}(1 - f_s^{ABF})\})((1 - e^{-kN/b})^k) \tag{22}$$

which is comparatively less than the false positives of DBF and SBF.

4. Experimental evaluation

A comparative analysis is performed for SBF, DBF and ABF since all three increment the original array whenever the defined fill ratio is surpassed. All the experiments have been performed on $i7 - 3612QM$ CPU @ 2.10 GHz with 8 GB of main memory. To maintain uniformity in the results *CityHash* 64 bit library is used to calculate initial two hash functions in double hashing. Data from IPv4 Routed /24 Autonomous System (AS) Links Dataset [51] is considered as input. The data instance shown in Fig. 5 is data of June 4, 2015 which include 400,468 entries. Each data file contain three attributes $team_{id}$, IP address of source and url accessed by resource during the transmission process. All experiments are performed using *RStudio*.

To make the comparative analysis of DBF, SBF and ABF uniform size of filter and false positives are fixed to 6400 bits and 0.01 respectively. Size of DBF has linear correlation with number of elements and new filters added are of same size of previous filter after the defined threshold is crossed. In SBF, size increases in stepwise manner, *i.e.*, slices of size s_s are added to the original filter as the incoming data increases, *i.e.*, ($s_s = \frac{m}{k} = 1600$ bits). Results achieved in Table 2 indicate that increase in size of ABF is $\frac{m}{kk_1}$ for every additional filter which stores the same amount of data efficiently and saves lot of memory. For experimental purposes, ABF considers b buckets and array of size 6400 bits. Each bucket has $\frac{6400}{b}$ bits associated with it. Four hash functions are used at each level, *i.e.*, $k = 4$ and $k_1 = 4$. Thus, slice size (s) added in the BF is 400 bits.

Another important measure of efficiency of the BF is membership query time. As the number of arrays or slices increase in DBF and SBF respectively, the filter checks in the active filter first followed by the previous one and keeps on scanning the entire array till the first filter. In other words, the query time is the function of incoming data N. In case of ABF, query time is quite less as number of buckets to be searched are determined by the first set of hash function only and if Level I check is TRUE then only filters associated with the respective buckets are checked. This limited number of array checking saves lot of time in ABF. Further, addition of counters indicate before hand how many filters need to be checked. Thus, it can be concluded that when compared on the basis of membership query time for an element, ABF outperforms both SBF and DBF.

Third important measure is false positive (fp) rate. As the size of the filter increases, in DBF fp increases manifold but in SBF increase in fp is limited through the error probability tightening ratio ρ. In ABF increase in fp is approximately same as that of SBF, indicating that increase in the data does not increase the false positives.

The results for the same are demonstrated in Figs. 6–9 where Fig. 6 shows the comparative analysis of query time with the increase in number of hash functions in SBF, DBF and ABF. It is clear from Fig. 6 that DBF shows significant increase in query time with the increase in hash functions. This happens because as the number of hash functions will increase, threshold will be crossed more frequently and hence more filters will be added leading to increased query time. In SBF, with increase in hash functions only the number of slices added will increase and hence query time will increase slowly, and in ABF this will be further decrease as the hash functions have been divided into two sets k and k_1.

Fig. 7 shows the effect on query time with the increase in number of elements (N). As the number of elements increase query time increases drastically in DBF but it is almost constant in SBF and ABF since both add only limited size slices with increase in number of elements.

Fig. 8 depicts false positive rate for DBF, SBF and ABF with respect to number of elements (N). It is clear from Fig. 8 that in DBF, fp increases as the number of elements increase while in SBF and ABF, false positive rates does not grow at the same rate as the number of elements increase.

Fig. 9 depicts variation in false positive (fp) rate with the change in hash functions at both levels in proposed approach. Variation in level II has been shown for k_i by keeping $k = 4$. It is evident from Fig. 9 shows that as the number of hash functions increase, fp rate decrease. With the increase in hash functions, number of collisions will definitely decrease and hence, fp decreases.

Thus, the results achieved from all the parameters clearly indicate that ABF outperforms significantly in terms of size, membership query time and false positive rate when compared with DBF and better in comparison to SBF as it can accommodate streaming data in small space with less query time and very less increase in false positive rate.

