Computing and Informatics, Vol. 37, 2018, 693–716, doi: 10.4149/cai_2018_3_693

STREAMED DATA ANALYSIS USING ADAPTABLE BLOOM FILTER

Amritpal Singh, Shalini Batra

Department of Computer Science and Engineering
Thapar university
Patiala, Punjab, India
e-mail: `amritpal.singh203@gmail.com, sbatra@thapar.edu`

Abstract. With the coming up of plethora of web applications and technologies like sensors, IoT, cloud computing, etc., the data generation resources have increased exponentially. Stream processing requires real time analytics of data in motion and that too in a single pass. This paper proposes a framework for hourly analysis of streamed data using Bloom filter, a probabilistic data structure where hashing is done by using a combination of double hashing and partition hashing; leading to less inter-hash function collision and decreased computational overhead. When size of incoming data is not known, use of Static Bloom filter leads to high collision rate if data flow is too much, and wastage of storage space if data is less. In such cases it is difficult to determine the optimal Bloom filter parameters (m, k) in advance, thus a target threshold for false positives (f_p) cannot be guaranteed. To accommodate the growing data size, one of the major requirements in Bloom filter is that filter size m should grow dynamically. For predicting the array size of Bloom filter Kalman filter has been used. It has been experimentally proved that proposed Adaptable Bloom Filter (ATBF) efficiently performs peak hour analysis, server utilization and reduces the time and space required for querying dynamic datasets.

Keywords: Bloom filter, partition hashing, double hashing, Kalmann filter

1 INTRODUCTION

In today's world, data is considered as one of the most valuable assets. It has been acknowledged by data scientists that timely and accurate analysis of available data helps in creating more opportunities by taking right decision at right time in ever changing business environments [1]. With the coming up of plethora of web applications and technologies like IoT [2], cloud computing [3], etc., the data generation resources are increasing exponentially. This change is leading to a shift in paradigm from existing relational data base based systems to systems which can efficiently accommodate Big data. Initially Big data [4] refereed to the collection of huge amount of unstructured data (volume and variety) only. But, with the rise in continuous data generation resources like traffic data, climate data, stock market data, etc., term velocity was introduced in Big data. Streamed data arriving from various resources requires fast processing and storage framework for handling huge amount of data. Storing entire data requires lot of memory and usage of fixed size data structures will require a lot of time for analysis [5]. Further, in case of streams, continuous analysis of data is required before storing it [6, 7].

Streaming data analytics [8] focuses on reducing time and space complexity of incoming data before storing it on disk. The important issue in stream processing is that data diminishes with time so data must be processed in a particular time window in single pass.

In many applications, fast and real time processing is required to make timely decisions accurately [9] for example in finance sector, the analysis of stock market streaming data is an essential tool for predicting stock price of the companies and real time fraud detection in a short time span. In dynamic recommender applications, processing of streamed data is necessary for referral of products according to interest of user and promotion of new products in the market [10]. In network applications, managing data streams for system monitoring can be time-varying, volatile and unpredictable since tasks to be managed include accessing the server's utilizations in particular time frame, tracking the number of unique visitors on a network in a particular time, identifying common users between two time slots, or calculating maximum number of hits on network in particular span of time. Results of network analysis can help to predict the resource usage over network, identify rush hours in network, management of network resources on the time slot basis and detect attacks like DoS and DDoS.

Fast matching of arbitrary identifiers to the values of incoming data and real time response are the basic requirements for majority of streaming data applications. Given millions or even billions of data elements, developing efficient solutions for storing, updating, and querying them becomes increasingly important especially when data is available for a short span.

Using traditional data base approaches which include performing filtering and analysis after storing the data is not efficient for the real time processing of streamed data. Since the size of incoming data is unpredictable, data structures used for the storage of data should be dynamically adjustable, but changing size in each iteration may lead to the extra computational overhead. Thus, some adaptive storage mechanism is required which performs predictive analysis to determine size of data structures being used. Provisions should also be available for adjustments on the basis of previous incoming data or on the basis of real time data flow.

Above mentioned issues clearly indicate that efficient storage and searching techniques are required for processing streaming data. Various solutions proposed by researchers in this domain utilize probabilistic techniques to reduce information processing and analytics cost. This paper proposes the use of Bloom filter [11], a probabilistic data structure [12] which can store the elements of a set in a space-efficient manner by using hashing principles with a small error in querying process. Presently Bloom filter is widely used in many networking and security algorithms like authentication, tracebacking, IP tracebacking, string matching, reply protection, etc. It is also used in fields as diverse as accounting, monitoring, load balancing, policy enforcement, routing, clustering, security and even in many database applications [13]. There are number of variants of Bloom filter which have been successfully used in different application domain [14].

The prime focus of the proposed framework is to efficiently query the incoming data within the limited time domain. To deal with instream data and store the information for a short time the proposed framework uses a Scalable Bloom filter [15] with Ageing Bloom filter properties, i.e. evicting data after fixed time interval. In the proposed framework Kalman filter [16] is used to make scalable Bloom filters adaptive in terms of size and reduce the computational overhead of adding extra filters at run time. Further, query complexity of dynamic data has also been reduced. The proposed Bloom filter is named as Adaptable Bloom Filter (ATBF) and it has been experimentally proved that the proposed filter outperforms the existing Scalable Bloom filter when dealing with in-stream data.

The plan of this paper is as follows: Section 2 provides the literature survey of the Bloom filter and its variants. In Section 3, proposed approach is discussed in detail. Section 4 provides experimental results and compares existing approach with the proposed approach. Finally, Section 5 concludes the paper.

2 RELATED WORK

2.1 Standard Bloom Filter

The Bloom Filter (BF) [11], a space efficient probabilistic data structure, is used to represent a set S of n elements. It consists of an array of m bits, denoted by $BF[1, 2, \ldots, m]$, initially all set to 0. To describe the elements in the set, the filter uses k independent hash functions $h_1, h_2, \ldots, h_k$ with their value ranging between 1 to m assuming that these hash functions independently map each element in the universe to a random number uniformly over the range. For each element $x \in S$; the bits $BF[h_i(x)]$ are set to 1 for $1 < i < k$. Given an item y, its membership is checked by examining the BF whether the bits at positions $h_1(y)$; $h_2(y)$; $\ldots$; $h_k(y)$ are set to 1. If all $h_i(y)$ $(1 < i < k)$ are set to 1, then y is considered to be part of S. If not, then y is definitely not a member of S. The accuracy of a Bloom filter depends on the filter size m, the number of hash functions k, and the number of elements n.

User can predefine false positives (f_p) according to application's requirement.

$$f_p = \left(1 - e^{-kn/m}\right)^k . \tag{1}$$

2.2 Scalable Bloom Filter

In some applications like in-stream data coming from sensors, network traffic, etc., the data is generated dynamically and size of the data set being generated cannot be determined a priori. When size of incoming data is not known, use of static Bloom filter will either lead to high collision rate or result in wastage of storage space. In such cases it is difficult to determine the optimal Bloom filter parameters (m, k) in advance, so a target false positives threshold cannot be guaranteed. In order to accommodate the growing data size, one of the major requirements in Bloom filter is that filter size m should grow dynamically.

