RESEARCH ARTICLE

Suitability of a new Bloom filter for numerical vectors with high dimensions

Chunyan Shuai[1], **Jiayou Lei**[1], **Zeweiyi Gong**[2], **Xin Ouyang**[3]*

1 Faculty of Transportation Engineering, Kunming University of Science and Technology, Kunming, China,
2 Faculty of Electric Power Engineering, Kunming University of Science and Technology, Kunming, China,
3 Faculty of Information Engineering and Automation, Kunming University of Science and Technology, Kunming, China

* kmoyx@hotmail.com

OPEN ACCESS

Citation: Shuai C, Lei J, Gong Z, Ouyang X (2018) Suitability of a new Bloom filter for numerical vectors with high dimensions. PLoS ONE 13(12): e0209159. https://doi.org/10.1371/journal.pone.0209159

Editor: Zhan Li, University of Electronic Science and Technology of China, CHINA

Received: July 13, 2018

Accepted: December 2, 2018

Published: December 21, 2018

Data Availability Statement: Data are available freely from http://corpus-texmex.irisa.fr/ and on figshare at the URLs: https://figshare.com/articles/sift_data/7428974 and https://figshare.com/articles/ivecs_read_m/7428971.

Funding: This work is supported by the National Key R&D Plan of China (No. 2017YFB0306405) to XO, the National Natural Science Foundation of China (No.61562056, No.61364008) to CS. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Abstract

The notable increase in the size and dimensions of data have presented challenges for data storage and retrieval. The Bloom filter and its generations, due to efficient space overheads and constant query delays, have been broadly applied to querying memberships of a big data set. However, the Bloom filter and most of the variants regard each element as a 1-dimensional string and adopt multiple different string hashes to project the data. The interesting problem is when the inputs are numerical vectors with high dimensions, it remains unknown whether they can be projected into the Bloom filter in their original format. Furthermore, we investigate whether the projection is random and uniform. To address these problems, this paper presents a new uniform Prime-HD-BKDERhash family and a new Bloom filter (P-HDBF) to retrieve the membership of a big data set with the numerical high dimensions. Since the randomness and uniformity of data mapping determines the performance of the Bloom filter, to verify these properties, we first introduce information entropy. Our theoretical and experimental results show that the P-HDBF can randomly and uniformly map the data in their native formats. Moreover, the P-HDBF provides an efficient solution alternative to implement membership search with space-time overheads. This advantage may be suitable for engineering applications that are resource-constrained or identification of the nuances of the graphics and images.

1. Introduction

With increasing data sizes, concise data representations and efficient query algorithms have become the key factors to large-scale data management. As a result, a large number of technologies have appeared, such as the Bloom filter (BF) [1]. The BF has a low query delay and a high time-space overhead, leading to its broad use in computing areas, such as network and network security [2–5], distributed systems [6–9] and applications or embedded devices [10,11], with limited computing and storage resources. Moreover, many variants have been proposed, including the counting Bloom filter (CBF) [12] and its improvements [13–14], the compressed Bloom filter[15], the spectral Bloom filter[16], the dynamic Bloom filter [17], the Cuckoo Filter[18], and the parallel BFs (PBF-HT and PBF-BF) [19, 20].

Competing interests: The authors have declared that no competing interests exist.

Abbreviations: $S = (V_1, \ldots, V_n)$, A set of n vectors for member query; $V(v_1 \ldots v_d)$, $q(q_1 \ldots q_d)$, A vector in S and a query q with d dimensions; $P(p_1 \ldots p_l)$, A prime number set P and a prime number p_i; H^p, A Prime_HD_ BKDRHash function family with a function of $h_k^P = S_i = \prod p_{i-1} \cdot S_{i-1} + v_i (i = 1 \ldots d)$; f_{P-HDBF}, fnp_{P-HDBF}, The false positive (negative) probability of P-HDBF.; m, The array size of P-HDBF.; k, The number of Prime_HD_ BKDRHash functions.

The BF can perform well and obtain a low false positive probability (FPP) only when the hash randomly and uniformly disperses the data, and usually, string hash functions [21] are the default choices. Regardless of the data format, the string hash takes the input as a 1-dimensional string, rather than its original format, and iteratively computes every character to obtain a random integer. To better scatter the data into different places and reduce the FPP, multiple different string hashes are usually selected. To project numerical vectors with high dimensions in their original formats, LshBFs [22–25] replace the string hashes with a uniform locality sensitive hashing (LSH) [26]. However, since the LSH gathers the data around the mean, LshBFs are more suitable for approximate nearest neighbours queries, rather than membership queries.

When the inputs are numerical vectors with high dimensions, this paper proposes dealing with them in their original formats other than strings. First, a unified prime BKDERhash [27] function family, denoted as Prime-HD-BKDERhash, is proposed to substitute for multiple different string hashes. Meanwhile, information entropy is introduced in the BF to verify the randomness and uniformity of the data mapped by the Prime-HD-BKDERhash. Next, by combining the unified Prime-HD-BKDERhash with a counter array, a new BF called P-HDBF is established to store and retrieve the memberships of the big data set. The theoretical analysis and experiments show that the Prime-HD-BKDERhash can disperse elements more effectively than the string hashes, and the P-HDBF is more suitable to represent and query the numerical vectors of a big data set in high-dimensional spaces, which has low space-time costs. Compared with the PBF-HT and PBF-BF, the P-HDBF possesses low false detection rates, low query delays and low space requirements. The advantages of the constant query delay and low space-time costs make the P-HDBF more appropriate for some engineering applications with constrained computing and storage resources, such as distinguish the nuances of the graphics and images.

The remainder of this paper is organized as follows. Related works are described in section 2. The design of Bloom filter and our structure are presented in section 3. The theoretical analyses and proofs are in Sections 4 and 5. Section 6 presents the related performance evaluation and experiments. Section 7 presents the study's conclusions.

2. Related work

This section provides a brief survey related to the Bloom filter designs and its variants that are suitable for element deletion and multi-dimensional vectors.

