Neural Computing and Applications
https://doi.org/10.1007/s00521-019-04397-1

ORIGINAL ARTICLE

Autoscaling Bloom filter: controlling trade-off between true and false positives

Denis Kleyko[1] · Abbas Rahimi[2] · Ross W. Gayler[3] · Evgeny Osipov[1]

Received: 2 February 2018 / Accepted: 18 July 2019

Abstract
A Bloom filter is a special case of an artificial neural network with two layers. Traditionally, it is seen as a simple data structure supporting membership queries on a set. The standard Bloom filter does not support the delete operation, and therefore, many applications use a counting Bloom filter to enable deletion. This paper proposes a generalization of the counting Bloom filter approach, called "autoscaling Bloom filters", which allows adjustment of its capacity with probabilistic bounds on false positives and true positives. Thus, by relaxing the requirement on perfect true positive rate, the proposed autoscaling Bloom filter addresses the major difficulty of Bloom filters with respect to their scalability. In essence, the autoscaling Bloom filter is a binarized counting Bloom filter with an adjustable binarization threshold. We present the mathematical analysis of its performance and provide a procedure for minimizing its false positive rate.

Keywords Bloom filter · Counting Bloom filter · Autoscaling Bloom filter · True positive rate · False positive rate

1 Introduction

Many applications require fast and memory-efficient querying of an item's membership in a set. A Bloom filter (BF) is a simple binary data structure, which supports approximate set membership queries.

From a neural processing point of view, BFs are a special case of an artificial neural network with two layers (input and output), where each position in a filter is implemented as a binary neuron (see more details in [1]). Such a network does not have interneuronal connections. That is, output neurons (positions of the filter) have only individual connections with themselves and the corresponding input neurons. BFs are also related to a neural network architecture called distributed connectionist production system [2].

The standard BF (SBF) allows adding new elements to the filter and is characterized by a perfect true positive rate (i.e., 1), but nonzero false positive rate. The false positive rate depends on the number of elements to be stored in the filter, and the filter's parameters, including the number of hash functions and the size of the filter. However, SBF lacks the functionality of deleting an element. Therefore, a counting Bloom filter (CBF) [3], providing the delete operation, is commonly used. When the size of CBF and the number of elements to be stored are known, the number of hash functions can be optimized to minimize the false positive rate.

Another practical issue is that the parameters of a BF (i.e., size of filter and number of hash functions) cannot be altered once it is constructed. If the current filter does not satisfy the performance requirements (e.g., false positive rate), it is necessary to rebuild the entire filter, which is computationally expensive. Therefore, the optimization of a BF is problematic and costly when the number of elements to be stored is unknown or varies dynamically. In fact, this is one of the major scalability difficulties of BFs. This paper presents a solution allowing overcoming it.

✉ Denis Kleyko
denis.kleyko@ltu.se

Abbas Rahimi
abbas@ee.ethz.ch

Evgeny Osipov
evgeny.osipov@ltu.se

[1] Luleå University of Technology, Luleå, Sweden

[2] ETH Zurich, Zurich, Switzerland

[3] Melbourne, Australia

Published online: 09 August 2019

To address the issue of optimizing BF performance without rebuilding the filter, we propose the autoscaling Bloom filter (ABF), which is derived from a CBF and allows minimization of the false positive rate in response to changes in the number of stored elements without requiring rebuilding of the entire filter. The reduction in false positive rate is achieved by optimizing a threshold parameter used to derive the ABF from the CBF. ABF operates with fixed resources (i.e., fixed size storage array and fixed k hash functions) for a wide dynamic range of number of input elements to be stored. The trade-off made by ABF for this flexibility is a slight reduction of the true positive rate (which is always 1 in CBF). It is important to note that a less than perfect true positive rate can be tolerated in many applications including networking [4], and generally in the area of approximate computing where errors and approximations are acceptable as long as the outcomes have a well-defined statistical behavior [5]. To the best of our knowledge, ABF is a novel simple construction of BFs, which makes them particularly useful in scenarios where a reduced true positive rate can be tolerated and where the number of stored elements is unknown or changes dynamically with time.

ABF belongs to a class of binary BFs and is constructed by binarization of a CBF with the binarization threshold (Θ) as a parameter. Querying the ABF also uses a decision threshold (T) to determine whether there is sufficient evidence to respond that the query item is an element of the stored set. Both parameters, Θ and T, can be varied, while the ABF is in use without requiring the filter data structure to be rebuilt. Figure 1 illustrates the main idea behind the ABF. Figure 1a shows an example CBF of size 20, which stores four elements (x_1 to x_4). Each element is mapped to three different positions of the filter, one position for each of the three hash functions. The value at each position is the number of elements mapped to that position by the three hash functions and varies between 0 and 4 (highlighted by different colors). The SBF (Fig. 1b) is formed by setting all nonzero positions of the CBF to one[1]. The two lower parts of the figure denoted as (c) and (d) present two examples of the ABF for two different sets of parameters: $\Theta = 1$ and $T = 2$; $\Theta = 3$ and $T = 1$, respectively. In all four examples, the filter is queried with the unstored element y, testing for membership of the set of stored elements. The correct answer in every case, obviously, is that y is not a member of the stored set. In the SBF example, all nonzero positions of y are set to one, which is interpreted by the SBF algorithm as indicating that the query element is a member of the stored set, thus generating a false positive response. In contrast, in Fig. 1c, y has only one position in common with the ABF, while all elements x_i have at least two positions. Thus, if a decision threshold T (for the number of activated positions) is set to two, then y will be correctly rejected by the ABF while all the stored elements are correctly reported as present. On the other hand, for the ABF in Fig. 1d, the binarization threshold ($\Theta = 3$) is too low and it is not possible to set a decision threshold T (even the smallest possible $T = 1$) such that all stored elements x_i are reported as present.