Dynamic and scalable Bloom filters deal with the scalability problem by adding bit arrays of varying sizes as the incoming data increases. In Dynamic Bloom Filter (DBF), 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, once the threshold or the fill capacity of the existing DBF exceeds. But this addition in DBF causes the significant increase in error rate. In scalable Bloom filter, a variable size array is added whenever the defined threshold is crossed, with an extra parameter ρ to maintain the error rate in defined bound.

Scalable Bloom Filter (SBF) [15, 17] is a BF variant that can adapt dynamically to the number of incoming elements, with an assured maximum false positives f_p^0. In addition to the initial array of size m_0, SBF includes two additional parameters: expected growth rate (s) and the error probability tightening ratio (ρ) ($0 < \rho < 1$); insert operation in SBF for an element x using k hash functions and parameters s and ρ is given by $Insert(SBF[.], x, _{i=1}^{k} h_i(x), f_p^0, s, \rho)$. When $m_0 = \log_2(f_p^0)^{-1}$ exceeds the defined threshold, a new array $m_1 = m_0 + \log_2 \rho^{-1}$ is added and error probability for new filter $f_p^1 = f_p^0 \rho$. Size of additional i^{th} array m_i is:

$$m_i = \log_2(f_p^i)^{-1} = m_0 + i \times \log_2 \rho^{-1}. \tag{2}$$

For flexible growth in SBF size, exponential growth factor s is added, generating i individual filters of size $m_0, m_0 s, m_0 s^2, \ldots, m_0 s^{i-1}$. When the fill ratio t_h for one filter exceeds the defined threshold, another filter is added to it with a well defined growth parameter s. Elements stored in i^{th} filter are approximately:

$$N_i \approx m_0 s^i (\ln(t_h)). \tag{3}$$

At a given time, error probabilities of all i individual filters ($0 < i < (i-1)$) is $f_p^0, f_p^0\rho, f_p^0\rho^2, \ldots, f_p^0\rho^{i-1}$. The compounded error probability for the SBF is:

$$f_p^{SBF} = 1 - \prod_{x=1}^{i} \left(1 - f_p^0 \rho^{x-1}\right). \tag{4}$$

Query process in SBF is accomplished by testing the presence of query element in each filter, starting from active filter to oldest filter. At the time of query, if N be the total incoming elements and N_0 be the elements in m_0, total number of arrays added in Bloom filter. Search complexity for the worst case analysis is:

$$O(k(\lfloor \log(N/N_0 + 1)\rfloor) + 1). \tag{5}$$

It has been experimentally verified that computational overhead of SBF surges as the size of SBF grows with the increase in incoming data set.

2.3 Ageing Bloom Filter

Some network applications require high-speed processing of packets. For this purpose, Bloom filters array should reside in a fast and small memory. In such cases, due to the limited memory size, the stale data in the Bloom filter should be deleted to make space for the new data.

To accommodate such type of issues, number of solutions are proposed by domain experts, one of these is using only one buffer [18], i.e. allocating a buffer for insertion of elements coming from a particular network stream. For each new element, the buffer can be checked, and the element may be identified as distinct if it is not found in the buffer, and duplicate otherwise. When the buffer reaches its fill ratio, whole data is evicted from the buffer, i.e. buffer is reset to original value. Search time complexity and false positive rate in this case is determined for a particular interval, same as that of the Bloom filter. Another solution proposed for aging scheme using similar concept is double buffering [19]. In this approach, concept of buffering is used but with two filters. Initially data is filled in the first filter and once the threshold exceeded, data is filled in the next filter but as soon as the threshold of the second Bloom filter is crossed, data is evicted from the first filter and this process continues. Advantage of this approach is that we can store data for more time by using double memory than by simple buffering approach. Example of aging Bloom filter includes techniques like A^2 buffering where one buffer is divided into two parts and then double buffering is performed. One of the short comings of this approach is that size of the filter used is static, and rough prediction of size of the filter required may affect the accuracy of membership query.

2.4 Partition Hashing

Partitioning hashing is a technique where small portion of large table is uniquely allocated to each hash function such that hash key l_i generated by hash function ℓ_i, is randomly distributed over a small part of the array, i.e., each hash function is allocated to a sub part of an array [20]. In Bloom filter, an array of m bits is partitioned into k disjoint arrays of size $\vartheta = \frac{m}{k}$ bits and k hash functions are used corresponding to each part. For an element $u_i \in U$, hash functions are calculated as:

$$\kappa_i(u_i) = \ell_1(u_i) + i \times \ell_2(u_i) \bmod \vartheta. \tag{6}$$

Each hash function $\kappa_i(.)$ changes bit in i^{th} array where $i|1 < i < k$. Two independent hash functions $\ell_1(x)$ and $\ell_2(x)$ are used to generate k hash functions such that $\forall i|i < k$. To get best results of this schema, ϑ should be prime; so size of array and number of buckets should be choosen in such a way that $\frac{m}{k}$ returns a prime number. This technique leads to less inter-hash function collision, further usage of only two hash functions to generate all k hash functions decreases the computational overhead [20].

2.5 Approximate Counting in Streaming Data

Besides the batch processing infrastructure of map/reduce, Big data analytics require techniques where streamed data is processed in near real time in single pass for some specific applications.

Approximate counting problem and solution for large data sets was defined in 1981 by J. S. Moore in the Journal of Algorithms [21], and later many solutions were proposed using approximate counting in massive data sets. Two solutions are considered: Counter-based algorithms which include Frequent majority [22], LossyCounting [23], SpaceSaving [24] and Sketch based algorithms like Count-Min Sketch [25], Count Sketch [26], etc.

Manku et al. [23] proposed lossy counting algorithm, which divides large data into B_i buckets and calculates the frequency of different type of elements. Count is maintained in bucket counters C_{Bi} for only those elements which cross a defined threshold. For adding a new bucket B_n, counter of previous bucket, i.e. $C_{B_{n-1}}$, is used as base. Random decrement of all counters on the extreme sides is done after the calculations are performed on each bucket.

To answer frequency queries and reduce computational complexity, a sketch data structure named CountMin Sketch (CMS) was proposed by Muthukrishnan and Cormode in 2003 and later improved in 2005 [25]. This data structure is based on probabilistic techniques which are used to answer various types of queries on streaming data. It is a histogram which stores elements and their associated counts. Major difference between Bloom filter and CMS is that Bloom filter effectively represents sets, whereas the CMS considers multisets instead of storing a single bit to answer a query, the count min sketch maintains a count of all object. It is called a 'sketch' because it is a smaller summarization of a larger data set. The probabilistic component of CMS provides more accurate results compared to proposed sketching algorithm solutions as it has less space complexity and decreased computational cost.

2.6 Kalman Filter

Kalman filter (KF) is a linear system model derived from stochastic process, making it ideal for systems which are continuously changing. In KF recursive approach is used where a common model is formulated and all future calculations are performed on the same equations without any modification. It is easy to implement and requires less memory since it does not keep record of old data except the previous state. Further, less computational cost makes it suitable for real time problems.