A Bloom filter [1] utilities a slightly array to store a big data set. This filter uses the mappings of multiple string hashes to answer whether a query is member of the set with a small false positive probability or not. To support element deletion, the counting Bloom filter (CBF) [12] proves that a 4-bit counter array will be sufficient to defend against overflows brought by element deletion. The FPP, array size and cardinality of the BF have been discussed in [28–30]. The variable incremental counting Bloom filter (VI-CBF) [31] increases the counter by a variable increment rather than the unaltered increment to reduce memory costs. Moreover, with the same counter width, the query in VI-CBF can get a more complete answer than in CBF. The Cuckoo filter [18] consists of an array of buckets where each item has two candidate buckets. The filter computes every item's two fingerprints and bucket positions using hash functions $h_1(x) = h(x)$ and $h_2(x) = h_1(x) \oplus h$ (h is x's fingerprint). The lookup procedure checks both buckets to see if either one contains the query to determine the membership. Since the insert procedure will continuously relocate existing fingerprints to their alternatives until no more buckets can be allocated, it efficiently reduces the memory costs but results in a long computational time.

Bloom-1 [32] achieves a reduced query overhead at the cost of a higher FPP for a given memory size. Reviriego [33] provides a correct analysis of Bloom-1 and gives out an exact FPP. For the fixed FPP and cardinality of a dataset, the spaces that a BF required are determined. Once a number of extra elements are added in, the FPP will increase quickly. Therefore, the traditional BF is suitable for static sets. The Spectral BF [16] and Dynamic BF (DBF) [17] extend the BF to multi-set and dynamic sets, respectively. To determine which BF an element belongs to in cloud environment, Bloofi [34] organizes different BFs in a hierarchical index structure similar to a B+ tree and the FPP of the hierarchical Bloofi is discussed in [35].

These BFs recognize the inputs as 1-dimensional strings. PBF [20], PBF-HT and PBF-BF [21] have been developed to store and query multi-dimensional elements. The PBF consists of multiple parallel standard BFs, and each standard BF represents an attribute. Due to the destruction of the integrity of the attributes, the PBF generates a high FPP. Furthermore, to reduce the FPP, the PBF-HT (PBF-BF) adds a hash table (a check BF) to the PBF. Let d be the number of dimensions, let m_1 and m_2 be the sizes of the array of the BF and the HT (or the checkBF), and let k_1 and k_2 be numbers of hash functions of the PBF and the HT (or the check BF), respectively. The memory cost and query delay of the PBF-BF (or PBF-HT) are dm_1+m_2 and k_1d+k_2, respectively. Both of them grow linearly as dimensions increase and result in huge memory wastes and query delays. Rather than applying multiple different string hashes to map the inputs into different integers, the LshBF schemes [22–25] apply locality sensitive hashing (LSH) [26] functions to directly transform high-dimensional vectors into serial real numbers by performing the dot product with the input dimensions and mapping similar vectors in the Euclidean space to near location(s). The LSH avoids "dimensional disasters" but results in a high FPP when querying memberships. To reduce the FPP, the LshBF-BF [23] adds a verification BF to further disperse vectors. According to the central limited theorem [36], the LSH shrinks all elements of the set around the mean. For example, when the LSH satisfies the standard normal distribution, approximately 68.5% of the elements gather between the negative and positive variance after mapping, which makes it more suitable for approximate nearest neighbours search.

3. Methods and structure

3.1 Standard Bloom filter and Counter Bloom filter

Definition 1. Bloom filter (BF) [**1**]. A Bloom filter contains k independent string hash functions $h_j(j = 1,\ldots,k)$ and an array of m bits initiated to 0. By projecting k hashes, the BF stores n elements of a set S $(V_1,V_2\ldots V_n)$ into the bit array. For h_j $(j = 1,\ldots,k)$ and $V_i(i \leq n)$, the bit $h_j(V_i)\%m$ is set to 1. A bit can be set to 1 multiple times, but only the first change has an effect. Given a query q, if $h_j(q)\%m = 1$ for all h_j $(j = 1,\ldots,k)$, the q is accepted as a member of S with a false positive probability (FPP).

The BF assumes that each h_j $(j = 1,\ldots,k)$ can randomly and uniformly map elements. Usually, h_j is a string hash [21], such as sax_hash and RSHash. By repeatedly iterating every character of V_i, h_j obtains an integer in the range of $[0–(2^{31}\text{-}1)]$ (32 bits length) as the random hash fingerprint of V_i. For example, given two vectors X(357,246,369) and Y(468,369,157), the sax_ hash function (h_1) uses ASCII codes of characters '3','5','7',',','2'. . . of X to iteratively compute a random integer. Then, the counter $h_1(X)\%m$ is added with 1, as shown in Fig 1.

3.2 Prime high dimensional Bloom filter

To address numerical vectors with high dimensions in their original formats other than strings, a new uniform hash function family, denoted as Prime_HD_BKDRHash, is proposed.

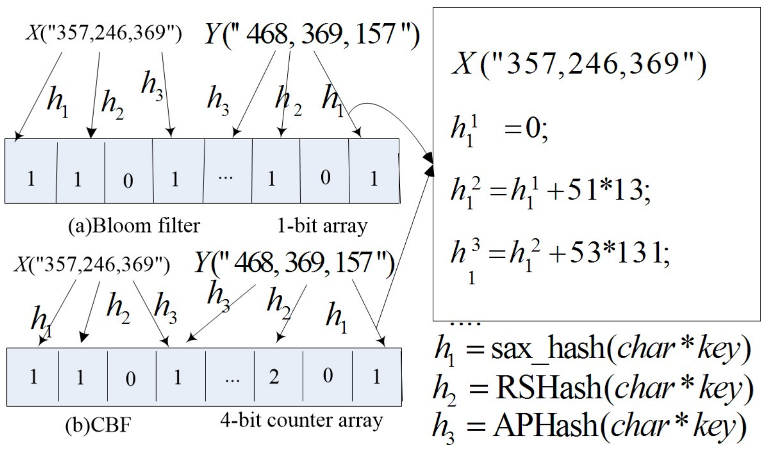


Fig 1. Bloom filter and counting Bloom filter.

https://doi.org/10.1371/journal.pone.0209159.g001

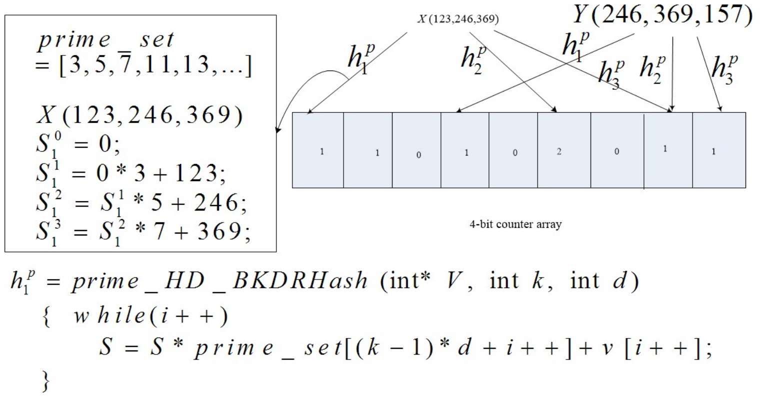
Based on the unified Prime_HD_BKDRHash and a counter array, a new BF called P-HDBF is built, as shown in Fig 2.