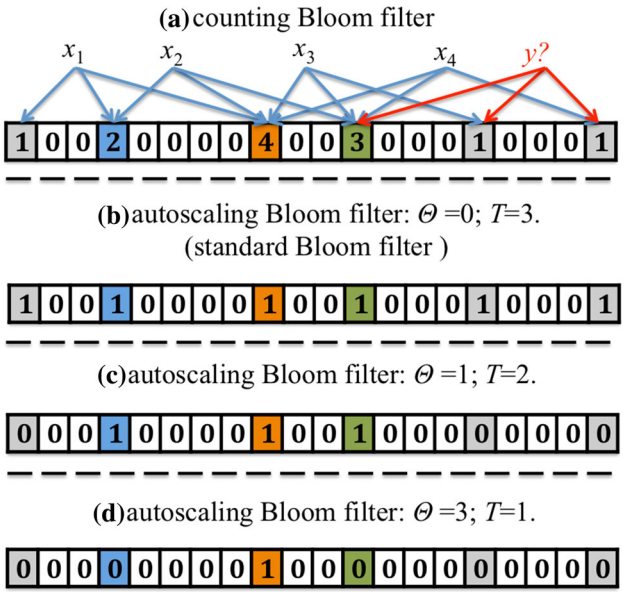

Fig. 1 **a** An example of the counting Bloom filter with size $m = 20$. The number of hash functions applied to each element was set to $k = 3$. The filter included $n = 4$ elements. **b** The standard Bloom filter derived from the counting Bloom filter. Note that it is equivalent to the autoscaling Bloom filter with $\Theta = 0$ and $T = k = 3$. **c** The autoscaling Bloom filter derived from the counting Bloom filter. The autoscaling Bloom filter parameters were set to $\Theta = 1$ and $T = 2$. **d** The autoscaling Bloom filter derived from the counting Bloom filter. The autoscaling Bloom filter parameters were set to $\Theta = 3$ and $T = 1$

Mathematically, the ABF has its roots in the theory of sparse distributed data representations [6]. ABF can also be interpreted in terms of hyperdimensional computing [7], where everything is represented as high-dimensional vectors and computation is implemented by arithmetic operations on the vectors. Both sparse distributed representations and hyperdimensional computing can be conceptualized as weightless artificial neural networks.

This paper presents a theoretical generalization of CBFs by exploring a direct correspondence between BFs and hyperdimensional representations along with the practical implications. BFs are treated as a special case application of distributed representations where each element stored in the BF is represented as a hyperdimensional binary vector constructed by the hash functions. The mathematics of

[1] Note that the SBF is a special case of the ABF, arising when the binarization threshold is set to zero and the decision threshold is set to the number of used hash functions.

sparse hyperdimensional computing [6] (SHC) is used for describing the behavior of the proposed ABF. The construction of the filter itself corresponds to the bundling operation [6] of binary vectors.

The main contributions of the paper are as follows:

- It proposes the ABF, which is a generalization of the CBF with probabilistic bounds on false positives and true positives;
- It presents the mathematical analysis and experimental evaluation of the ABF properties;
- It gives a procedure for *automatic* minimization of the false positive rate adapting to the number of the elements stored in the filter;
- For the first time, it shows that BFs are a special case of hyperdimensional computing.

The paper is structured as follows: Sect. 2 presents a concise survey of the related approaches. Section 3 describes the ABF and introduces analytical expressions characterizing its performance. The evaluation of the ABF is presented in Sect. 4. The paper is concluded in Sect. 5.

2 Related work

A recent probabilistic analysis of the SBF is presented in [8]. Detailed surveys on BFs and their applications are provided in [9] and [10]. BFs are often applied in the area of pattern recognition [11, 12]. For example, recent applications of BFs and their modifications include certificate revocation for smart grids [13], classification of text strings [14], and detection of Transmission Control Protocol (TCP) network worms [15]. An important aspect for the applicability of BFs in modern networking applications is the processing speed of a filter. In order to improve the speed of the membership check, the authors in [16] proposed a novel filter type called ultra-fast BFs. In [17], it was shown that BFs can be accelerated (in terms of processing speed) by using particular types of hashing functions.

This section overviews the approaches most relevant to the presented ABF approach. One direction of research is to propose new types of data structures supporting approximate membership queries. For example, recently proposed invertible Bloom lookup tables [18], quotient filters [19], counting quotient filters [20], TinySet [21], and cuckoo filters [22] support dynamic deletion. Another popular research topic is to improve the performance of the SBF via modifications of the original approach. The ternary BF [23] improves the performance of the CBF as it only allows three possible values of each position. The deletable BF [24] uses additional positions in the filter, which are used to support the deletion of elements from the filter without introducing false negatives. The complement Bloom Filter [25] uses an additional BF in order to identify the trueness of BF positives. The on–off BF [26] reduces false positives by including in the filter additional information about those elements that generate false positives. Fingerprint counting BF [27] is a modification improving the CBF with the usage of fingerprints on the filter elements. In [13], the authors propose to use two BFs and an external mechanism in order to resolve cases when the membership is confirmed by both filters. In a similar fashion, the cross-checking BF [28] constructs several additional BFs, which are used to cross-check the main BF if it issues a positive result. The scalable Bloom filter [29] can maintain the desired false positive rate even when the number of stored elements is unknown. However, it has to maintain a series of BFs in order to do so. Another related approach called variable-increment CBF (VICBF) was presented in [30]. Similar to the CBF, the VICBF supports the delete operation; however, it requires less memory for achieving the same false positive rate. The improvement is due to the usage of a hashed variable increment rather than a counter increment as in the CBF. In comparison with the VICBF, the ABF could fully operate with the binary components; however, it would lose the ability to delete elements. Nevertheless, once the VICBF is designed, it does not have the built-in functional to tolerate large variations in the number of stored elements. While the VICBF is a generalization of the CBF, it would not be trivial to apply the ABF to a given VICBF as the different variable increments values are used to get the final values in each position of a filter. The retouched BF (RBF) [4] is conceptually the most relevant approach to the ABF since it allows some false negatives as a trade-off for decreasing the false positive rate. The major difference to the proposed approach is that RBF eliminates false positives that are known in advance. When the potential false positives are not known in advance, the RBF could randomly erase several nonzero positions of the filter.