5. Conclusion

This paper proposes Accommodative Bloom Filter (ABF) along with algorithms for insertion and query of ever-increasing data coming from sensor devices. Theoretical and experimental results

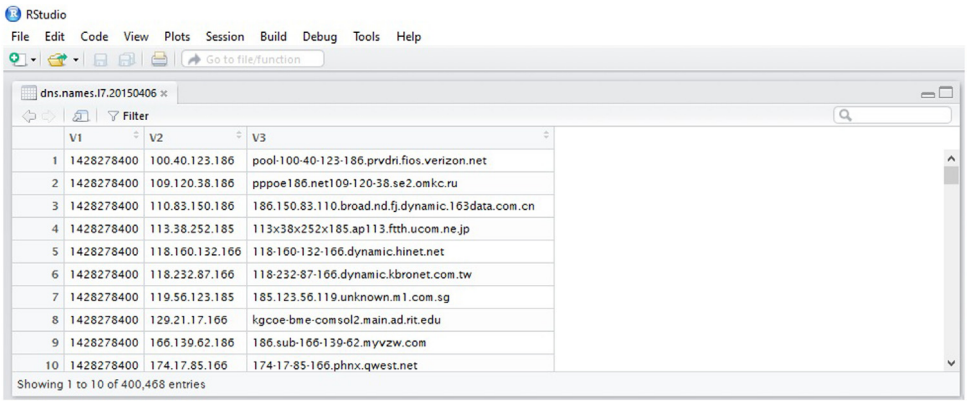

Fig. 5. An instance of dataset of IPv4 routed /24 topology dataset [51].

Table 2
Comparative analysis.

Types and parameters	Dynamic bloom filter	Scalable bloom filter	Accommodative bloom filter
Size of bloom filter	$r \times m = (\lfloor (N/N_0) \rfloor + 1) \times m$	$m \times (2^{\lfloor \log(N/N_0+1) \rfloor + 1} - 1)$	$b(\hat{a} + l)$
Query complexity	$O(k \times N)$	$O(k \times logN)$	$O(k) + O(k_1 x)$
False positive rate	$1 - \{(1 - (1 - e^{-kN_0/m})^k)^{(r-1)} \times (1 - (1 - e^{-kt/m})^k)\}$	$1 - \{\prod_{j=0}^{j=i-1}(1 - (1 - e^{-kNo/m_j})^k)\} \times (1 - (1 - e^{-kt/m_\ell})^k)$	$F_{ABF} = (1 - \{\prod_{s=1}^{s=i}(1 - f_s^{ABF})\})((1 - e^{-kN/b})^k)$

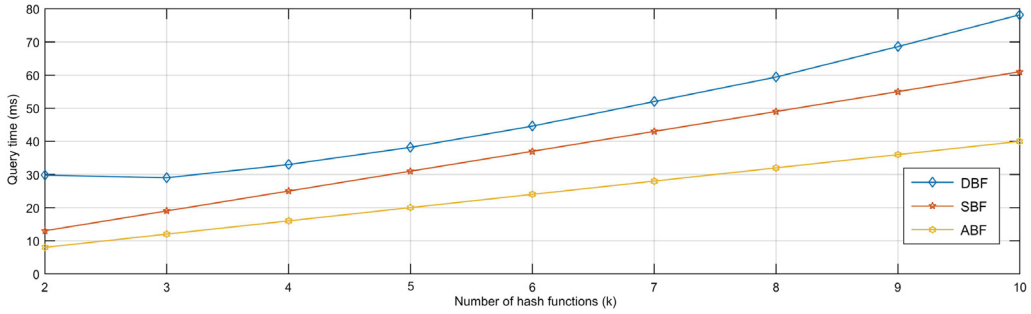

Fig. 6. Query time (in ms) vs. Number of hash functions (k).

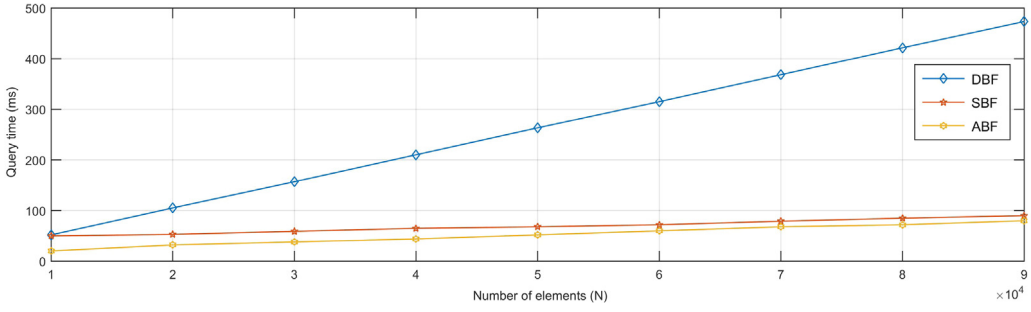

Fig. 7. Query time (in ms) vs. Number of elements (N).

demonstrate that ABF proposed in this paper offers better performance and scalability with less storage requirement and less query time complexity in comparison to dynamic and scalable BF while dealing with in-streaming data. Presently the experimentation has been performed on real time in-stream data received from sensors for a limited time span and in future it will be extended to heterogeneous datasets coming from various data sources.