Notaions	Description
$\hat{x}_k$	Posteriori state estimate
$\hat{x}_k^-$	Priori state estimate
$\hat{P}_k^-$	Priori estimated error
$\hat{P}_k$	Posteriori state estimate
K	Kalman gain
v_k	Measurement noise
R	Co-variance for measurement noise
w_k	Process noise
u_k	Control signal
Q	Co-variance for process noise
z_k	Measured value
A, B, H	Constants according to process

Table 1. Nomenclature for Kalman filter

Kalman filter is a powerfull mathematical tool mainly used for stochastic estimation from noisy sensor data or data streams occurring at regular intervals. The basic assumption of Kalman filter is that system should be continuous and can be modeled as a normally distributed random process X, with mean μ and variance σ (the error covariance), i.e. $X \sim N(\mu, \sigma)$ [16]. Kalman filter addresses the problem of estimation of state x_k of a discrete-time controlled process on the basic of previous state x_{k-1} using following equation:

$$x_k = A.x_{k-1} + B.u_k + w_{k-1} \tag{7}$$

with a measured value z_k for k^{th} state given by:

$$z_k = H.x_k + v_k \tag{8}$$

where $Pr(w) \sim N(0, Q)$ and $Pr(v) \sim N(0, R)$.

2.6.1 Discrete Kalman Filter

Kalman filter is a set of mathematical equations that build a predictor-corrector type estimator model to optimally minimize the estimated error covariance. It provides an estimate of a process for k^{th} state by using a feedback control model. In this, filter first estimates the value for k^{th} state based on the current information of the process and then obtains feedback from some measured value, i.e. noisy input. Based on the error in estimated value, Kalman gain is calculated which helps in minimizing error in further iterations. The algorithm converges to the near optimal result after few iterations.

Kalman filter is divided in two groups: time update equations and measurement update equations. The time update equations help in projecting the priori current state value ($\hat{x}_k^-$) and priori error covariance estimates ($\hat{P}_k^-$) for the next step. The time update equations act as predictor equations for estimation model [27].

$$\hat{x}_k^- = A.\hat{x}_k + B.u_k, \tag{9}$$

$$\hat{P}_k^- = A.\hat{P}_k.A^T + Q. \tag{10}$$

The measurement update equations provide feedback to the time update equations for incorporating new measurement in priori estimate to obtain an improved posteriori estimate. The measurement update equations are also known as corrector equations.

$$\hat{x}_k = \hat{x}_k^- + K_k.(z_k - H.\hat{x}_k^-), \tag{11}$$

$$K_k = \frac{\hat{P}_k^-.H^T}{H.\hat{P}_k^-.H^T + R}, \tag{12}$$

$$\hat{P}_k = (1 - K_k.H)\hat{P}_k^-. \tag{13}$$

Wiener filter deals with static data only; Kalman filter, a generalization of Wiener Filter [28] allows dynamic data with noisy parameters as input. Predictor model based on polynomial regression [29] uses combination of number of linear regression models which increases the computational complexity of calculations for each prediction manifolds. Extended Kalman Filter (EKF) [30] is an extension of Kalman filter, where at each step non-linear system is transferred to linear system by calculating first and second order derivative. Generally, EKF are considered for multi-class problems.

Simplicity of Kalman filter in implementation, less memory requirement and support for dynamic environment makes it a wonderful candidate for predicting the size of Bloom filter in streaming data.

3 ADAPTABLE BLOOM FILTER (ATBF)

To perform timely analysis on streaming data, an adaptive data structure is required which performs analysis in one pass with minimum computational complexity and less storage overhead. For a stream of network data $S : (x_1, x_2, \ldots, x_n)$ over a time based window of h time slots i.e. $T : (t_1, t_2, \ldots, t_h)$, this paper addresses the following points:

- Analysis of network traffic for a particular time slot.
- Predicting amount of in-coming data in the next slot.

- Allocation of memory for the next time slot based on prediction in the present time slot.

Proposed model is hybrid of two types of Bloom filters: scalable Bloom filter (for dynamic data input) and ageing Bloom filter (store data for particular time interval only). In the proposed framework, an efficient learning model is propounded for a time slot based analysis of network traffic using a novel technique called Adaptable Bloom Filter (ATBF), a variant of scalable Bloom filter. Figure 1 provides the basic framework and coming section elaborate the proposed framework along with its phases.

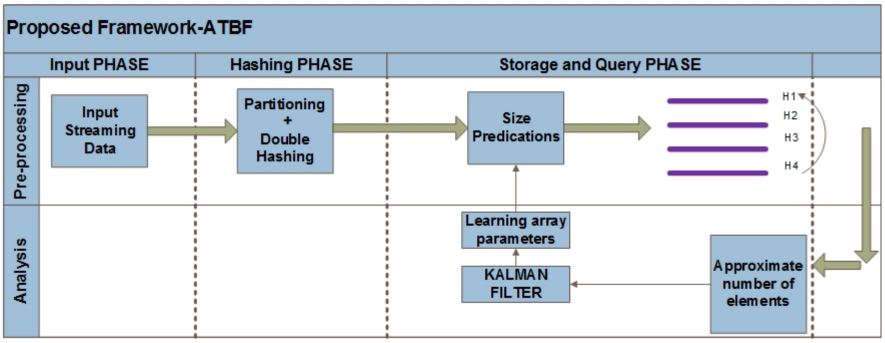


Figure 1. Proposed framework

3.1 Input and Hashing Phase

A stream of data $S = (x_1, x_2, \ldots, x_n)$ coming from any resource like sensor, social networking websites, network data and mobile data, etc., is assumed to be the input for the proposed framework. It is assumed that data is available only for limited time and hence it has to be processed in the single pass in the defined time frame. Data may be in varied formats like IP address for network data, website names, email address, etc.

In the proposed scheme, the format of the incoming data is not an issue as all the inputs irrespective of the format (numeric, alphanumeric, text) are hashed using a combination of double hashing and partition hashing. Two independent hash functions $h_1(x)$ and $h_2(x)$ are used to generate k hash functions such that each hash function has a disjoint range of $p = m/k$ (p must be prime for efficient hashing) consecutive bit locations (bucket) instead of having one shared array of m bits, i.e., partitioning hashing is used, where m is size of array, k is number of hash functions and p is prime number denoting the buckets in an array. $\forall i | i < k$

$$g_i(x) = \{h_1(x) + i \times h_2(x)\} \bmod p. \tag{14}$$

To achieve uniformity in maintaining bucket for each hash function at runtime, a parameter σ^i has been introduced, with initial value $\sigma^0 = p$. For each element $x \in S$ $(\forall i | i < k)$

$$H_i^j(x) = (h_1(x) + (i-1)h_2(x)) \bmod \sigma^j. \tag{15}$$

Corresponding to j^{th} slice added in i^{th} slot of ATBF, new σ^j is defined to synchronize bucket size for each hash function. $\Phi(x)$ function returns an optimal number p s.t. $p \leftarrow \Phi(p \geq x$ and p is prime).

3.2 Storage

After hashing is done for each incoming element, i.e. $\forall x_i \in S$, next task is to store the data in the array for a defined time slot say one hour or two hours. For each time slot, i.e. $t_i \in T$, a Bloom filter $(ATBF_i[])$ is maintained to store the elements for that particular time slot.