(1) Prime_HD_BKDRHash. It originates from the BKDRHash function [27] and prime numbers. Given a prime number set $P = [3,5,7,11,13,17...]$(except of 2) and a d dimensional numerical vector $V(v_1,...,v_d)$, Prime_HD_BKDRHash considers V as a d dimensional vector. By iteratively computing $h^p = S_i = \prod p_{i\text{-}1} \cdot S_{i-1} + v_i$ ($i = 1...d$), d dimensions contribute to the last hash value. Although the *jth* operation and the (j+1)*th* operation are same, the corresponding prime numbers are different. Therefore, $h_j(V)$ and $h_{j+1}(V)$ will get different hash values (details in section 4.1).

(2)A counter array (CA). The array of P-HDBF contains m counters and each counter occupies 4 bits, which is enough to defend against the FNP brought by deleting elements [12]. When k random integers are calculated by k Prime_HD_BKDRHash functions, the counter $h_j(V)\%m(1 \leq j \leq k)$ of the CA is added to 1.

4. Theoretical analysis

The BF structure can work well only when the hashes can randomly and uniformly project all elements, since it is the basis of the BF. Therefore, this section will discuss the hash family- Prime_H-D_BKDRHash which is based on BKDRHash [27], and demonstrate why it is effective



```
prime_set = [3,5,7,11,13,...]
X(123,246,369)
$S_1^0 = 0;$
$S_1^1 = 0*3+123;$
$S_1^2 = S_1^1*5+246;$
$S_1^3 = S_1^2*7+369;$
```

```
$h_1^p$ = prime_HD_BKDRHash (int* V, int k, int d)
  {  while(i++)
        S = S * prime_set[(k-1)*d+i++]+v[i++];
  }
```

Fig 2. P-HDBF and prime_HDBKDRHash.

https://doi.org/10.1371/journal.pone.0209159.g002

in the projection and query of high-dimensional vectors. The definition, proof and algorithm are shown as follows.

4.1 Prime_HD_ BKDRHash

Definition 2. A family $H^p = \{h^p{:}R^d \to U\}$ of functions is called a prime high-dimensional BKDRHash (Prime_HD_BKDRHash), if $\forall V \in R^d, V(v_1,\ldots,v_d)$ and a prime number set $P(p_1,\ldots, p_l)(l \in \infty)$ (without 2), such that $h^p = S_i = \prod p_{i-1} \cdot S_{i-1} + v_i$ $(i = 1\ldots d)$.

Theorem 1. By h^p mapping, all vectors $V_j(v_1,\ldots,v_d)$ with d dimensions in a set will be randomly and uniformly projected to different integers.

Proof. Since $h^p = S_i = \prod p_{i-1} \cdot S_{i-1} + v_i$ $(i = 1\ldots d)$, then

$$\begin{aligned} S_d &= (p_2 \cdot \ldots \cdot p_i \cdot \ldots \cdot p_d)v_1 + (p_3 \cdot \ldots \cdot p_i \cdot \ldots \cdot p_d)v_2 \\ &\quad + \ldots + (p_{i+1} \cdot \ldots \cdot p_d)v_i + \ldots + v_d \\ &= v_1 \prod_{i=2}^{d} p_i + v_2 \prod_{i=3}^{d} p_i + \ldots + v_i \prod_{i=i+1}^{d} p_i + \ldots + v_d \end{aligned} . \tag{1}$$

Let $p_i < 2^{32}$, $0 < v_i \ll 2^{32}$ $(i = 1,\ldots,d)$, $\alpha_d = 1$ and $\alpha_{d-1} = p_d \cdot \alpha_d$. For any $i = 1,\ldots,d-1$, $\alpha_i = p_{i+1} \cdot \alpha_{i+1} > 0$ and $\alpha_j > \alpha_i (j > \mathrm{i})$. If $\alpha_i < 2^{32}$ and any $\alpha_j \cdot v_j > 2^{32}$ $(j > i)$, $\alpha_j \cdot v_j$ overflows, and the overflow part will be discarded by the 32-bit CPU. Since $p_j \in P$ (without 2), p_i is an odd number, and the multiplication of odd numbers is still an odd number. Since $\exists x' \in N$, $\prod_{j=j+1}^{d} p_j = 2x' + 1$. Meanwhile, $v_j \in N$ and $0 < v_i \ll 2^{32}$, therefore

$$\begin{aligned} a_i \cdot v_j &= v_j \prod_{j=j+1}^{d} p_j \\ &= v_j(2x' + 1)(x' \in N) \\ &= v_j \cdot 2x' + v_j \\ &= 2^{32} + \beta_j v_j \end{aligned} . \tag{2}$$

If $\alpha_i < 2^{32}$ and $\alpha_{i-1} = p_i \cdot \alpha_i > 2^{32}$, there always exists $z_{i-1} = 2x+1 (x \in N)$, which makes

$$\begin{aligned} \alpha_{i-1} \cdot v_{i-1} &= (2^{32} + z_{i-1}) \cdot v_{i-1} \\ &= 2^{32} \cdot v_{i-1} + z_{i-1} \cdot v_{i-1} \end{aligned} , \tag{3}$$

and

$$\begin{aligned} \alpha_{i-2} \cdot v_{i-2} &= p_{i-1} \cdot \alpha_{i-1} \cdot v_{i-2} \\ &= p_{i-1} \cdot (2^{32} + z_{i-1}) \cdot v_{i-2} \\ &= p_{i-1} \cdot 2^{32} \cdot v_{i-2} + p_{i-1} \cdot z_{i-1} \cdot v_{i-2} \end{aligned} . \tag{4}$$