6. Discussion

In distributed databases such as Casandra, every cluster node has an in-memory bloom filter which helps in determining whether the given hash values are present in the database or not [52]. Use of BF decreases latency in such scenarios. In Hadoop, when the input is received from varied sources then Mapper does

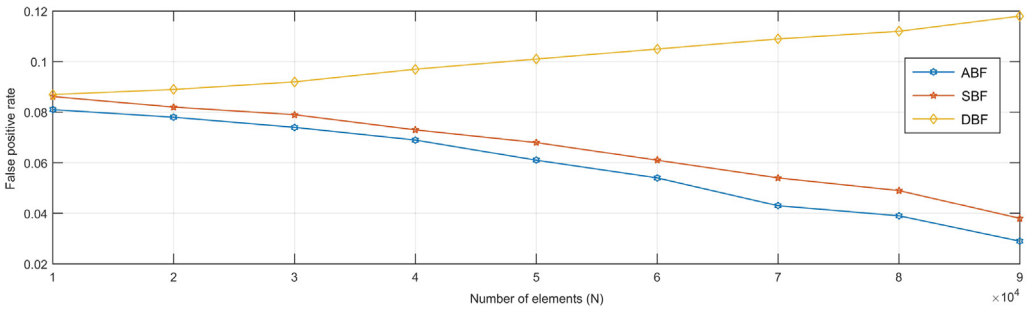

Fig. 8. False positive rate vs. Number of elements (N).

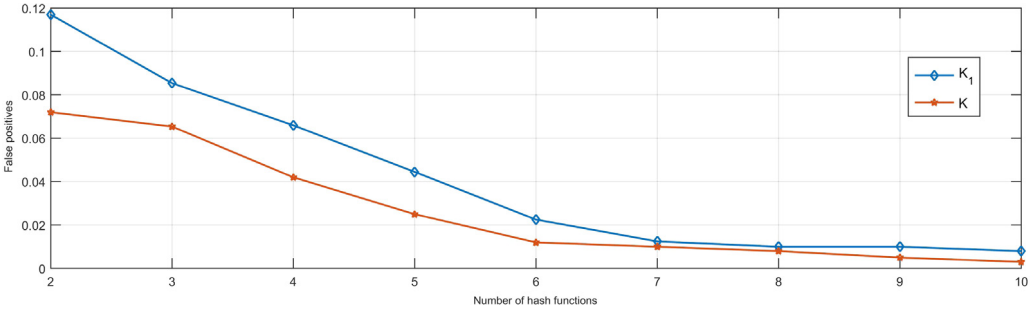

Fig. 9. False positive rate vs. Number of hash functions (k_1).

not pass the inputs directly to the reducer, it generates its bloom values and then passes it to Reducer class. This helps in reducing its computational efforts. Hadoop has a Bloom Filter class in its util package [53]. Squid, distributed web proxy caches frequently access the web content using BF which provides users a faster web experience [54]. Apache HBase uses BF to boost read speed by filtering out unnecessary disk reads of HFile blocks which do not contain a particular row [55]. BF finds its application in Quora which uses a shared in the feed back-end to filter out stories that people have seen before. Facebook performs various applications through BF which include: type-ahead search, fetching friends and friends of friends, a user typed query, etc. Bloom pruning of partitions is done using BF in Oracle [56]. From the above cited application domains, it is clear that BF can be efficiently used for fast and efficient membership query with drastic storage space reduction. Thus, the proposed ABF scheme can be easily integrated with existing big-data storage solutions of stream-data processing engines such as-Hadoop, Storm, Samza, Spark, Flink, etc.

Acknowledgment

Two of the authors (Amritpal Singh and Sahil Garg) would like to acknowledge the financial support given to them by Department of Electronics and Information Technology (DeitY), Ministry of Communications and IT under Government of India to complete their Doctoral studies.