Selection of initial size of the Bloom filter for each slot in every iteration is critical task because it affects time and query complexity. For the very first iteration, an array of size m_0 is allocated and for further time slots size of Bloom filter is decided based on data received in the previous slot. Initial array size for each time slot t_h is decided on the basis of number of elements accommodated in previous slot t_{h-1}, using an array called Learning Array (LA) which keep the track of size of Bloom filter in each slot. The intent of providing an additional counting array is to reduce the slice addition overhead at the run time. The size of the array required for the next time slot is predicted through Kalman filter and each slot is provided the required slices at the beginning in the form of a single array instead of multiple chunks called slices. This process helps in adjusting the size of $ATBF_i[]$ to accommodate the dynamic input and reduces search time since query is done on single array instead of slices where we traverse from latest to oldest slice one by one. To maintain the uniformity in the partition hashing, size of slice is decided on the basis of $\phi()$ function. Insertion is performed by setting all hash indexed values one in the active slice of filter.

After t time slots when maximum number of time slot for which data records are maintained is reached, insertion is performed in first slot, i.e. $ATBF_1[]$, by evicting its old data. The proposed model works in round robin manner, i.e., slots after h hours perform insertion on same array during next iteration, i.e., $(t_i + o \times h) \leftrightarrow t_i$ where $o \in Z$. After completion of insertion in each time slot $InsertLA()$ function is invoked to update the values for performing size estimation for next time slot (Algorithm 1).

One of the major issues in SBF is how to measure the defined threshold for addition of the new slice. Number of solutions have been proposed for this issue which include 50 % percent rule, i.e., threshold is reached, when the maximum number of one's which a Bloom filter can accommodate reaches 50 % of its original capacity; but how to find that a filter is 50 % occupied is again a tedious task.

One of the options is to maintain a counter which increment every time an element is added or the number of one's in filter have to be counted after regular intervals. Another method is to keep the track of false positives after every insertion to check whether the results are within the desired false positive rate or not, but these solutions lead to extra computational overhead as one needs to continuously check when a filter gets saturated and such operations will definitely dilute the very purpose of using Bloom filter.

Proposed scheme addresses the issue of finding threshold for addition of new filter by the usage of buckets generated through partition hashing. Instead of calculating the threshold of the entire array, a function named *CheckFp*(), which uses standard threshold calculation technique, is used to find the threshold value of the randomly chosen bucket. Such technique limits the threshold calculation to a single bucket instead of entire array, reducing the overall computation time. To avoid calling *CheckFp*() after every iteration, a function *Random*() has been defined which returns a random value through which *CheckFp*() function is called, leading to further optimization of the entire process.

For experimental analysis, data is considered for varying time slots, e.g., one time slot is equal to four or six hours, i.e., all the hashed data of first time slot is added to the array $ATBF_{t1}$, data of second time slot moves to array $ATBF_{t2}$ and size of $ATBF_{t2}$ is determined by LA, based on the traffic in t_1 time slot. The proposed approach is flexible enough to accommodate n time slots, with each time slot represented by one array. Based on data stored in these Bloom filters, i.e. $ATBF_{1\ldots t_n}$, further analysis like peak hour analysis, detecting approximate number of users in each time slot and server utilization are performed.

3.3 Query Process in ATBF

To query the occurrence of a particular element in a time window, *Query*() function is used.

$Query(LA[], p, Q, T, _{i=1}^{i=k} H_i)$ in ATBF checks each Bloom filter, i.e. $ATBF_{t_i} | \forall t_i \in T$ from latest array to oldest array, and in each Bloom filter all slices (if added), i.e. from r to 1, are checked corresponding to the queried element. Query process is made fast by calculating hash functions at the run time, i.e. for a particular query, all hash functions are not computed in advance, each hash function is calculated and comparison is performed in defined bucket of hash function. If bit at hash index is one then next hash function is computed and comparison is performed otherwise query process terminates. Query process terminates as soon as first zero is encountered in a bucket and thus time is saved as remaining hash functions for other buckets are not calculated. The query process is terminated successfully if element is found, i.e. all ones are returned (Algorithm 2).

Algorithm 1 Insertion procedure in ATBF

```
1: procedure INSERT($ATBF[], p, S, T, _{i=1}^{i=k} H_i$) ▷ Insert $x_i \in S$ for $t_i \in T$ in $ATBF_i$ array
2:     for $\forall i | i \le T$ do
3:         $LA[i][][] \leftarrow InsertLA()$.
4:         $r_i \leftarrow 1$
5:         $\sigma^{r_i} \leftarrow \phi((LA[i] \times p)/k)$ ▷ Return optimal prime number according to variable size of filter
6:         $ATBF_i[r] \leftarrow SizeOf(\sigma^{r_i} \times k)$ ▷ Assign initial size to $i^{\text{th}}$ filter
7:         $C_{Slice} \leftarrow 1$
8:     end for
9:     for $\forall x_j \in S$ do
10:        while $t_c == t_i$ do ▷
11:            if $t_h ATBF_i[r] > thresVal$ then
12:                $\sigma^{r_i} \leftarrow \phi((s^{r-1} \times p)/k)$
13:                $r \leftarrow r + 1$
14:                $SizeOf(ATBF_i[r] \leftarrow \sigma^{r_i} \times k)$
15:                $C_{Slice} + +$
16:            else
17:                for $\forall z | z \le k$ do
18:                    $h_z(x_j) \leftarrow H_z(x_j)$
19:                    $ATBF_i[r](h_z(x_j)) \leftarrow HIGH$
20:                end for
21:            end if
22:            if $Random() == TRUE$ then
23:                $CheckFp(ATBF[r])$
24:            end if
25:        end while
26:        $InsertLA(FR_{LA[i]}, C_{slice})$
27:    end for
28: end procedure
```

Lemma 1. The worst case query time complexity in proposed model for filter with h time slots, assuming r slices in each slot with k hash functions is always less then $O(rhk)$.

Proof. Searching starts with hashing of the query element y, i.e. $\forall i | y \in Q, h_{i=1}^{k}(y) \leftarrow H_{i=1}^{k}(y)$ and corresponding hash indexes are checked for value zero. Query process begins from the latest time slot to the oldest one, i.e. $t_{h\text{to}1}$ and same is followed in search from slices $s_{r\text{to}1}$ in Bloom filter. During search operation when hash indexed value 0 is encountered, searching for that particular array is terminated and previous slice is not searched. In such case, number of evaluated for unsuccessful query, i.e. not finding the element queried, is always less than k hash functions. For a Bloom

Algorithm 2 Querying in proposed framework

```
1: procedure QUERY($LA[], p, Q, T, \overset{i=k}{\underset{i=1}{;}} H_i$)
2:     for ∀ Query elments($y$)|$y \in Q$ do
3:         for ∀ Time slots($t$)|$t \in T$ do
4:             for $l = (ATBF_t[.] \ldots 1)$ do
5:                 if $(ATBF_t[l](\overset{i\leq k}{\underset{i=1}{;}} h_i(y)) == 1)$ then
6:                     ELEMENT FOUND
7:                 end if
8:             end for
9:         end for
10:        ELEMENT NOT FOUND
11:    end for
12: end procedure
```

filter with r slices, it will be always less than $O(rk)$. Thus, for h time slots from $t_{1\ldots h}$ having r slices each, the worst case query complexity is always less than $O(rhk)$. $\square$

3.4 Learning Array (LA)

Since the amount of incoming data will keep on varying in every time slot, the size of array will change. Calculating the threshold after every addition and providing new slice accordingly in every time slot at run time requires lot of computation which can be saved if record of size of array, i.e. a counter C_{slice}, is maintained which keeps the count of number of slices added in a particular $ATBF_{ti}$ in a particular time slot. Initially a constant size Bloom filter m_0 is allocated for the first time slot and if the incoming data increases, more slices are added and counter c_{slice} is incremented. To make proposed framework adaptive, a Learning Array $LA[value][c]$ is initially added. The main role of LA is to record the array size of $ATBF_{ti}$ after filling of data in each time slot. This helps in predicting the array size required in the next time slot.

