Information Processing Letters 116 (2016) 304–309

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Improving counting Bloom filter performance with fingerprints

Salvatore Pontarelli [a,*,1], Pedro Reviriego [b,2], Juan Antonio Maestro [b,2]

[a] Consorzio Nazionale Interuniversitario per le Telecomunicazioni, CNIT, Via del Politecnico 1, 00133 Rome, Italy
[b] Universidad Antonio de Nebrija, C/ Pirineos, 55, E-28040 Madrid, Spain

ARTICLE INFO

Article history:
Received 23 September 2014
Received in revised form 4 November 2015
Accepted 4 November 2015
Available online 11 November 2015
Communicated by M. Chrobak

Keywords:
Algorithms
Bloom filters
Data structures

ABSTRACT

Bloom filters (BFs) are used in many applications for approximate check of set membership. Counting Bloom filters (CBFs) are an extension of BFs that enable the deletion of entries at the cost of additional storage requirements. Several alternatives to CBFs can be used to reduce the storage overhead. For example schemes based on d-left hashing or Cuckoo hashing have been proposed. Recently, also a new type of CBF, the Variable Increment Counting Bloom Filter (VI-CBF) has been introduced to improve performance. The VI-CBF uses different increments in the filter counters to reduce the false positive rate and therefore the storage requirements. In this paper, another mechanism to improve CBF performance: the Fingerprint Counting Bloom Filter (FP-CBF) is presented. The proposed scheme is based on the use of fingerprints on the filter entries to reduce the false positive rate. This results in a simpler implementation than VI-CBFs in terms of number of hash functions and arithmetic operations. The false positive rate of the proposed scheme has also been analyzed theoretically and by simulation and compared with the VI-CBF. The results show that the proposed scheme can achieve lower false positive rates than those of a simple VI-CBF implementation. When compared with a better and more complex VI-CBF implementation, the FP-CBF outperforms it when the number of bits per element is large while the VI-CBF is better for low number of bits per element.

1. Introduction

Introduced by Burton Bloom more than forty years ago, Bloom Filters [1] have found numerous applications in computing and networking [2–4]. Bloom filters (BF) are a simple data structure that provides check set membership with a small probability of false positives. A BF uses an array of m bits and k hash functions. To add an element x, the positions given by the k hash functions $h_1(x), h_2(x), \ldots, h_k(x)$ are set to one. A query for a given element is successful when the positions given by the k hash functions are all ones. If the element has been previously added to the BF, the query will always be successful. For elements that have not been added to the BF, in most cases, the query will fail. However, there is a small probability that the query is successful producing a false positive. False positives can be reduced by increasing the size of the array and thus the storage requirements. A limitation of BFs is that elements cannot be removed as a position in the array may have been set to one by different elements. To overcome this issue, Counting Bloom filters (CBF) were introduced in [5]. In CBFs, the array instead of single bits is composed of several bits and each position is used as a counter. When an entry x is added the counters with positions $h_1(x), h_2(x), \ldots, h_k(x)$ are incremented. Conversely, to delete an entry those counters

* Corresponding author.
E-mail addresses: pontarelli@ing.uniroma2.it (S. Pontarelli), previrie@nebrija.es (P. Reviriego), jmaestro@nebrija.es (J.A. Maestro).
[1] Tel.: +39 0672597369.
[2] Tel.: +34 914521100; fax: +34 914521110.

http://dx.doi.org/10.1016/j.ipl.2015.11.002
0020-0190/

are decremented. A query is successful when all the counters with positions $h_1(x), h_2(x), \ldots, h_k(x)$ are larger than zero. The use of counters increases the storage requirements of CBFs compared to BFs. For example, if c bits are used for the counters, a CBF requires c^*m bits. In most practical configurations a value of $c = 3$ or 4 bits is enough to ensure a low probability of counter overflow [5,6]. Even if BFs and CBFs have been adopted in many applications, it is worth to note that in some cases, it may be better not to use a BF [7]. This occurs when the false positive rate is not smaller than the probability that the element is actually in the set.

Many optimizations of BFs have been proposed for example to efficiently deal with dynamic datasets [8] or to determine the optimal parameter settings for some applications [9]. Several enhancements to CBFs have also been proposed focusing on reducing the storage requirements and the false positive rate. For example in [10] the use of variable length counters was proposed. In [11], a coding for the counters that reduces the number of bits required was presented while in [12,13] multiple level implementations for the BFs were considered. In [14], an alternative construction of CBFs based on d-left hashing was proposed to reduce the storage requirements. More recently in [15] the use of Cuckoo hashing has been proposed to efficiently implement CBF functionality.