In contrast to the previous work, the ABF is suitable for reducing the false positive rate even when the whole universe of elements is either unknown or is too large to use additional mechanisms for encoding the elements not included in the filter.

3 Autoscaling Bloom filter

3.1 Preliminaries: BFs

At the initialization phase, a BF can be seen as a vector of length m where all positions are set to zero. The value of m determines the size of the filter. In order to store in the filter an element q, from the universe of elements, the

element should be mapped into the filter's space. This process is usually seen as application of k different hash functions to the element. The result of each hash function is an integer between 1 and m. This value indicates the index of the position of the filter which should be updated. In the case of the SBF, an update corresponds to setting the value of the corresponding position of the SBF to 1. If the position already has value 1, it stays unchanged. In the case of the CBF, an update corresponds to incrementing the value of the corresponding position of the CBF by 1. Thus, when storing a new element in the filter, at most k positions of the filter update their values. Note that there is a possibility that two or more hash functions return the same result. In this case, there would be less than k updated positions. However, it is usually recommended to choose hash functions such that they have a negligible probability of returning the same index value. Therefore, without loss of generality, suppose that the k results of k hash functions applied to q never coincide. That is, all k indices pointing to positions in the filter are unique.

Instead of considering the result of mapping q as the k indices produced by the hash functions, it is convenient to represent the mapping in the form of the SBF that stores the single element q. This SBF is sometimes called the individual BF. It is a vector with m positions, where values of only k positions are set to one, and the rest to zero. The nonzero positions are determined by the hash functions applied to q. The representation of an element q in this form is denoted as $\mathbf{q}$. Note that throughout this section bold terms denote vectors. Given this vectorized form of representation, the CBF (denoted as **CBF**) storing a set of n elements x_i can be calculated as the sum of representations (denoted as $\mathbf{x}_i$) of each individual element x_i in the set:

$$\mathbf{CBF} = \sum_{i=1}^{n} \mathbf{x}_i. \tag{1}$$

The SBF (denoted as **SBF**) representing the set of elements is related to the CBF representing the same set of elements as follows:

$$\mathbf{SBF} = [\mathbf{CBF} > 0], \tag{2}$$

where [] means 1 if true and 0 otherwise (applied elementwise to the argument vector).

Given the values of m and n, the value of k that minimizes the false positive rate (see also [31, 32] for recent improvements) for the SBF (CBF) can be found as:

$$k = (m/n) \ln 2. \tag{3}$$

When performing the set membership query operation with query element q (represented by $\mathbf{q}$) on an SBF containing q, the dot product (d) between **SBF** and $\mathbf{q}$ must equal the number of nonzero positions in $\mathbf{q}$, i.e., k:

$$d(\mathbf{SBF}, \mathbf{q}) = \mathbf{SBF} \cdot \mathbf{q} = k \tag{4}$$

3.2 Preliminaries: probability theory

Two probability distributions are useful for the analysis presented here. These are binomial and hypergeometric distributions. Both are discrete. They describe the probability of s successes (draws for which the drawn entities are defined as successful) in g random draws from a finite population of size G that contains exactly S successful entities. The difference between binomial and hypergeometric distributions is that the binomial distribution describes the probability of s successes in g draws with replacement, while the hypergeometric distribution describes the probability of s successes in g draws without replacement. Binomial and hypergeometric distributions are the most natural choice for modeling BFs since they correspond to the discrete nature of values in BFs. It is worth mentioning that when the number of random draws g is large, both distributions could be approximated by normal or Poisson distributions depending on relations between g, s, and G. We do not use the approximations in this paper as this allows avoiding errors introduced by approximations.

Note that if 1 denotes a successful draw while 0 denotes a failure draw, then we can represent g draws from a distribution as a binary vector of length g. This binary vector corresponds to a realization of a (hypergeometric/binomial) experiment. The probability of a success in a particular position of the realization for both distributions is:

$$p_s = S/G. \tag{5}$$

The difference is that for the binomial distribution positions are independent while for the hypergeometric distribution they are not. For example, if the actual values of some positions are known for the realization of a hypergeometric experiment, then the probability of a success for the rest of the positions should be updated accordingly. This is because draws from the population are done without replacement.

If the random variable Z is described by the binomial distribution (denoted as $Z \sim \mathrm{B}(g, p_s)$), then the probability of getting exactly s successes in g draws is described by the probability mass function:

$$\Pr(Z = s) = \binom{g}{s} p_s^s (1 - p_s)^{g-s}. \tag{6}$$

As the probability mass function for the hypergeometric distribution is not used below, it is omitted here.

3.3 Preliminaries: relation between BFs and probability theory

The hypergeometric distribution comes into play when considering the mapping of an element q. Given the assumption that the results of hash functions do not coincide, the mapping $\mathbf{q}$ of an element q is a binary vector of length m with exactly k positions having value 1 and the rest 0. It is worth noting that this assumption is very realistic since it is a usual requirement during the design of a filter that the used hash functions are independent, and therefore, for large filters, there is a small chance of overlapping. The assumption will introduce a subtle difference as discussed below, and however, this difference is only important for impractical small lengths of the filter. Because hash functions map different elements into different indices, a mapping $\mathbf{q}$ can be seen as a single realization of the experiment from the hypergeometric distribution with $g = m$ draws from the finite population of size $G = m$ that contains exactly $S = k$ successes (positions set to 1). In this case $g = G$. Therefore, the probability of exactly $s = k$ successes is 1 and all other probabilities are 0. The probability of a success in a particular position is:

$$p_1 = p_s = k/m. \tag{7}$$

A value in ith position of **CBF** [see (1)] can be seen as a discrete random variable (denoted as I) in the range $I \in \mathbb{Z} | 0 \le I \le n$, where n denotes the number of elements stored in a filter. Because representations $\mathbf{x}_i$ stored in **CBF** are independent realizations of the hypergeometric experiment, I follows the binomial distribution: $I \sim \mathrm{B}(g, p_s)$ where $g = n$, $p_s = p_1$.