With the help of LA an optimal size of $ATBF_i[]$ required for successive time slots is decided. If for a time slots no slices are added, indicating unused Bloom filter bits, then value of LA is decremented for next time slot (Algorithm 3).

To make the functioning of LA more efficient, Kalman filter is used for predicting array size. The approximate number of elements are estimated through Algorithm 3 and the number of slices 'x' added to the initial filter in a particular time slot serves as input parameter to Kalman filter. After observing incoming data patterns for particular t_i, proposed model decides the optimal size required for next time slot, i.e. t_{i+1}, reducing the overhead of slice addition at run time for each time slot, thus improving the search time complexity of $ATBF_i[]$.

Number of slices (s_n) added to a particular time slot is recorded in LA, from this we can compute the total size of filter required for the particular time slot, i.e. $\hat{S}_s$.

The number of elements n_a accommodated by ATBF is given by:

$$n_a \approx m_0 s^i(\ln(t_h)). \tag{16}$$

From the approximate number of elements accommodated, the size of filter, i.e. $\hat{S}_e$, is calculated as:

$$\hat{S}_e = n_a \times k. \tag{17}$$

These two estimates for the size of Bloom filter act as input for Kalman filter and help the framework to predict the approximate size for coming time slots in further iterations.

Since the incoming data is one dimensional, Kalman filter parameters A, B, H, Q and R in Equations (7), (8), (9), (10), (11), (12), (13) have constant values in the proposed model. u_l is assumed to be zero because no control signal is used in the model. $\hat{S}_l$ denotes posterior estimated size and $\hat{S}_l^-$ denotes priori estimated size for ith time slot and for lth iteration. Thus

Time update:

$$\begin{aligned} \hat{S}_l^- &= \hat{S}_{l-1}, \\ \hat{P}_l^- &= \hat{P}_{l-1}. \end{aligned}$$

Measurement update:

$$\begin{aligned} K_l &= \frac{\hat{P}_{l-1}^-}{\hat{P}_{l-1}^- + R}, \\ \hat{S}_l &= \hat{S}_{l-1}^- + K_l(z_l + \hat{S}_{l-1}^-), \\ \hat{P}_l &= (1 - K_l)\hat{P}_{l-1}^- \end{aligned}$$

where

$$z_l = .5(\hat{S}_s + \hat{S}_e).$$

Lemma 2. Use of Kalman Filter based *LA* in proposed model reduces the query complexity of *ATBF* in handling in-stream data compared to *SBF* by approximate $O(\frac{1}{r})$ i.e. $\approx< O(k)$, where r is number of slices added and k is number of hash functions considered.

Proof. In case of SBF, when an array crosses defined threshold a new slice is added and insertion is performed. Assuming N_s is elements in stream, let us assume SBF needs r slices to accommodate the incoming data. Query process in SBF is accomplished by testing the presence of query element in each filter, starting from active filter to oldest filter. Search complexity for the worst case analysis is $O(k \times r))$.

In *ATBF* first time slot is functionally similar to SBF, but size for next time slot can be predicted using *LA* and Kalman filter. Predicting size for next time

Algorithm 3 Learning array algorithm

```
1: procedure InsertLA($LA[i], j$)                                        ▷
2:     if ($j > 1$) then
3:         if $LA[i] < j$ then
4:             $LA[i][c] + +$
5:         end if
6:     end if
7:     if ($j == 1$) then
8:         if $FR_{LA[i]} < thres_{fill}$ then
9:             $LA[i][] - = 1$
10:            EXIT
11:        end if
12:    end if
13:    $\hat{s}_n \leftarrow LA[i][c]$
14:    $\hat{S}_s \leftarrow m_0 \sum_{i=1}^{s_n}\{i \times \frac{m_0}{k}\}$
15:    $\hat{S}_e \leftarrow Count(ATBF_i[], r)$
16:    Set$\hat{S}_1^- = 0$
17:    Set$\hat{P}_1^- = 1$
18:    for ($l$ : 1 to $\ell$) do
19:        $z_l = .5(\hat{S}_s + \hat{S}_e)$
20:        Time update
21:        $\hat{S}_l^- = \hat{S}_{l-1}$
22:        $\hat{P}_l^- = \hat{P}_{l-1}$
23:        Measurement update
24:        $K_l = \frac{\hat{P}_{l-1}^-}{\hat{P}_{l-1}^- + R}$
25:        $\hat{S}_l = \hat{S}_{l-1}^- + K_l(z_l + \hat{S}_{l-1}^-)$
26:        $\hat{P}_l = (1 - K_l)\hat{P}_{l-1}^-$
27:    end for
28:    $LA[i][] \leftarrow \hat{S}_\ell$
29: end procedure
```

slot leads to decreased computational overhead as addition of new slices at run time is not required. Further, since the size of new array is combination of initial array and additional slices, inter-function collisions are reduced especially when partition hashing is used. From the second time slot onwards the query complexity is always less than $O(rk)$, because from the the second array onwards the number of new arrays added will always be less than r. In best case when no extra slice is added in future time slots, i.e., the input data arrival rate is constant, search complexity is equal to standard Bloom filter $\approx O(k)$. Thus, for the h time slots, search time complexity for $(h-1)$ slots is reduced drastically. □

3.5 Network Traffic Analysis for a Particular Time Slot

The standard algorithms for counting number of element in streams like CMS, probability based counter and DGIM are quite accurate but need lot of extra space and have computational overhead. Proposed model provides a rough estimate of number of elements using Kalman filter.

To calculate the approximate number of elements in a particular time slot t_i, $Count_i(.)$ is used with initial parameters like slices added in the array (r), threshold fill ratio (f_r), number of hash function (k), initial size of filter (m_0) and prime number used in first filter (p). Two methods have been used to calculate the number of elements in a particular time slot and results are verified by both methods (Algorithm 4). In the first method, growth parameter (s) are considered as $s = 2$ for slow growing data and $s = 4$ for fast growing data with optimal threshold t_h value as 50 % same as that considered in SBF [15]. Total number of elements accommodated by Bloom filter (N_i) is given by:

$$N_i \approx m_0 2^i * (.693). \tag{18}$$

The second method is to calculate the total size of the Bloom filter used and then predict the number of elements accommodated by it. Since σ^g is optimal prime for g^{th} slice, i.e., size of bucket and number of buckets are equal to number of hash functions (k), total size of an array with r slice of σ bits, is given by:

$$\text{Number of Slices}(r) \times \text{Size of Slice}(\sigma).$$

Thus total size (t_s) of $ATBF_i$ with r slices is given by:

$$t_s \leftarrow \sum_{g=1}^{r} (\sigma^g \times k). \tag{19}$$

Bits available for insertion in ATBF are determined by threshold fill ratio (f_r). Total available bits t_a are:

$$t_a \leftarrow t_s \times f_r. \tag{20}$$

Thus, maximum number of elements (E_a) accommodated by $ATBF_i$ are:

$$E_a \leftarrow \frac{t_a}{k}. \tag{21}$$

4 OBSERVATIONS AND ANALYSIS

All the experiments have been performed on i7-3612QM CPU @ 2.10 GHz with 8 GB of RAM. To maintain the uniformity in the results *CityHash* 64 bit library is used to compute two hash functions in double hashing. In all experiments five hash functions have been used with initial size of the filter m_0 as 1 285 bits, slice size σ^0 for all the iterations is considered as 275 ($s = \frac{1\,285}{5}$) for first array in all iterations.