The same operations are applied on other fields from $\alpha_1 \cdot v_1$ to $\alpha_{i-3} \cdot v_{i-3}$. For $v_i \in N$, all $2^{32} \cdot v_{i-1}$ and $p_j \cdot \ldots \cdot p_{i-1} \cdot 2^{32} \cdot v_j (1 \le j < i-2)$ will be discarded by the 32-bit CPU due to overflow. Then,

$$\begin{aligned} S_d &= (p_1 \cdot \ldots \cdot p_{i-1} \cdot z_{i-1})v_1 + (p_2 \cdot \ldots \cdot p_{i-1} \cdot z_{i-1})v_2 \\ &\quad + \ldots p_{i-1} z_{i-1} \cdot v_{i-2} + z_{i-1} \cdot v_{i-1} + \alpha_i v_i + \ldots \beta_j v_j + \ldots + v_d \end{aligned} . \tag{5}$$

If $p_j \cdot \ldots \cdot p_{i-1} \cdot z_{i-1} = 2^{32} + z (j < i)$, the CPU will iteratively discard the overflows. After multiple iterations,

$$\begin{aligned} S_d &= z_1 \cdot v_1 + \ldots + \ldots p_{i-2} z_{i-1} \cdot v_{i-2} \\ &+ z_{i-1} \cdot v_{i-1} + \alpha_i v_i + \ldots \beta_j v_j + \ldots + v'_d \end{aligned}, \tag{6}$$

where, for example, $p_i \in P$, $z_{i-1} = 2x+1 (x \in N)$, $z_1 = p_1 \cdot p_2 \ldots \cdot p_l \cdot z_{i-1}$ $z_2 = p_2 \ldots \cdot p_l \cdot z_{i-1}$. Therefore, $p_1 p_2 \ldots z_{i-1} \neq p_2 \ldots z_{i-1} \neq p_{i-2} z_{i-1} \neq z_{i-1} \neq \alpha_1 \neq \beta_j \ldots < 2^{32}$. Next, according to congruence theory [37],

$$\begin{aligned} S_d \% m &= (z_1 \cdot v_1 + \ldots + p_{i-2} \cdot z_{i-1} \cdot v_{i-2} + z_{i-1} \cdot v_{i-1} + \alpha_i v_i + \ldots \beta_j v_j + \ldots + v_d) \% m \\ &= (z_1 \cdot v_1 \% m \ldots p_{i-2} \cdot z_{i-1} \cdot v_{i-2} \% m + z_{i-1} \cdot v_{i-1} \% m + \alpha_i v_i \% m + \ldots \beta_j v_j \% m + \ldots + v_d \% m) \% m \end{aligned}. \tag{7}$$

Worst case: Since $p_i \in P$ (without 2), p_i is an odd number, and the multiplication of odd numbers is still an odd number. Let n_i be positive integers. Then, $\prod_{i=1} p_i = 2n_1 + 1$, $\prod_{i=2} p_i = 2n_2 + 1$ and $\prod_{i=i} p_i = 2n_i + 1$. Therefore,

$$S_d = (2n_1 + 1)v_1 + (2n_2 + 1)v_2 + \ldots + (2n_i + 1)v_i + \ldots + v_d. \tag{8}$$

For S_d, if $2n_i \geq 2^{32}$, the CPU will discard the overflow part. At worst, for all i, $2n_i = k_i 2^{32} (k_i \in N)$, then $S_d = v_1 + v_2 + \ldots + v_i + \ldots + v_d$.

The function $S_d\%m$ maps the S_d into the counter array, according to congruence theory [37].

$$\begin{aligned} S_d \% m &= (v_1 + \ldots + v_i + \ldots + v_d) \% m \\ &= (v_1 \% m + \ldots + v_i \% m + \ldots + v_d \% m) \% m \end{aligned}. \tag{9}$$

From formulas (7) and (9), even in the worst case, every dimension v_i contributes to $S_d\%m$. In fact, from formula (7), all of the coefficients are odd numbers and they are different. For a well-selected m, different v_i will have different contributions to the final result, and the change of a v_i will change $S_d\%m$. Therefore, h^p satisfies the avalanche effect of hash functions [38] and can be regard as a uniform hash function.

Lemma 1. For a vector V, functions h_i^p and h_j^p are independently selected from the Prime_HD_BKDRhash family. There exist $h_i^p(V) \neq h_j^p(V)$ and $h_i^p(V)\%m \neq h_j^p(V)\%m$.

Explain. For any two hash functions h_i^p and h_j^p, $h_i^p(V) \neq h_j^p(V)$. For simplicity, let $d = 3$, $V(v_1,v_2,v_3)$, h_1^p and h_2^p. From formula (1),

$$\begin{aligned} S_3^{h_1^p} &= (p_1 p_2 p_3) v_1 + (p_2 p_3) v_2 + p_3 v_3 \\ S_3^{h_2^p} &= (p_4 p_5 p_6) v_1 + (p_5 p_6) v_2 + p_6 v_3 \end{aligned}, \tag{10}$$

where $p_1 < p_2 < p_3 < p_4 < p_5 < p_6$ and $p_i \in P$. Therefore, $S_3^{h_1^p} \neq S_3^{h_2^p}$ and $h_1^p(V) \neq h_2^p(V)$.

Let q_1 and q_2 be quotients and r_1 and r_2 be remainders. $S_3^{h_1^p}$ and $S_3^{h_2^p}$ mod m are

$$\begin{aligned} r_1 &= S_3^{h_1^p} \% m = ((p_1 p_2 p_3) v_1 + (p_2 p_3) v_2 + p_3 v_3) \% m \\ r_2 &= S_3^{h_2^p} \% m = ((p_4 p_5 p_6) v_1 + (p_5 p_6) v_2 + p_6 v_3) \% m \end{aligned}. \tag{11}$$

For a proper m and $r_1 \neq r_2$, h_1^p and h_2^p can scatter vectors into different positions. Without the loss of generality, 3 dimensions expand to d dimensions and h_1^p and h_2^p spread to h_i^p and h_j^p. For well selected prime numbers, the worst case of formula (9) can be avoided, since $h_i^p(V) \neq$

$h_j^p(V)$ and $r_i \neq r_j$,

$$\begin{aligned} r_i &= S_d^{h_i^p} \% m = ((p_{i*d+1} p_{i*d+2} \cdots p_{i*d+d}) v_1 + (p_{i*d+t} \cdots p_d) v_t + p_d v_d) \% m \\ r_j &= S_d^{h_j^p} \% m = ((p_{j*d+1} p_{j*d+2} \cdots p_{j*d+d}) v_1 + (p_{j*d+t} \cdots p_d) v_t + p_d v_d) \% m \end{aligned}. \tag{12}$$