Given the parameters of the binomial distribution, the probability that I takes the value v can be calculated according to (6):

$$\Pr(I = v) = \binom{n}{v} p_1^v (1 - p_1)^{n-v}. \tag{8}$$

According to (8), the probability of an empty position p_0 in the CBF (and also for SBF) is:

$$p_0 = \Pr(I = 0) = \left(1 - \frac{k}{m}\right)^n. \tag{9}$$

It should be noted that the probability of an empty position p_0 in the CBF (SBF) when the results of hash functions can coincide, is:

$$p_0 = (1 - (1/m))^{kn}. \tag{10}$$

In fact, (9) differs from the standard expression (10) for p_0. However, both produce different results only for small lengths of the filter ($m < 50$), which are not of practical importance.

Because each position in **CBF** can be treated as an independent realization of I, the expected number of positions l with value v equals:

$$l(v) = m \Pr(I = v) = m \binom{n}{v} p_1^v (1 - p_1)^{n-v}. \tag{11}$$

3.4 Definition of autoscaling Bloom filter

Given a CBF, the derived ABF is formed by setting to zero all positions with values less than or equal to the chosen binarization threshold Θ; positions with values greater than Θ are set to one:

$$\mathbf{ABF} = [\mathbf{CBF} > \Theta]. \tag{12}$$

Note that when $\Theta = 0$, the ABF is equivalent to the SBF.

In general, the expected dot product (denoted $\bar{d}_x$) between the ABF and an element x included in the filter is less than or equal to k.[2] As the binarization threshold Θ increases, more of the nonzero positions in the CBF are mapped to zero values in the corresponding ABF. This necessarily reduces the dot product of the ABF vector with the query vector. Therefore, there is a need for the second parameter of the ABF, which determines the lowest value of dot product indicating the presence of an element in the filter. Denote this decision threshold parameter as T ($0 \le T \le k$), then an element of the universe q is judged to be a member of the ABF if and only if the dot product between **ABF** and $\mathbf{q}$ is greater than or equal to T.

3.5 Probabilistic characterization of the autoscaling Bloom filter

When the binarization threshold Θ for the ABF is more than zero, the probability of an empty position in the ABF (denoted as P_0) is higher than in the SBF because some of the nonzero positions in the CBF are set to zero. For a given Θ, the expected P_0 is calculated using (8) as follows:

$$P_0 = \sum_{v=0}^{\Theta} \Pr(I = v) = \sum_{v=0}^{\Theta} \binom{n}{v} p_1^v (1 - p_1)^{n-v}. \tag{13}$$

Then, the probability of 1 in the ABF (denoted as P_1) is:

$$P_1 = 1 - P_0 = 1 - \sum_{v=0}^{\Theta} \binom{n}{v} p_1^v (1 - p_1)^{n-v}. \tag{14}$$

[2] It should be noted that the calculation of expected similarity (e.g., dot product) between two vectors, one of which may store the other, is a general problem formulation in hyperdimensional computing and can be seen as the "detection" type of retrieval (see [33] for details).

The expected dot product $\bar{d}_x$ for an element x included in the ABF is calculated as:

$$\bar{d}_x = k - \frac{m}{n}\sum_{v=0}^{\Theta} v \Pr(I = v). \tag{15}$$

Note that when $\Theta = 0$, $\bar{d}_x(\mathbf{ABF}, \mathbf{x}) = k$ which corresponds to the SBF [see (4)]. In other words, the SBF can be seen as a special case of the ABF. The calculations in (15) when $\Theta > 0$ can be interpreted in the following way. The dot product between **SBF** and **x** is k. A position in **CBF** with value $v > 0$ contributes 1 to the values of dot products of v stored elements. Thus, if this position is set to zero in the SBF, there will be v elements with the dot product equal to $k - 1$ while the dot products for the rest of the elements still equal k. Then, the expected dot product between the filter and an element is decremented by v / n. In fact, the number of positions with value v is unknown, but it is possible to calculate the probability $\Pr(I = v)$ of such position in **CBF** using (8). Then the expected number of such positions in **CBF** is determined via (11) and equals $m\Pr(I = v)$. When the ABF suppresses all such positions, each of them decrements the expected dot product by v / n. Then, the total decrement of the expected dot product by the suppressed positions with value v is expected to be $mv\Pr(I = v)/n$. Because the ABF suppresses all positions with values less than or equal to Θ, the decrements of the expected dot product introduced by each value v should be summed up.

The expected dot product (denoted $\bar{d}_y$) between the ABF and an element y which is not included in the filter is determined by the number of nonzero positions in the filter and calculated as:

$$\bar{d}_y = kP_1. \tag{16}$$

Both dot products d_x and d_y are characterized by discrete random variables (denoted as X and Y, respectively) which in turn are described by binomial distributions: $X \sim \mathrm{B}(k, p_x)$ and $Y \sim \mathrm{B}(k, p_y)$.