Algorithm 4 Approximate number of elements in $ATBF_i[]$

```
 1: procedure $Count_i(ATBF_i[], r)$                    ▷
 2:     Method 1
 3:     $N_i \leftarrow \ln(f_r) \times m_0.s^r$
 4:     Method 2
 5:     for g:1 to r do
 6:         $\sigma^g \leftarrow \phi((g \times p)/k)$
 7:     end for
 8:     $t_s \leftarrow \sum_{g=1}^{r}(\sigma^g \times k)$
 9:     $E_a \leftarrow \frac{t_s.f_r}{k}$
10: end procedure
```

4.1 Performance Evaluation of SBF and ATBF

The performance of SBF and ATBF is compared on the basis of computational time taken for hashing, querying and extra slice addition as the incoming data increases. Figure 2 provides a comparative analysis on the basis of hashing complexity of SBF and *ATBF*. In *SBF*, for every input, hash value is computed for all hash functions (k) while in *ATBF* only two hash functions have been used to generate k hash functions, leading to a major decrease in computational overhead.

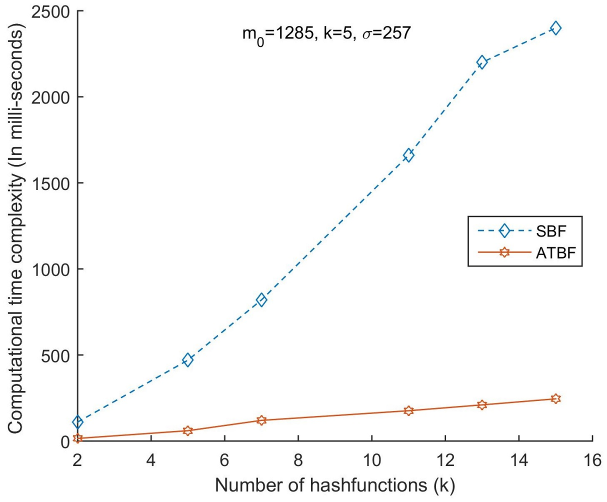


Figure 2. Computational time complexity vs. number of hash functions

Query complexity and slice addition overhead for both the filters is checked on dynamically growing environment. Both filters, i.e. SBF and ATBF, start with the size $m_0 = 1\,285$ and $5\,000$ elements have been considered for the first iteration and each iteration adds $1\,000$ element to previous value.

Figure 3 depicts the analysis performed on the basis of number of slices needed to accommodate the dynamically growing data. In SBF, filter starts with size m_0 and as the number of incoming elements increases more slices are added in each iteration. In case of ATBF, as the incoming data increases the size of t_{n+1}^{th} iteration is predicted in advance, based on the elements accommodated per iteration in t_n^{th} using Kalman filter.

Based upon the data considered for experiments, i.e. $1\,000$ elements increase from previous value per iteration, in t_{n+1}^{th} iteration only one slot is added to accommodate additional elements in ATBF. The graph of ATBF becomes constant after the first iteration since one slice is added in every successive iteration and no overflow of data is registered (Figure 3). Hence overhead of adding new slices at the run time is reduced to a large extent in the proposed scheme.

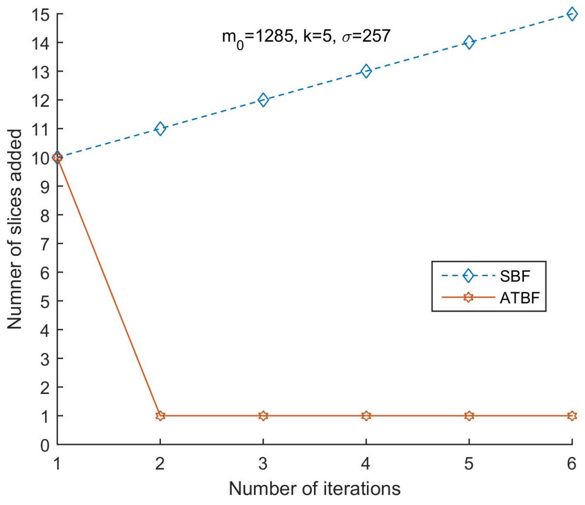


Figure 3. Number of slices required vs. number of iterations for dynamically growing dataset

Figure 4 depicts the comparative analysis of the worst case query complexity for an element (when the element is not present in the set), i.e., scenario where all slices need to be scanned. As the size of data grows in each iteration in SBF, more slices are added to accommodate the data elements. In SBF, all slices need to be scanned in query process which increases the query complexity many folds. ATBF has the advantage of size adaptation from second iteration onwards. For the first iteration the process is similar to SBF, but from the i^{th} iteration (where $i \neq 1$), the size of Bloom filter is predicted on the basis of previous $(i-1)^{\text{th}}$ iteration. The predicted size of Bloom filter is added as a single Bloom filter. So, in querying process only one Bloom filter needs to be scanned, thus the total cost is $O(k)$. As the data grows, the number of slices added are always less than SBF for same number of elements thus search time complexity of ATBF shows a significant improvement.

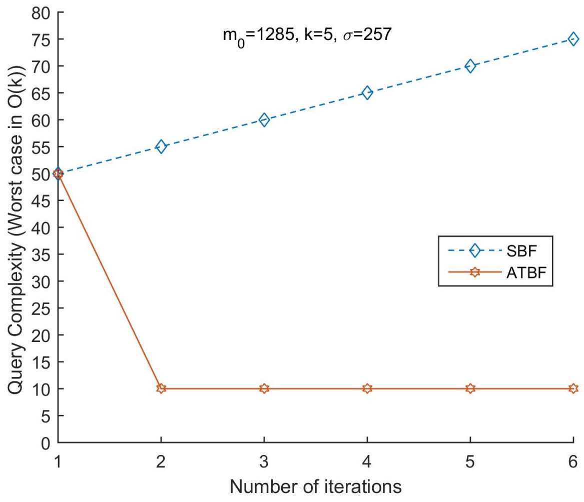


Figure 4. The worst case query complexity vs. number of iterations

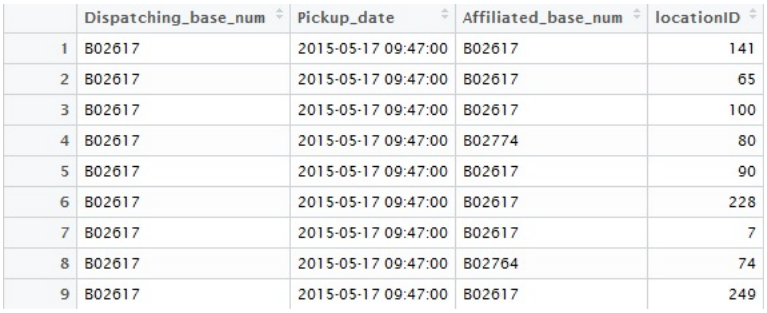
4.2 Experimental Evaluations

Two data sets from different application domains have been considered for evaluating the performance of proposed model, one is data of pickup calls of Uber cabs [31] and other is incoming data generated for network server. Results are represented for first few iterations only, which can be extended to n number of iterations according to application's requirements.