4.2 Algorithm

From the above discussion, the Prime_HD_BKDRhash functions can randomly and uniformly map high-dimensional vectors to integers and Algorithms 1 combined with Fig 2 demonstrate the working process.

```
Algorithm 1.
unsigned int Prime_HD_BKDRHash (int* V, int k, int d)
{
1. unsigned int prime_set = [3,5,7,11,13,17 ...];
2. unsigned int S = 0, i = 0;
3. while (*V)
4.       S = prime_set [k*d+i++] *S+ (*V++);
5. return S&0x0FFFFFFF;
}
```

The input parameters contain the vector V, dimensions d and the kth hash function. After d loops, lines 3 and 4 of Algorithm 1 obtain $h_k^p = S_i = \prod p_{i-1} \cdot S_{i-1} + v_i \ (i = 1 \ldots d)$. By performing a bitwise AND on S_i and 0x0FFFFFFF, the kth Prime_HD_BKDRHash transforms the vector V into an integer that ranges from $[0–(2^{32}–1)]$. Since different hash functions adopt different prime numbers, the return integers are different.

5. Performances

In section 4, we have demonstrated that the Prime_HD_ BKDRHash can randomly and uniformly scatter the high-dimensional vectors of a set to integers in the range of 0 to $2^{32}–1$. Therefore, the P-HDBF satisfies the theory of the BF, including all parameters and their relationships.

5.1 False positive probability (FPP), m, n, k and false negative probability (FNP)

FPP. Let there be k hash functions, a counter array of size m and a set containing n vectors with d numerical dimensions. After the n vectors are mapped onto the P-HDBF, the false positive probability of the P-HDBF is [15].

$$f_{P-HDBF} = (1 - (1 - 1/m)^{nk})^k \approx (1 - e^{-\frac{nk}{m}})^k. \tag{13}$$

Counters. For fixed k, n and FPP, the counters that the P-HDBF requires are

$$m = -\frac{kn}{\ln(1 - f_{P-HDBF}{}^{1/k})}. \tag{14}$$

Maximum cardinality. For fixed m,k and FPP, the maximum number of the vectors the P-HDBF can represent is

$$n_0 = -\frac{\ln(1 - e^{\ln\frac{f_0}{k}}) \cdot m}{k}. \tag{15}$$

Minimum number of hash functions. For fixed m, n and FPP, the minimum number of hash functions is

$$k_{\min} = \ln 2(\frac{m}{n}). \tag{16}$$

False negative probability (FNP). The FNP of the P-HDBF is

$$fnp_{P-HDBF} = 0. \tag{17}$$

5.2 Time complexity

For a hash h_j and a query q, every numerical dimension q_i will participate in the computation. By computing the k hashes, the P-HDBF obtains k integers. Next, by mapping $h_j(q)\%m$, the P-HDBF checks whether the corresponding k counters are greater than or equal to 0. If any counter is 0, we know that the query is not in the set. If all counters are larger than 0, the query is determined as a member of the set with a small FPP. For a set of n elements with d dimensions, its initialization time complexity is

$$O(kn). \tag{18}$$

The time complexity of insertion/deletion/query of a vector is

$$O(k). \tag{19}$$

6. Experiment

6.1. Dataset and setting

To verify the effectiveness of the P-HDBF on high-dimensional numeric vectors, this paper adopts 3 picture datasets, including Colour [39], Sift [40] and Gist [40], used in most experiments. On these datasets, we compare the performances of the P-HDBF with the CBF, PBF-HT and PBF-BF. The CBF is the classical method in all variants and the PBF-HT and PBF-BF support the query of multiple dimensions. The Colour includes 70,000(70K) vectors with 32 dimensions, and the values are expanded to positive integers. The Sift and Gist contain 100,000 (100K) vectors with 128 and 960 dimensions, respectively, and values of dimensions are all positive integers. All query vectors are different from the samples and are set to 10,000 (10K). The experiments were conducted on a computer with an Intel Xeon E5-2603 v3 and 16GB RAM.

6.2 Distribution and entropy

The key of the BF is that the data can be randomly and uniformly projected by hash functions. To verify this performance, we first introduce information entropy of the array in the BF after Prime_HD_BKDRHash mapping. Information entropy can describe the randomness of a system, and a larger entropy indicates a greater dispersed state. Let v' be the number of the elements allocated in a counter of the array, and n and m be the size of the set and the array, respectively. Then, the proportion of the vectors allocated in the counter can be calculated by $p \approx v'/kn$, and the entropy of all counters is defined as follows.

$$E = \sum_m -p\log p \approx \sum_m -(v'/kn)\log(v'/kn). \tag{20}$$

Let $k = 6$ and $m = 25n$. Figs 3 and 4 display the number of the vectors allocated in different counters (denoted as distribution) and entropies of the P-HDBF and the CBF on the 3 datasets. As Fig 3 shows, the distribution of the P-HDBF is similar to the CBF, which implies that the

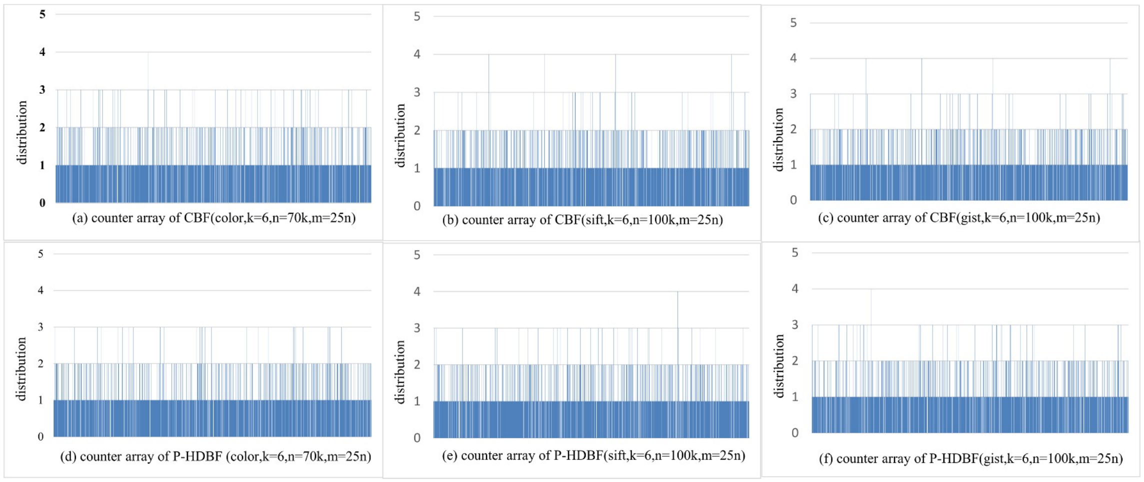


Fig 3. Number of vectors allocated in different counters of the CBF and the P-HDBF on colour (d = 32), Sift (d = 128) and Gist (d = 960).