The success probabilities (p_x and p_y) of these distributions are determined from the expected values of dot product as in (15) and (16):

$$p_x = \bar{d}_x/k = 1 - \frac{m}{nk}\sum_{v=0}^{\Theta} v \Pr(I = v), \tag{17}$$

$$p_y = \bar{d}_y/k = P_1. \tag{18}$$

3.6 Performance properties of ABF

Given the decision threshold T, the true positive rate (TPR) of the ABF can be calculated using the probability mass function of X as:

$$\mathrm{TPR} = \sum_{d=T}^{k} \Pr(X = d) = \sum_{d=T}^{k} \binom{k}{d} p_x^d (1 - p_x)^{k-d}. \tag{19}$$

Similarly, the false positive rate (FPR) is calculated using the probability mass function of Y as:

$$\mathrm{FPR} = \sum_{d=T}^{k} \Pr(Y = d) = \sum_{d=T}^{k} \binom{k}{d} p_y^d (1 - p_y)^{k-d}. \tag{20}$$

4 Evaluation of ABF

4.1 Optimization of ABF's parameters

In order to choose the best value of T (or even both Θ and T), an optimization criterion is needed. It is proposed to optimize the accuracy (ACC) of the filter. This is defined as the average value of true positive rate and true negative rate: $\mathrm{ACC} = (\mathrm{TPR} + (1 - \mathrm{FPR}))/2$. Note that this definition of accuracy is also known as unweighted average recall. Note also that the accuracy does not have to be the only choice for the optimization criterion. The choice of ACC implies that false positives and false negatives are treated as equally costly. However, in a practical application this may not be true. Instead, each of the four possible outcomes (true positive, false positive, true negative, false negative) will have an associated domain-dependent cost. The designer would then optimize the design parameters so as to minimize the cost in the application scenario. For example, if the total number of elements and the number of elements stored in the filter are known, then such performance metrics as F1 score and Matthews correlation coefficient [34] can be used for optimization. In the absence of a specific application, we are forced to use a general performance summary. We have chosen to use accuracy as a general summary because it is simple and well understood.

In addition, an application may specify the lowest acceptable TPR (denoted as L_{TPR}). Then, the optimal value of T (for fixed Θ) is found as:

$$T_{opt} = \max_{T}(\mathrm{ACC}|\mathrm{TPR} \geq L_{\mathrm{TPR}}). \tag{21}$$

In general, both parameters of the ABF, Θ and T, can be optimized as:

$$\max_{\Theta,T}(\mathrm{ACC}|\mathrm{TPR} \geq L_{\mathrm{TPR}}). \tag{22}$$

4.2 An example: ABF in action

The behavior of ABF for different Θ is illustrated in Figure 2. The length of the CBF (and all derived ABFs) is

$m = 10{,}000$. It stores $n = 500$ unique elements, and each element is mapped to an individual BF with $k = 100$ nonzero positions. Note that the value of k in this example is intentionally not optimized for the given m and n. The particular value of k is chosen for demonstration purposes to clearly illustrate the situation when the SBF has a high false positive rate which can be significantly decreased by the ABF. Similar effects can be seen for other values of k, m, and n.

Six ABFs are formed from the CBF using different thresholds in the range $0 \leq \Theta \leq 5$. Each plot in Fig. 2 corresponds to one ABF and depicts probability mass functions for X (circle markers) and Y (diamond markers). where X and Y denote random variables characterizing distributions of dot products for elements stored in the filter (X) and elements not included in the filter (Y).

The plot for $\Theta = 0$ corresponds to the SBF. In this case, X is deterministic and located at $k = 100$ as expected given $k = 100$ nonzero positions for the SBF. Hence, the optimal value of T is trivially equal to k and TPR $= 100\%$. A large portion of the distribution for Y is also concentrated at $k = 100$, which leads to high FPR $= 52\%$. On the other hand, the ABFs with $\Theta > 0$ have better separation of the two distributions. Much lower FPR can be achieved by reducing the TPR below 100%. The optimal values of T (indicated by black vertical bars) were found for each value of Θ according to (21). The lowest acceptable value of TPR, L_{TPR} was set to 0.97. This particular value was chosen to demonstrate that, in principle, a large reduction of the FPR can be achieved via a small reduction in the TPR. The best values of TPR, FPR, and ACC for each plot are depicted in the figure. For example, even changing Θ from 0 to 1 allows FPR to be reduced from 0.52 to 0.24 at the cost of reducing TPR by only 3%. Overall, the accuracy is improved by 0.13. The best performance among the considered range is achieved for $\Theta = 4$, resulting in TPR $= 0.98$, FPR $= 0.04$, ACC $= 0.97$, thus improving the accuracy of the SBF by 31%. It should be noted that the presented example considered only a narrow range of Θ. In principle, Θ could be chosen between 0 and n, and therefore, it is important to observe the performance of the ABF for larger Θ. Figure 3 demonstrates the dependency between Θ and ACC, where for each Θ in the range $0 \leq \Theta \leq 20$, T was optimized according to (21) without limiting L_{TPR}. The first six values of ACC in Figure 3 correspond to the values depicted in Figure 2. These values lie in the region where the ACC was increasing for each new value of Θ. However, for values of $\Theta > 5$, we observe that ACC is constantly decreasing until it reaches 0.5. This decrease happens because with the increased Θ the sparsity of the ABF is increasing until all positions in the filter are set to zero. This moment corresponds to ACC $= 0.5$ because an empty filter has no information about the stored elements, and thus, its TPR is zero, but it also has no false positives (i.e., FPR $= 0$), which results in ACC $= 0.5$. Therefore, we observed that the dependency between Θ and ACC is nonlinear and that there is a peak value of ACC, which in the considered example was achieved for $\Theta = 4$.