Tables 2 and 3 provide the count of actual number of users and number of users identified using Kalman filter. $\hat{S}^-$ represents the size of Bloom filter in the current iteration by considering the previous one, initially size of Bloom filter is set to m_0. Number of slices added in $ATBF_i$ is maintained by c_{slice} counter. $\hat{S}$ is the array size predicted by Kalman filter for the next iteration. Peak hours analysis is performed by "peak hour ranking" with 1 indicating maximum and 5 as minimum value. Peak hour rank helps in identifying changing patterns of data in current iteration in relation to the previous iteration. Initially for all iteration, the peak hour rank is set to a default value of -1. This ranking system helps in allocating resources in accordance with the frequency of incoming data.

4.2.1 Experiment 1: Uber Pickups Data Sets

Data of 14 270 479 trips of Uber pickups in New York City from January 2015 to June 2015 for around 265 different locations is considered for 12 hours a day as input. The data set is time series based having attributes like date, time, location id and base number. A snapshot of an instance of data is shown in Figure 5. In proposed model "location id" is used as insertion element in Bloom filter and attributes "Pickup_date" and "Time" are used to select the size of a time slot.



	Dispatching_base_num	Pickup_date	Affiliated_base_num	locationID
1	B02617	2015-05-17 09:47:00	B02617	141
2	B02617	2015-05-17 09:47:00	B02617	65
3	B02617	2015-05-17 09:47:00	B02617	100
4	B02617	2015-05-17 09:47:00	B02774	80
5	B02617	2015-05-17 09:47:00	B02617	90
6	B02617	2015-05-17 09:47:00	B02617	228
7	B02617	2015-05-17 09:47:00	B02617	7
8	B02617	2015-05-17 09:47:00	B02764	74
9	B02617	2015-05-17 09:47:00	B02617	249

Figure 5. An instance from data set of Uber pickups

Iteration	**No. of actual users**	**Initial array size ($\hat{S}^{-}$)**	**No. of slots added (C_{slice})**	**No. of users predicted by ATBF**	**Error (In %)**	**Size of Bloom filter predicted for next time slot (in bits) ($\hat{S}$)**	**Previous Peak hour ranking**	**Current Peak hour ranking**
Time Slot = 4 hours								
1/1/2015								
Time Slot 1 (1 to 4) hrs.	5 864	1 285	11	5 746	2.01	28 160	−1	1
Time Slot 2 (5 to 8) hrs.	2 389	28 160	0	2 358	1.3	24 320	−1	3
Time Slot 3 (9 to 12) hrs.	2 922	24 320	0	2 935	−0.4	20 736	−1	2
2/1/2015								
Time Slot 1 (1 to 4) hrs.	1 765	1 285	4	1 732	1.9	8 960	−1	3
Time Slot 2 (5 to 8) hrs.	2 437	8 960	2	2 387	2.1	14 336	−1	2
Time Slot 3 (9 to 12) hrs.	2 534	14 336	0	2 456	−0.9	20 736	−1	1
Time Slot = 6 hours								
1/1/2015								
Time Slot 1 (1 to 6) hrs.	7 314	1 285	12	7 287	0.4	36 660	−1	1
Time Slot 2 (7 to 12) hrs.	2 326	36 660	0	2 342	−0.6	32 256	−1	2
2/1/2015								
Time Slot 1 (1 to 6) hrs.	2 915	1 285	7	2 867	1.7	17 408	−1	1
Time Slot 2 (7 to 12) hrs.	3 312	17 408	0	3 264	1.5	17 408	−1	2

Table 2. Bloom filter size prediction and Peak hour analysis for Uber pickup call for 1st January 2015 and 2nd January 2015

Table 2 shows the result of two days for peak time slot in Uber pickups, for date 1st January 2015 and 2nd January 2015 using two time slot ranges: four hours as a single time slot and six hours as a single time slot, respectively.

4.2.2 Experiment 2: Incoming Data on a Network Server

Table 3 provides the results for server utilization and peak hour analysis. Experiment is done for six time slots of one hour each. The results are simulated on network traffic with maximum per hour capacity of server as 15 000 users. Server utilization is given by $(\frac{n}{N} \times 100)$, where n is approximate number of users detected and N is server capacity. The network data has IP address, date and time as its attributes. IP address is used as primary element for insertion in proposed model.

Iteration1	**No. of actual users**	**Initial array size ($\hat{S}^{-}$)**	**No. of slots added**	**No. of users predicted by ATBF**	**Error (In %)**	**Size of Bloom filter predicted for next time slot (in bits) ($\hat{S}$)**	**Server utilization (%)**	**Previous Peak hour ranking**	**Current Peak hour ranking**
Time slot 1	10 000	1 285	15	9 975	0.25	51 200	68.53	−1	2
Time slot 2	12 000	51 200	2	12 145	−1	62 210	82.97	−1	1
Time slot 3	9 000	62 210	0	9 216	−2	46 080	61.44	−1	3
Time slot 4	6 000	46 080	0	6 052	−0.8	32 256	43.01	−1	5
Time slot 5	8 000	32 256	2	7 952	0.6	41 216	54.97	−1	4
Time slot 6	4 000	41 216	0	3 924	1.9	20 736	27.65	−1	6
Iteration 2									
Time slot 1	9 000	1 285	14	8 982	0.2	46 080	60.84	2	3
Time slot 2	14 000	46 080	5	14 248	−1.7	74 240	98.99	1	1
Time slot 3	13 000	74 240	0	13 184	−1	70 400	92.17	3	2
Time slot 4	7 000	70 400	0	7 013	−1	36 352	48.09	5	4
Time slot 5	3 000	36 352	0	2 989	0.4	21 365	23.21	4	6
Time slot 6	4 000	21 365	0	3 968	0.8	20 736	27.65	6	5

Table 3. Hourly analysis of server utilization, the peak hour and Bloom filter size prediction for next time slot for incoming data on a network

5 CONCLUSION

In-stream data analytics works by processing data in a defined time windows. To accommodate dynamic data and query the hourly information, the proposed framework uses Bloom filter with Ageing Bloom filter properties, i.e., evicting data after fixed time interval. Partition hashing has been used which leads to less inter-hash function collision. Further usage of double hashing where only two hash functions are used to generate all k hash functions decreases the computational overhead.

A learning array has been introduced which stores the size of Bloom filter required in the next iteration and Kalman filter has been used to predict the size of Bloom filter required for the next iteration. Results achieved clearly indicate that the proposed framework performs efficiently for the peak hour analysis and server utilization analysis.

Acknowledgment

The first author would like to acknowledge the financial support given to him by the Department of Computer Science and Technology under the Department of Electronics and Information Technology (DeitY) to complete doctoral studies.

REFERENCES

[1] Mayer-Schönberger, V.—Cukier, K.: Big Data: A Revolution That Will Transform How We Live, Work, and Think. John Murray Publishers, UK, 2013.