References

[1] A. L'Heureux, K. Grolinger, H.F. ElYamany, M. Capretz, Machine learning with big data: Challenges and approaches, IEEE Access (99) (2017) 1–1.
[2] J. Merino, I. Caballero, B. Rivas, M. Serrano, M. Piattini, A data quality in use model for big data, Future Gener. Comput. Syst. 63 (Supplement C) (2016) 123–130 Modeling and Management for Big Data Analytics and Visualization.
[3] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, M. Ayyash, Internet of things: A survey on enabling technologies, protocols, and applications, IEEE Commun. Surv. Tutor. 17 (4) (2015) 2347–2376.
[4] K. Kaur, T. Dhand, N. Kumar, S. Zeadally, Container-as-a-service at the edge: Trade-off between energy efficiency and service availability at fog nano data centers, IEEE Wirel. Commun. 24 (3) (2017) 48–56.
[5] S. Garg, S. Batra, A novel ensembled technique for anomaly detection, Int. J. Commun. Syst. (2016). http://dx.doi.org/10.1002/dac.3248.
[6] S. Garg, S. Batra, Fuzzified cuckoo based clustering technique for network anomaly detection, Comput. Electr. Eng. (2017). http://dx.doi.org/10.1016/j.compeleceng.2017.07.008.
[7] C.-W. Tsai, C.-F. Lai, M.-C. Chiang, L.T. Yang, et al., Data mining for internet of things: A survey, IEEE Commun. Surv. Tutor. 16 (1) (2014) 77–97.
[8] M. Chen, S. Mao, Y. Zhang, V.C. Leung, Big data applications, in: Big Data, Springer, 2014, pp. 59–79.
[9] Y. Demchenko, C. Ngo, P. Grosso, C. de Laat, P. Membrey, Cloud based infrastructure for data intensive e-science applications: Requirements and architecture, in: Cloud Computing with e-Science Applications, CRC Press, 2015, pp. 17–40.
[10] P. Russom, et al., Big data analytics, TDWI best practices report, fourth quarter, Vol. 19, 2011, p. 40.
[11] D. Gil, I.-Y. Song, Modeling and management of big data: Challenges and opportunities, Future Gener. Comput. Syst. 63 (Supplement C) (2016) 96–99. Modeling and Management for Big Data Analytics and Visualization.
[12] K. Krishnan, Data Warehousing in the Age of Big Data, Newnes, 2013.
[13] V. Mayer-Schönberger, K. Cukier, Big Data: A Revolution That Will Transform How We Live, Work, and Think, Houghton Mifflin Harcourt, 2013.
[14] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami, Internet of Things (IoT): A vision, architectural elements, and future directions, Future Gener. Comput. Syst. 29 (7) (2013) 1645–1660.
[15] X. Jin, B.W. Wah, X. Cheng, Y. Wang, Significance and challenges of big data research, Big Data Res. 2 (2) (2015) 59–64.
[16] J.J. Jung, Computational Collective Intelligence with Big Data: Challenges and Opportunities, Elsevier, 2017.
[17] J. Wang, X. Chen, J. Li, J. Zhao, J. Shen, Towards achieving flexible and verifiable search for outsourced database in cloud computing, Future Gener. Comput. Syst. 67 (Supplement C) (2017) 266–275.
[18] S. Geravand, M. Ahmadi, Bloom filter applications in network security: A state-of-the-art survey, Comput. Netw. 57 (18) (2013) 4047–4064.
[19] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (7) (1970) 422–426.
[20] O. Rottenstreich, I. Keslassy, The bloom paradox: When not to use a bloom filter, IEEE/ACM Trans. Netw. 23 (3) (2015) 703–716.
[21] D. Guo, J. Wu, H. Chen, Y. Yuan, X. Luo, The dynamic bloom filters, IEEE Trans. Knowl. Data Eng. 22 (1) (2010) 120–133.
[22] P.S. Almeida, C. Baquero, N. Preguiça, D. Hutchison, Scalable bloom filters, Inform. Process. Lett. 101 (6) (2007) 255–261.
[23] K. Xie, Y. Min, D. Zhang, J. Wen, G. Xie, A scalable bloom filter for membership queries, in: Global Telecommunications Conference, (GLOBECOM'07), IEEE, 2007, pp. 543–547.