In [16] a simple enhancement of the CBF, the Variable Increment Counting Bloom Filter (VI-CBF) was introduced. In a VI-CBF, additional hash functions $(g_1(x), g_2(x), \ldots, g_k(x))$ are used to determine the increments i to be added to the CBF counters when adding an entry. The same values are used to decrement the counters when an entry is removed. The increments are designed in such a way that the counter values can be used to reduce the false positive rate. This can be done in several ways. The simplest implementation is to take the increments from the set $D_L = \{L, L+1, \ldots, 2^*L-1\}$. Therefore a counter with a value equal to or smaller than 2^*L-1 has been set by a single element. When querying for an element, the additional hashes can be used to compare against the stored counters when they have a value equal to or smaller than 2^*L-1. If the values are different, the query fails. This occurs with a probability of $(L-1)/L$. This provides significant reductions in the false positive rate as a large percentage of positions in the array have been set by a single element for practical false positive rates. In fact, the distribution of the number of elements mapped to a given position can be approximated by a Poisson distribution that when the number of elements is equal to or smaller than the number of positions has its largest value (not considering zero) at one. This implementation of the VI-CBF is also capable of detecting in some cases when a counter has been set by two elements and use that information to also reduce the probability of false positives. A more general implementation of the VI-CBF uses increments from a set D formed by any given set of integers. This implementation is more complex but can reduce the false positive rate. For example in [16] the set $D = \{8, 12, 14, 15\}$ was shown to provide good results. In the rest of the paper, both implementations of the VI-CBF will be considered and denoted as D_L and D respectively. Although more bits are required to avoid the overflow of the counters, the VI-CBF has been shown to reduce the false positive rate and the storage requirements compared to those of a traditional CBF. That is 1) for a given storage size in bits, the VI-CBF provides a lower false positive rate and 2) to achieve a given false positive rate, the VI-CBF requires less storage than a traditional CBF. The main overhead of the VI-CBF is the need for additional hash functions and the arithmetic operations needed for addition, removal and query operations.

In this paper, the Fingerprint Counting Bloom filter (FP-CBF) is introduced. The FP-CBF adds fingerprints to the elements stored in the CBF to reduce the probability of false positives. The use of fingerprints in combination with variable increments was suggested in [16] to improve the performance of alternative constructions of BFs [14]. Here, our goal is different, the fingerprints are used on a traditional CBF implementation to replace the variable increments. A key contribution is that to enable addition/removal of several items, the fingerprints are updated using *xor* operations. This means that there is no limit on the number of fingerprints that can be placed on a given position (as opposed to hash based constructions of BFs). Another key contribution is to note that during query operations, the fingerprints can be used to reduce the probability of false positives for positions that have a counter value of one. This is linked to the observation that for low false positive rate many positions will have a counter value of one. The FP-CBF puts all those ideas together to achieve a performance comparable to that of VI-CBFs with a simpler implementation. The false positive rate of FP-CBFs has been evaluated both analytically and by simulation. The results show that it outperforms the D_L VI-CBF implementation for a wide range of parameters. The FP-CBF also outperforms the D VI-CBF implementation when the number of bits per element is large. Conversely, the D VI-CBF implementation provides lower false positive rates when the number of bits per element is small. In terms of implementation complexity, only one additional hash function $h_{fp}(x)$ is needed to compute the fingerprint compared to k hash functions in the VI-CBF. Additionally, the FP-CBF does not require arithmetic operations. Therefore its implementation is also simpler than that of VI-CBF.

The rest of the paper is organized as follows. Section 2 presents the Fingerprint Counting Bloom Filter (FP-CBF) and discusses its operation. In Section 3, the performance of the FP-CBF is analyzed theoretically providing approximations for the false positive rate. Then in Section 4, the scheme is evaluated by simulation and compared with the VI-CBF. Finally, the conclusions are presented in Section 5.

2. The fingerprint counting Bloom filter

The proposed scheme uses an array of m positions each of which has two fields, a counter with c bits and a fingerprint with f bits. Therefore the FP-CBF requires $(c+f)^*m$ bits. To add/remove and query for elements, $k+1$ hash functions are needed. The first k functions $(h_1(x), h_2(x), \ldots, h_k(x))$ are those used in a traditional CBF and map an element to one of the m positions. The