4.3 Comparison with the optimized BF

Figure 4 demonstrates the results of comparison of four filters: the autoscaling BF (dash-dot line), the optimized BF

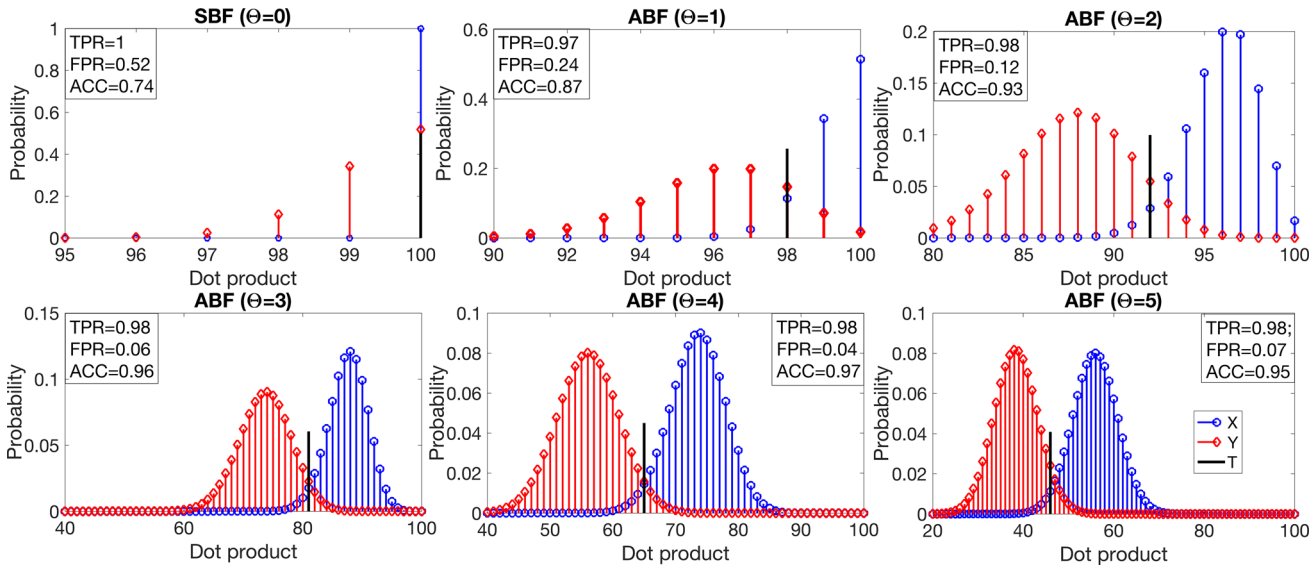

Fig. 2 Probability mass functions for X (query present) and Y (query absent) for different thresholds Θ in the range $0 \leq \Theta \leq 5$; $k = 100$, $n = 500$, $m = 10{,}000$

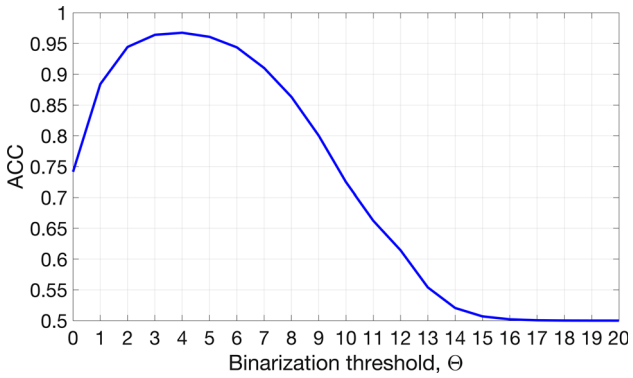

Fig. 3 Comparison of the highest possible accuracy (ACC) for different thresholds Θ in the range $0 \le \Theta \le 20$; $k = 100$, $n = 500$, $m = 10{,}000$

(solid line), the nonoptimized BF (dashed line), and the nonoptimized RBF (dotted line). The nonoptimized RBF was created via randomly erasing 0.1% of nonzero positions in the nonoptimized BF. The nonoptimized BF shows the performance of the CBF (SBF) without using the ABF, and thus, it shows a fair comparison of the proposed approach and the standard approach. The nonoptimized RBF is chosen for comparison, as it is conceptually the most relevant modification of the SBF to the ABF, and thus, it shows an alternative, also decreasing FPR by introducing some false negatives. Finally, the optimized BF demonstrates the best possible performance achievable by the CBF (SBF). Each panel in Fig. 4 corresponds to a performance metric: left—TPR; center—FPR; right—ACC. Please recall that ACC is not the only possible metric cumulatively characterizing TPR and FPR, however, it was adopted in this paper as the optimization criterion for the ABF. Please see the discussion in Sect. 4.1 for the motivation of that choice and possible alternatives. The performance was studied for a range of numbers of unique elements stored in the filter ($50 \le n \le 5000$). The length of the filters was the same as in Fig. 2, $m = 10{,}000$. For the optimized BF, k was calculated as in (3) for each value of n and varied between 1 and 139. For three other BFs, k was fixed to 100. The values used in the experiments are summarized in Table 1. The ABF was formed from the CBF according to (12). Only two parameters (Θ and T) of ABF were optimized for each value of n according to (22) with $L_{\mathrm{TPR}} = 0.9$. Note that these two parameters do not change the hardware resources required for an ABF implementation since k and m are fixed, while an optimized BF implementation might require 40% more hash functions. This overhead directly translates to a larger silicon area or slower speed for the hardware implementation of the optimized BF compared to the ABF.