[24] S.F. Ochoa, G. Fortino, G.D. Fatta, Cyber-physical systems, internet of things and big data, Future Gener. Comput. Syst. 75 (Supplement C) (2017) 82–84.
[25] M. Babar, F. Arif, Smart urban planning using Big Data analytics to contend with the interoperability in Internet of Things, Future Gener. Comput. Syst. 77 (Supplement C) (2017) 65–76.
[26] H. Cai, B. Xu, L. Jiang, A.V. Vasilakos, IoT-based big data storage systems in cloud computing: Perspectives and challenges, IEEE Internet Things J. 4 (1) (2017) 75–87.
[27] L. Zhou, On data-driven delay estimation for media cloud, IEEE Trans. Multimedia 18 (5) (2016) 905–915.
[28] L. Zhou, QoE-driven delay announcement for cloud mobile media, IEEE Trans. Circuits Syst. Video Technol. 27 (1) (2017) 84–94.
[29] M.A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B.C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R.P. Spillane, E. Zadok, Don't thrash: how to cache your hash on flash, Proc. VLDB Endow. 5 (11) (2012) 1627–1637.
[30] G. Cormode, S. Muthukrishnan, An improved data stream summary: the count-min sketch and its applications, J. Algorithms 55 (1) (2005) 58–75.
[31] P. Flajolet, É. Fusy, O. Gandouet, F. Meunier, Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm, in: AofA: Analysis of Algorithms, in: Discrete Mathematics and Theoretical Computer Science, 2007, pp. 137–156. URL: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=296BD6CC49EE6536B2888CD260F57BAD?doi=10.1.1.142.9475&rep=rep1&type=pdf.
[32] H. Song, F. Hao, M. Kodialam, T. Lakshman, Ipv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards, in: INFOCOM, IEEE, 2009, pp. 2518–2526.
[33] B. Donnet, B. Baynat, T. Friedman, Retouched bloom filters: Allowing networked applications to flexibly trade off selected false positives against false negatives, 2017.
[34] F. Nikitin, Bloom filters and their applications, 2006.
[35] S. Tarkoma, C.E. Rothenberg, E. Lagerspetz, et al., Theory and practice of bloom filters for distributed systems, IEEE Commun. Surv. Tutor. 14 (1) (2012) 131–155.
[36] S.K. Pal, P. Sardana, K. Yadav, Efficient multilingual keyword search using bloom filter for cloud computing applications, in: Fourth International Conference on Advanced Computing, (ICoAC), IEEE, 2012, pp. 1–7.
[37] T. hoon Kim, C. Ramos, S. Mohammed, Smart city and IoT, Future Gener. Comput. Syst. 76 (Supplement C) (2017) 159–162.
[38] A. Kalmar, R. Vida, M. Maliosz, CAEsAR: A context-aware addressing and routing scheme for RPL networks, in: 2015 IEEE International Conference on Communications, ICC, 2015, pp. 635–641. http://dx.doi.org/10.1109/ICC.2015.7248393.
[39] A. Alrawais, A. Alhothaily, C. Hu, X. Cheng, Fog computing for the Internet of Things: Security and privacy issues, IEEE Internet Comput. 21 (2) (2017) 34–42.
[40] C. Muñoz, P. Leone, A distributed event-based system based on compressed fragmented-iterated bloom filters, Future Gener. Comput. Syst. (2017). http://dx.doi.org/10.1016/j.future.2017.02.021.
[41] D. Zhang, Z. He, Y. Qian, J. Wan, D. Li, S. Zhao, Revisiting unknown RFID tag identification in large-scale internet of things, IEEE Wirel. Commun. 23 (5) (2016) 24–29.
[42] M. Amoretti, O. Alphand, G. Ferrari, F. Rousseau, A. Duda, DINAS: A lightweight and efficient distributed naming service for All-IP wireless sensor networks, IEEE Internet Things J. 4 (3) (2017) 670–684.
[43] C. Li, Z. Qin, E. Novak, Q. Li, Securing SDN infrastructure of IoT-fog network from MitM attacks, IEEE Internet Things J. (99) (2017). http://dx.doi.org/10.1109/JIOT.2017.2685596. 1–1.
[44] W. Quan, C. Xu, J. Guan, H. Zhang, L.A. Grieco, Scalable name lookup with adaptive prefix bloom filter for named data networking, IEEE Commun. Lett. 18 (1) (2014) 102–105.
[45] H. Alexander, I. Khalil, C. Cameron, Z. Tari, A. Zomaya, Cooperative web caching using dynamic interest-tagged filtered bloom filters, IEEE Trans. Parallel Distrib. Syst. 26 (11) (2015) 2956–2969.
[46] J. Lee, M. Shim, H. Lim, Name prefix matching using bloom filter pre-searching for content centric network, J. Netw. Comput. Appl. 65 (2016) 36–47.
[47] W. Liu, W. Qu, J. Gong, K. Li, Detection of superpoints using a vector bloom filter, IEEE Trans. Inf. Forensics Secur. 11 (3) (2016) 514–527.
[48] J.H. Mun, H. Lim, Cache sharing using bloom filters in named data networking, J. Netw. Comput. Appl. (2017).
[49] Frequent traffic flow identification through probabilistic bloom filter and its GPU-based acceleration, J. Netw. Comput. Appl. 87 (2017) 60–72.
[50] D. Puthal, S. Nepal, R. Ranjan, J. Chen, A dynamic prime number based efficient security mechanism for big sensing data streams, J. Comput. System Sci. 83 (1) (2017) 22–42.
[51] Ipv4 routed/24 topology dataset, 2015. URL: http://www.caida.org/data/active/ipv4routed24topologydataset.xml [Accessed on: Aug. 2017].
[52] Apache casandra, 2016. URL: http://cassandra.apache.org/doc/latest/operating/bloom_filters.html [Accessed on: Aug 2017].
[53] M. Valenty, Hadoop mapreduce join optimization with a bloom filter, 2013. URL: http://www.mikevalenty.com/hadoop-mapreduce-join-optimization-with-a-bloom-filter/ [Accessed: Aug 2017].
[54] Squid-cache wiki, 2008. URL: https://wiki.squid-cache.org/SquidFaq/CacheDigests[Accessed on: Aug 2017].
[55] Interface bloomfilter, 2017. URL: https://hbase.apache.org/devapidocs/org/apache/hadoop/hbase/util/BloomFilter.html [Accessed on: Sept 2017].
[56] How do giant sites like facebook and google check username or domain availability so fast?, 2017. URL: http://blog.hackerearth.com/how-websites-check-username-quickly [Accessed on: Sept 2017].