https://doi.org/10.1371/journal.pone.0209159.g003

vectors are uniformly allocated in different counters. Fig 4 shows that the entropy of the P-HDBF is slightly larger than that of the CBF under different samples and dimensions, especially in a high-dimensional space. From the view of the entropy, larger entropy means better discretization and less collision. Figs 3 and 4 reflect that the Prime_HD_BKDRHash can scatter vectors more randomly and uniformly, especially for the high-dimensional vectors of a big data set (on Gist with d = 960). This implies that the FPP of the query after mapping by the uniform Prime_HD_BKDRHash will be less than that of multiple string hashes. The uniform Prime_HD_BKDRHash can project the numerical vectors as the inputs of the original formats and substitute multiple different string hashes of the BF.

6.3 Relationships of the FPP, n, m, d and k

This section will show whether the P-HDBF is consistent with the theory of the Bloom filter, and the indicators include the FPP, n, m, k and their relationships. The CBF, as a classical BF,

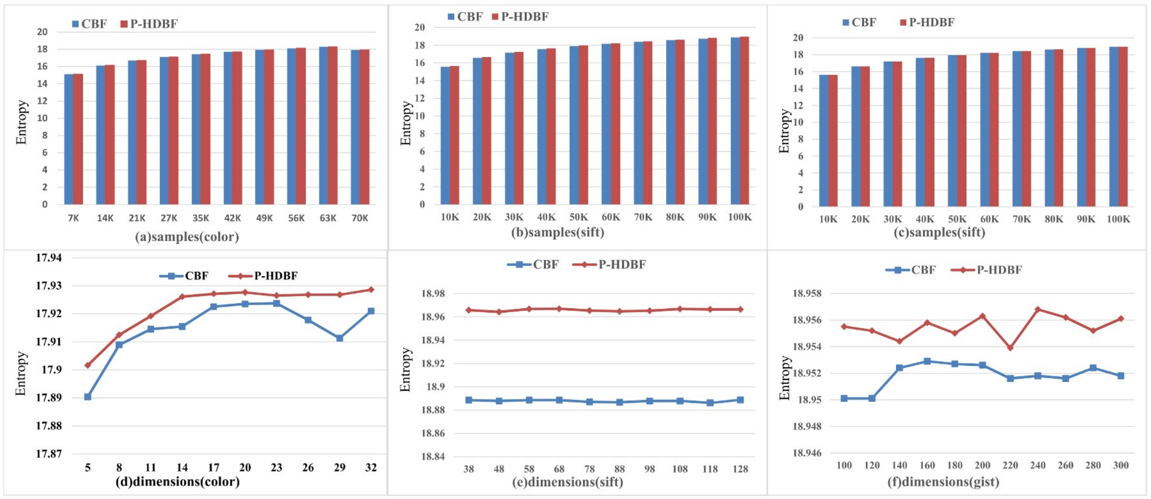


Fig 4. Entropy of the CBF and P-HDBF under different samples and dimensions.

https://doi.org/10.1371/journal.pone.0209159.g004

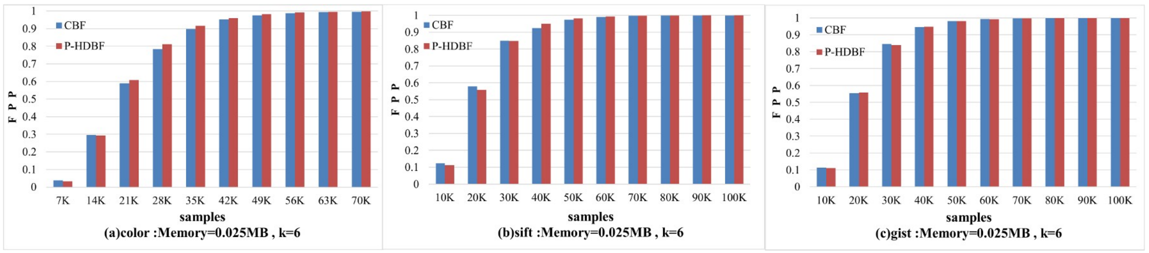


Fig 5. FPPs of the CBF and P-HDBF with an increasing n.

https://doi.org/10.1371/journal.pone.0209159.g005

is applied for comparison with the P-HDBF, and the P-HDBF should show the same tendencies as the CBF. By fixing one or two parameter(s) in turn, Figs 5, 6 and 7 show the FPPs' changes with other parameters changing. For the fixed memory costs (m) and the number of the hash functions (k), the increased collision rate causes the FPP growing. Let $m = 50k$ and $k = 6$. Fig 5 displays the increased tendencies of the FPPs as the cardinality of the set increases, even to 100%.

Then, by fixing memory costs (0.21 MB, 0.28 MB, and 0.35 MB), Fig 6 demonstrates the FPPs as k increases on the 3 datasets. For the fixed number of samples (n) and memory costs (m), the number of the hash functions (k) determines the FPP. Firstly, the FPP will decrease as k grows and reach to a minimum value, then the increasing collisions will result in a low FPP. With k rising, both FPPs sharply decrease, reach a minimum value, and then increase slowly, which is consistent with the theory of the BF.

Lastly, for $k = 6$, Fig 7 displays the similar changes in the FPPs of the CBF and the P-HDBF as m increases from $5n$ to $25n$. For fixed k and n, the FPP will be decided by memories allocated to them, and a large m can effectively reduce the FPPs.

To further observe the performance of the P-HDBF in a high dimensional space, an extra experiment is added. Let $n = 70K(100K,100K)$, $k = 6$ and the memory be 0.35 MB. Fig 8 demonstrates the changes of the FPPs with increasing dimensions. The FPPs of the P-HDBF are lower than those of the CBF, especially in certain high dimensional cases.

For different m, n and k, the FPPs' changes of the CBF and the P-HDBF are almost the same. Even the performance of the P-HDBF is better than the CBF, which implies that the P-HDBF can replace the CBF to process high-dimensional vectors. Meanwhile, the FPPs' changes of the P-HDBF are consistent with the theory in section 5. Next, we will continue to compare the performance of the P-HDBF with other methods.