[2] Gubbi, J.—Buyya, R.—Marusic, S.—Palaniswami, M.: Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions. Future Generation Computer Systems, Vol. 29, 2013, No. 7, pp. 1645–1660.

[3] Amazon. What is Cloud Computing? `http://aws.amazon.com/what-is-cloud-computing/`, 2013.

[4] Krishnan, K.: Data Warehousing in the Age of Big Data. 1st edition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.

[5] Labrinidis, A.—Jagadish, H. V.: Challenges and Opportunities with Big Data. Proceedings of VLDB Endowment, Vol. 5, 2012, No. 12, pp. 2032–2033.

[6] Babcock, B.—Babu, S.—Datar, M.—Motwani, R.—Widom, J.: Models and Issues in Data Stream Systems. Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '02), ACM, New York, NY, USA, 2002, pp. 1–16, doi: 10.1145/543613.543615.

[7] UN Global Pulse: Big Data for Development: Challenges and Opportunities. `http://www.unglobalpulse.org/projects/BigDataforDevelopment/`, 2012.

[8] Liberty, E.—Nelson, J.: Streaming Data Mining. Presented at Princeton University by Yahoo Research Group.

[9] Ylijoki, O.—Porras, J.: Conceptualizing Big Data: Analysis of Case Studies. Intelligent Systems in Accounting, Finance and Management, Vol. 23, 2016, No. 4, pp. 294–310, doi: 10.1002/isaf.1393.

[10] Bobadilla, J.—Ortega, F.—Hernando, A.—Gutiérrez, A.: Recommender Systems Survey. Knowledge-Based Systems, Vol. 46, 2013, pp. 109–132, doi: 10.1016/j.knosys.2013.03.012.

[11] Bloom, B. H.: Space/Time Trade-Offs in Hash Coding with Allowable Errors. Communications of the ACM, Vol. 13, 1970, No. 7, pp. 422–426, doi: 10.1145/362686.362692.

[12] Katsov, I.: Probabilistic Data Structures for Web Analytics and Data Mining. `http://highlyscalable.wordpress.com/2012/05/01/probabilistic-structures-web-analytics-data-mining/`, 2012.

[13] Geravand, S.—Ahmadi, M.: Survey Bloom Filter Applications in Network Security: A State-of-the-Art Survey. Computer Networks, Vol. 57, 2013, No. 18, pp. 4047–4064, doi: 10.1016/j.comnet.2013.09.003.

[14] Tarkoma, S.—Rothenberg, C. E.—Lagerspetz, E.: Theory and Practice of Bloom Filters for Distributed Systems. IEEE Communications Surveys and Tutorials, Vol. 14, 2012, No. 1, pp. 131–155, doi: 10.1109/SURV.2011.031611.00024.

[15] Almeida, P. S.—Baquero, C.—Preguiça, N.—Hutchison, D.: Scalable Bloom Filters. Information Processing Letters, Vol. 101, 2007, No. 6, pp. 255–261, doi: 10.1016/j.ipl.2006.10.007.

[16] Kalman, R. E.: A New Approach to Linear Filtering and Prediction Problems. Transactions of the ASME, Journal of Basic Engineering, Vol. 82, 1960, Series D, pp. 35–45, doi: 10.1115/1.3662552.

[17] Xie, K.—Min, Y.—Zhang, D.—Wen, J.—Xie, G.: A Scalable Bloom Filter for Membership Queries. IEEE Global Telecommunications Conference (GLOBECOM '07), 2007, pp. 543–547, doi: 10.1109/GLOCOM.2007.107.

[18] Chang, F.—Feng, W.-C.—Li, K.: Approximate Caches for Packet Classification. Twenty-Third Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2004), 2004, Vol. 4, pp. 2196–2207, doi: 10.1109/INFCOM.2004.1354643.

[19] Yoon, M.: Aging Bloom Filter with Two Active Buffers for Dynamic Sets. IEEE Transactions on Knowledge and Data Engineering, Vol. 22, 2010, No. 1, pp. 134–138.

[20] Kirsch, A.—Mitzenmacher, M.: Less Hashing, Same Performance: Building a Better Bloom Filter. Random Structures and Algorithms, Vol. 33, 2008, No. 2, pp. 187–218, doi: 10.1002/rsa.20208.

[21] Moore, J. S.: A Fast Majority Vote Algorithm. Technical Report ICSCA-CMP-32, Institute for Computer Science, University of Texas, 1981.

[22] Boyer, R. S.—Moore, J. S.: MJRTY – A Fast Majority Vote Algorithm. Automated Reasoning, Springer, 1991, pp. 105–117, doi: 10.1007/978-94-011-3488-0_5.

[23] Manku, G. S.—Motwani, R.: Approximate Frequency Counts over Data Streams. Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02), 2002, pp. 346–357, doi: 10.1016/B978-155860869-6/50038-X.

[24] Metwally, A.—Agrawal, D.—El Abbadi, A.: Efficient Computation of Frequent and Top-k Elements in Data Streams. In: Eiter, T., Libkin, L. (Eds.): Database Theory (ICDT 2005). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 3363, 2004, pp. 398–412.

[25] Cormode, G.—Muthukrishnan, S.: An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. Journal of Algorithms, Vol. 55, 2005, No. 1, pp. 58–75, doi: 10.1016/j.jalgor.2003.12.001.

[26] Charikar, M.—Chen, K.—Farach-Colton, M.: Finding Frequent Items in Data Streams. In: Widmayer, P., Eidenbenz, S., Triguero, F., Morales, R., Conejo, R., Hennessy, M. (Eds.): Automata, Languages, and Programming (ICALP 2002).

Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 2380, 2002, pp. 693–703.

[27] Jain, A.—Chang, E. Y.—Wang, Y.-F.: Adaptive Stream Resource Management Using Kalman Filters. Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04), ACM, New York, NY, USA, 2004, pp. 11–22, doi: 10.1145/1007568.1007573.

[28] Wiener, N.: Extrapolation, Interpolation, and Smoothing of Stationary Time Series. MIT Press Cambridge, MA, 1949.

[29] Shaw, P.—Greenstein, D.—Lerch, J.—Clasen, L.—Lenroot, R.—Gogtay, N.—Evans, A.—Rapoport, J.—Giedd, J.: Intellectual Ability and Cortical Development in Children and Adolescents. Nature, Vol. 440, 2006, No. 7084, pp. 676–679.

[30] Julier, S. J.—Uhlmann, J. K.: A New Extension of the Kalman Filter to Nonlinear Systems. Signal Processing, Sensor Fusion, and Target Recognition VI (AeroSense '97). Proceedings of the SPIE, Vol. 3068, 1997, pp. 182–193, doi: 10.1117/12.280797.

[31] FiveThirtyEight. Uber Pickups in New York City. `https://www.kaggle.com/fivethirtyeight/uber-pickups-in-new-york-city`, 2016.

Amritpal Singh received his M.Eng. 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 January 2015. He served both industry and academia. His research interests include probabilistic data structures, machine learning and big data.

Shalini Batra received her Ph.D. degree in computer science and engineering from Thapar University, Patiala, India, in 2012. She is currently working as 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.Eng./M.Tech. She has authored more than 60 research papers published in various conferences and journals. Her research interests include machine learning, web semantics, big data analytics and vehicular ad-hoc networks.