The TPR of the optimized and nonoptimized BFs is always 1, while for the ABF and nonoptimized RBF it can be less. In particular, the TPR of the ABF varies in the allowed range between L_{TPR} and 1. For large values of n (>1000), the TPR of the ABF is approximately equal to L_{TPR}. In the case of nonoptimized RBF the TPR was around 0.9 over the whole range of n. The FPR of all the filters grows with increasing n. As anticipated, the nonoptimized BF soon (at $n \approx 1000$) achieves FPR = 1 and stays there until the end. A similar behavior is demonstrated by the nonoptimized RBF with the exception that the highest value of FPR is 0.9. Note that with RBF, the price one has to pay for the lower FPR is the decreased TPR. Two other filters, the ABF and the optimized BF, demonstrate a smooth increase in FPR. The FPR is lower than 1 for both filters even when $n = 5000$ (approximately 0.6 and 0.4, respectively). The accuracy curves aggregate the behavior for TPR and FPR. For most values of n, the nonoptimized BF and RBF reach ACC = 0.5 as their FPRs reach the maximal values. Their accuracies for large values of n are the same because the gain in FPR equals the loss in TPR for the nonoptimized RBF. The accuracies of the ABF and the optimized BF smoothly decay with the growth of n, being 0.66 and 0.8 when $n = 5000$. Thus, the ABF significantly outperforms the nonoptimized BF and RBF when

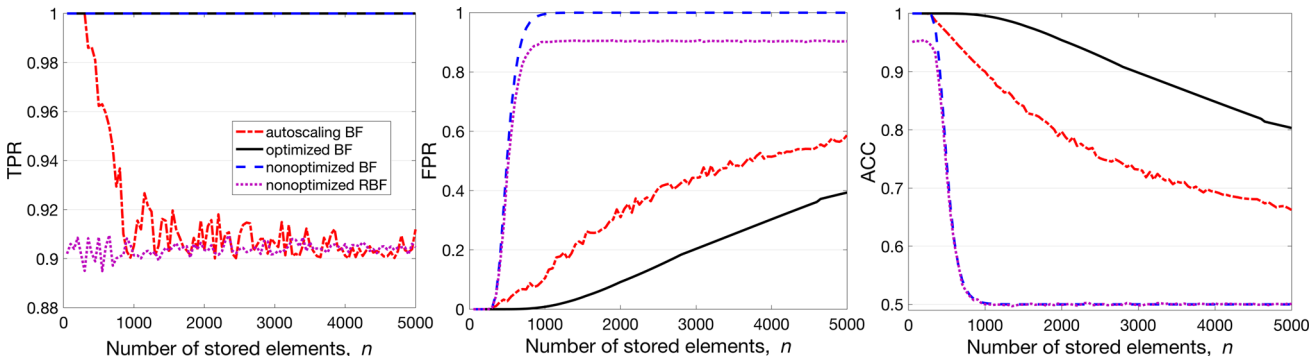

Fig. 4 Comparison of performance (TPR, FPR, and ACC) of four different BFs against varying number of stored elements n ($50 \le n \le 5000$ step 50)

Table 1 Summary of the experimental setup

Parameter	RBF	ABF	Nonoptimized BF	Optimized BF
Number of hash functions, k	100			Calculated for n using (3) ranged between 1 and 139
Number of stored elements, n	$50 \leq n \leq 5000$ step 50			
Filter length, m	10,000			
Number of times rebuilt	1			23

their FPRs are increasing. In general, the performance of the ABF follows that of the optimized BF with some constant loss. The increase in accuracy from ABF to optimized BF can be understood as the value delivered by being able to specify in advance precisely the number of elements to be stored in the filter. The best trade-off between TPR and FPR is in the region of n where FPR of the nonoptimized BF is steeply increasing from 0 to 1.

It is important to reemphasize the advantages of the ABF over the optimized BF. In the experiments above, the ABF addressed the major difficulty of the SBF, which is its limited scalability, since the ABF does not require the recalculation of the whole filter as the number of the stored elements is increasing. Thus, the ABF allows adopting the performance of the filter even when the number of elements to be stored simultaneously is not known in advance. On the contrary, the SBF (i.e., the optimized BF in the experiments) is not scalable as it must be rebuilt if a new value of k is chosen. In the experiments reported in Fig. 4, k varied between 1 and 139 and the optimized BF was rebuilt 23 times (cf. Table 1). The fact that the optimized BF has to be rebuilt every time when k changes limits its use-cases for situations with dynamic ranges of elements such as in Fig. 4. Another very important advantage of the ABF is that due to its adaptiveness, the number of hash functions k can be fixed for a wide range of stored elements. Fixed k allows significantly simplifying hardware implementations since there would be no need to account for increased area and power of a chip [5] when k grows. Obviously, since the optimized BF has to work in a dynamic range of k, it does not have this advantage.

5 Conclusion

This paper introduced the autoscaling Bloom filter. The ABF is a generalization of the standard binary BF, derived from the counting BF, with procedures for achieving probabilistic bounds on false positives and true positives. It was shown that the ABF can significantly decrease the false positive rate at a cost of allowing a nonzero false negative rate. The evaluation revealed that the accuracy of the ABF follows the standard BF with the optimized number of hash functions with some constant loss. As opposed to the optimized BF, the ABF provides means for optimization of the filter's performance without requiring the entire filter to be rebuilt when the number of stored elements in the filter is changing dynamically. This optimization can be achieved while the number of hash functions remains fixed.

There are several limitations to this study. First, since the paper focused on presenting and characterizing the algorithm rather than a solution to any problem, no particular attention has been paid to study the effect of an optimization criterion on the ABF's performance. Instead, we simply adopted the accuracy. Second, the analysis of the ABF presented in this paper used counting BFs with the unlimited range of counters. In practice, however, the size of counters is limited to several bits [35]. In the future work, we will focus on analyzing the effect of restricted counters in counting BFs on the ABF.

Acknowledgements Open access funding provided by Lulea University of Technology. This work was supported by the Swedish Research Council under Grant 2015-04677.

Compliance with ethical standards

Conflict of Interest The authors declare that they have no conflict of interest.