6.4 Compared with other methods

Let $FPP \subset [0.0001-0.0005]$, $m = 25n$ and $k = 6$. This paper compares the memory usages of the CBF, PBF-BF and PBF-HT with the P-HDBF on 3 datasets, as shown in Fig 9. For the fixed FPP, the CBF and the P-HDBF have memory overheads. However, the memory costs of the PBF-BF and the PBF-HT grow with increased sample sizes and dimensions.

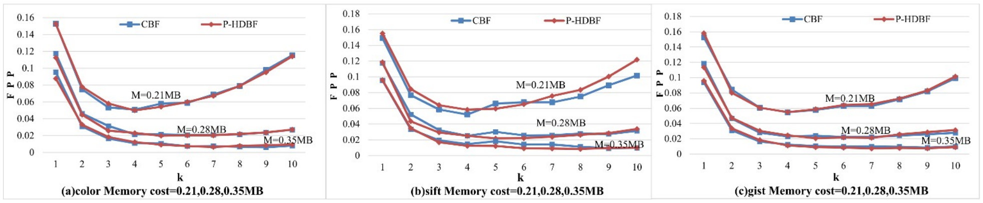


Fig 6. FPPs of the CBF and the P-HDBF under different k and memory costs.

https://doi.org/10.1371/journal.pone.0209159.g006

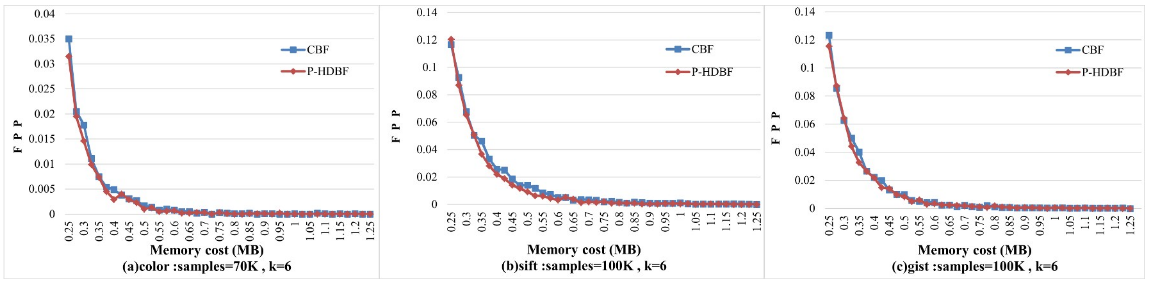


Fig 7. FPPs of the CBF and P-HDBF with the memory increase.

https://doi.org/10.1371/journal.pone.0209159.g007

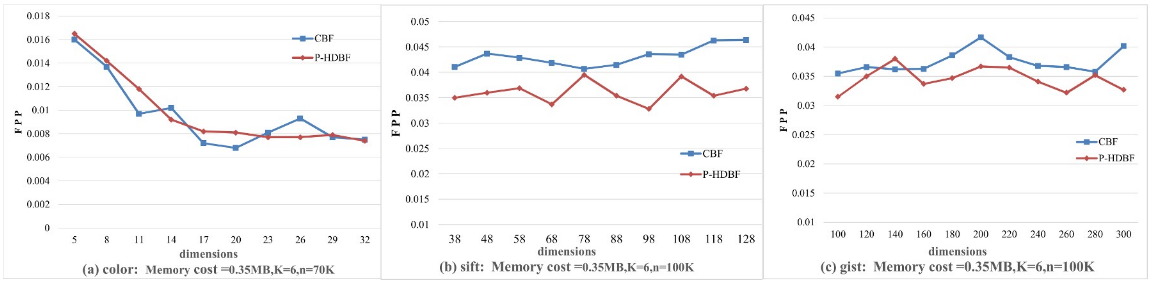


Fig 8. FPPs of the CBF and P-HDBF under different dimensions.

https://doi.org/10.1371/journal.pone.0209159.g008

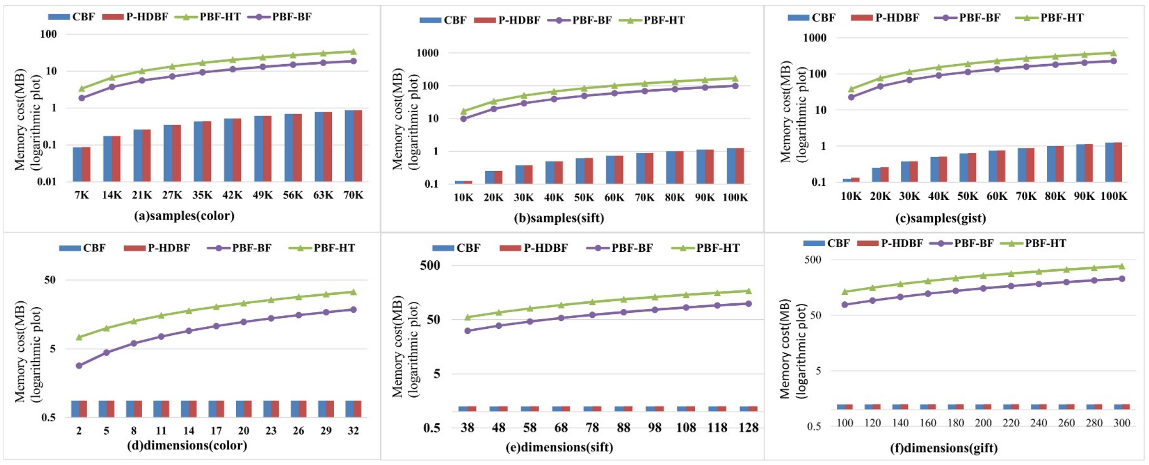


Fig 9. Memory costs under different samples and dimensions.

https://doi.org/10.1371/journal.pone.0209159.g009

Figs 10 and 11 exhibit the average initiation and query time of different schemes under 10K query vectors. Since these schemes need to split all vectors and project them into the storing arrays, the initiation and query times will continue to increase with larger samples and more dimensions. Compared with the PBF-BF and PBF-HT, the CBF and the P-HDBF only require dividing the dimensions and computing the hash values. Therefore, their initiation and query times increase slowly with more dimensions. The initiation time and query delays of the CBF and P-HDBF are far smaller than those of the PBF-BF and PBF-HT.