Amritpal Singh received M.E. Degree from Thapar University, Punjab, India, with a minor in big data and advanced data structures, in 2013. He is working as Research Scholar with Computer Science Department at Thapar University, Punjab, India since Jan 2015. He served both industry and academia. His research interest includes Probabilistic Data Structures, Machine Learning and Big data.

Sahil Garg received the B.Tech Degree from Maharishi Markandeshwar University, Mullana, Ambala, India, in 2012, and M.Tech degree from Punjab Technical University, Jalandhar, India, in 2014, both in computer science and engineering. He is currently working towards the Ph.D. degree in computer science and engineering from Thapar University, Patiala, India. His research interests include machine learning, big data analytics, knowledge discovery, game theory and vehicular ad-hoc networks.

Shalini Batra received the Ph.D. Degree in computer science and engineering from Thapar University, Patiala, India, in 2012. She is currently working as an Associate Professor with the Department of Computer Science and Engineering, Thapar University, Patiala, India. She has guided many research scholars leading to Ph.D. and M.E./M.Tech. She has authored more than 60 research papers published in various conferences and journals. He research interests include machine learning, web semantics, big data analytics and vehicular ad-hoc networks.

Neeraj Kumar received his Ph.D. in CSE from Shri Mata Vaishno Devi University, Katra (J & K), India. He was a postdoctoral research fellow in Coventry University, Coventry, UK. He is currently an Associate Professor in the Department of Computer Science and Engineering, Thapar Institute of Engineering and Technology (Deemed University), Patiala (Pb.), India. He has published more than 150 technical research papers in leading journals and conferences from IEEE, Elsevier, Springer, John Wiley etc. Some of his research findings are published in top cited journals such as IEEE TIE, IEEE TDSC, IEEE TITS, IEEE TCE, IEEE Netw., IEEE Comm., IEEE WC, IEEE IoTJ, IEEE SJ, FGCS, JNCA, and ComCom. He has guided many research scholars leading to Ph.D. and M.E./M.Tech. His research is supported by fundings from Tata Consultancy Service and Department of Science & Technology.

Joel J.P.C. Rodrigues received the Habilitation in computer science and engineering from the University of Haute Alsace, France, the Academic Title of Aggregated Professor and a Ph.D. degree in informatics engineering and an M.Sc. degree from the UBI, and a five-year B.Sc. degree (licentiate) in informatics engineering from the University of Coimbra, Portugal. He is a professor and senior researcher at the National Institute of Telecommunications (Inatel), Brazil and senior researcher at the Instituto de Telecomunicações, Portugal. Prof. Rodrigues is the leader of the Internet of Things research group (CNPq), Member of the IEEE ComSoc Board of Governors as Director for Conference Development, and IEEE ComSoc Distinguished Lecturer.