References

1. Gritsenko V, Rachkovskij D, Frolov A, Gayler R, Kleyko D, Osipov E (2017) Neural distributed autoassociative memories: a survey. Cybern Comput Eng 2(188):5–35
2. Touretzky D, Hinton G (1988) A distributed connectionist production system. Cognit Sci 12(3):423–466

3. Fan L, Cao P, Almeida J, Broder A (2000) Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans Netw 8(3):281–293
4. Donnet B, Baynat B, Friedman T (2006) Retouched Bloom filters: allowing networked applications to trade off selected false positives against false negatives. In: ACM CoNEXT conference, pp 1–12
5. Akhlaghi V, Rahimi A, Gupta RK (2016) Resistive Bloom filters: from approximate membership to approximate computing with bounded errors. In: Conference on Design, Automation and Test in Europe (DATE), pp 1–4
6. Rachkovskij DA (2001) Representation and processing of structures with binary sparse distributed codes. IEEE Trans Knowl Data Eng 3(2):261–276
7. Kanerva P (2009) Hyperdimensional computing: an introduction to computing in distributed representation with high-dimensional random vectors. Cognit Comput 1(2):139–159
8. Grandi F (2018) On the analysis of Bloom filters. Inf Process Lett 129:35–39
9. Tarkoma S, Rothenberg CE, Lagerspetz E (2012) Theory and practice of Bloom filters for distributed systems. IEEE Commun Surv Tutor 14(1):131–155
10. Broder A, Mitzenmacher M (2004) Network applications of Bloom filters: a survey. Internet Math 1(4):485–509
11. Kazemi SMR, Bidgoli BM, Shamshirband S, Karimi SM, Ghorbani MA, Chau KW, Pour RK (2018) Novel genetic-based negative correlation learning for estimating soil temperature. Eng Appl Comput Fluid Mech 12(1):506–516
12. Wu CL, Chau KW (2011) Rainfall-runoff modeling using artificial neural network coupled with singular spectrum analysis. J Hydrol 399:394–409
13. Rabieh K, Mahmoud M, Akkaya K, Tonyali S (2017) Scalable certificate revocation schemes for smart grid AMI networks using Bloom filters. IEEE Trans Dependable Secure Comput 14(4):420–432
14. Ma H, Tseng YC, Chen LI (2016) A CMAC-based scheme for determining membership with classification of text strings. Neural Comput Appl 27(7):1959–1967
15. Anbar M, Abdullah R, Munther A, Al-Betar MA, Saad RMA (2017) NADTW: new approach for detecting TCP worm. Neural Comput Appl 28(1):525–538
16. Lu J, Wan Y, Li Y, Zhang C, Dai H, Wang Y, Zhang G, Liu B (2017) Ultra-fast Bloom filters using SIMD techniques. In: 2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS), pp 1–6
17. Zhang Y, Zheng Z, Zhang X (2017) Efficient Bloom filter for network protocols using AES instruction set. IET Commun 11(11):1815–1821
18. Pontarelli S, Reviriego P, Mitzenmacher M (2014) Improving the performance of invertible bloom lookup tables. Inf Process Lett 114(4):185–191
19. Bender M, Farach-Colton M, Johnson R, Kraner R, Kuszmaul B, Medjedovic D, Montes P, Shetty P, Spillane RP, Zadok E (2012) Don't thrash: how to cache your hash on flash. Proc VLDB Endow 5(11):1627–1637
20. Pandey P, Bender M, Johnson R, Patro R (2017) A general-purpose counting filter: making every bit count. In: SIGMOD'17 Proceedings of the 2017 ACM international conference on management of data, pp 775–787
21. Einziger G, Friedman R (2017) TinySet—an access efficient self adjusting Bloom filter construction. IEEE/ACM Trans Netw 25(4):2295–2307
22. Fan B, Andersen D, Kaminsky M, Mitzenmacher M (2014) Cuckoo filter: practically better than bloom. In: CoNEXT'14 Proceedings of the 10th ACM international on conference on emerging networking experiments and technologies, pp 75–88
23. Lim H, Lee J, Byun H, Yim C (2017) Ternary Bloom filter replacing counting Bloom filter. IEEE Commun Lett 21(2):278–281
24. Rothenberg CE, Macapuna CAB, Verdi FL, Magalhaes MF (2010) The deletable Bloom filter: a new member of the Bloom family. IEEE Commun Lett 14(6):557–559
25. Lim H, Lee J, Yim C (2015) Complement Bloom filter for identifying true positiveness of a Bloom Filter. IEEE Commun Lett 19(11):1905–1908
26. Carrea L, Vernitski A, Reed M (2016) Yes-no Bloom filter: a way of representing sets with fewer false positives. ArXiv 160301060:1–28
27. Pontarelli S, Reviriego P, Maestro J (2016) Improving counting Bloom filter performance with fingerprints. Inf Process Lett 116(4):304–309
28. Lim H, Lee N, Lee J, Yim C (2014) Reducing false positives of a Bloom filter using cross-checking Bloom filters. Appl Math Inf Sci 8(4):1865–1877
29. Almeida P, Baquero C, Preguica N, DHutchison (2007) Scalable Bloom filters. Inf Process Lett 101(6):255–261
30. Rottenstreich O, Kanizo Y, Keslassy I (2014) The variable-increment counting Bloom filter. IEEE/ACM Trans Netw 22(4):1092–1105
31. Bose P, Guo H, Kranakis E, Maheshwari A, Morin P, Morrison J, Smid M, Tang Y (2008) On the false-positive rate of Bloom filters. Inf Process Lett 108(4):210–213
32. Christensen K, Roginsky A, Jimeno M (2010) A new analysis of the false positive rate of a Bloom filter. Inf Process Lett 110(21):944–949
33. Frady EP, Kleyko D, Sommer FT (2018) A theory of sequence indexing and working memory in recurrent neural networks. Neural Comput 30:1449–1513
34. Fawcett T (2006) An introduction to ROC analysis. Pattern Recognit Lett 27:861–874
35. Bonomi F, Mitzenmacher M, Panigrahy R, Singh S, Varghese G (2006) An improved construction for counting Bloom filters. In: 14th Annual European Symposium on Algorithms, LNCS 4168, pp 684–695