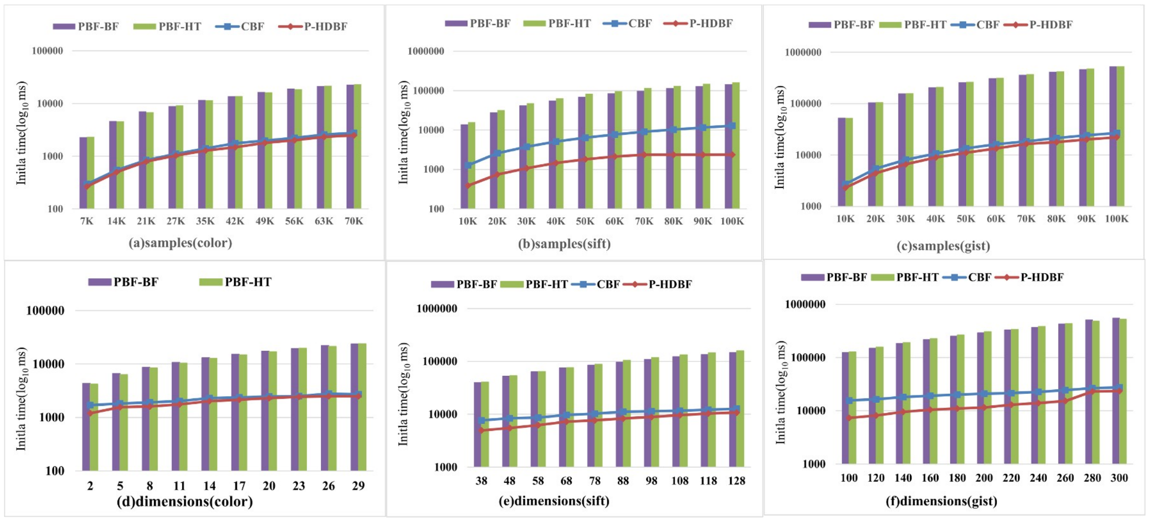


Fig 10. Average initiation time of the PBF-HT, PBF-BF, CBF and P-HDBF with *FPP*⊂[0.0001–0.0005] and *k* = 6.

https://doi.org/10.1371/journal.pone.0209159.g010

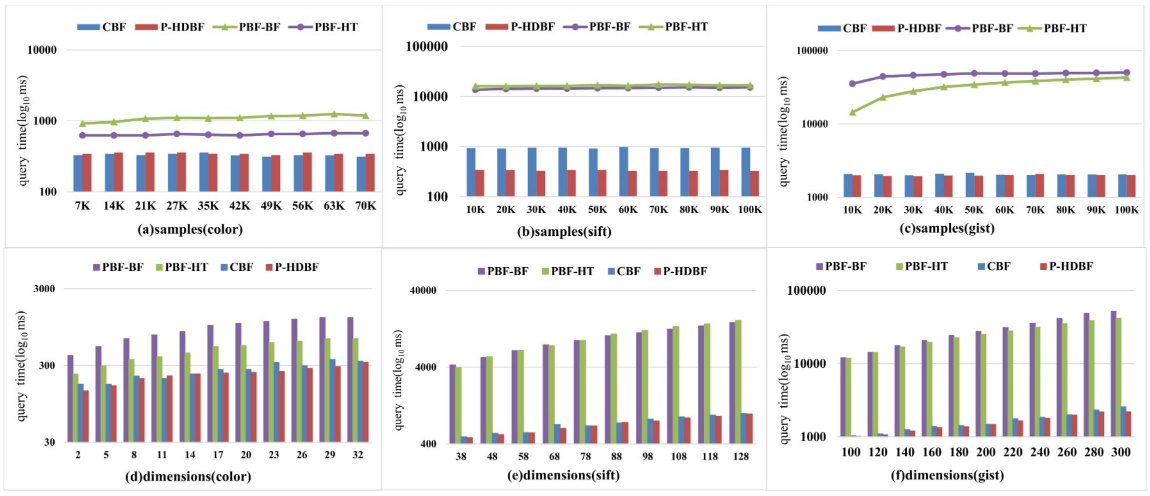


Fig 11. Average query delays of the PBF-HT, PBF-BF,CBF and P-HDBF with *FPP*⊂[0.0001–0.0005] and *k* = 6.

https://doi.org/10.1371/journal.pone.0209159.g011

Therefore, for a given FPP and a dataset with high-dimensional vectors, the P-HDBF will be a better choice than the PBF-based schemes by avoiding a long member query delay and huge memory costs.

7. Conclusions

Regardless of the formats of the inputs, the traditional Bloom filters adopt multiple string hashes to implement memberships queries of a big data set. To map the inputs with numerical high dimensions in their original type(s), this paper proposes a uniform Prime_HD_BKDR-Hash function and establishes a P-HDBF structure, a new Bloom filter, to store and query members of a big data set with numerical dimensions. The unified Prime_HD_BKDRHash can randomly and uniformly project the inputs (other than multiple string hashes) into different integers. The performances and parameters of the P-HDBF have been theoretically discussed. The experiments show that the P-HDBF, as a substitute for the counting Bloom filter in high-dimensional numerical spaces, can obtain excellent data discretization and a good performance. Compared with the methods based on the parallel Bloom filters, the P-HDBF will not increase memory use or query delays as dimensions increase and can be used in applications with limited CPU and memory resources. The P_HDBF can be applied in some applications, such as identify the nuances of pictures.

Acknowledgments

The authors would like to thank Matthijs Douze and Cordelia Schmid for their data sources, data and interpretation of data (http://corpus-texmex.irisa.fr/).

Disclaimer: The authors alone are responsible for the views expressed in this article and they do not necessarily represent the views, decisions or policies of the institutions with which they are affiliated.

Author Contributions

Conceptualization: Chunyan Shuai.

Data curation: Xin Ouyang.

Funding acquisition: Chunyan Shuai, Xin Ouyang.

Methodology: Chunyan Shuai, Jiayou Lei.

Software: Xin Ouyang.

Supervision: Chunyan Shuai.

Visualization: Zeweiyi Gong.

Writing – original draft: Zeweiyi Gong.

Writing – review & editing: Chunyan Shuai.

References

1. Bloom BH. Space/Time Trade-Offs in Hash Coding with Allowable Errors. Communications of the ACM. 1970; 13: 422–426.
2. Geravand S, Ahmadi M. Bloom Filter applications in network security: A state-of-the-art survey. Computer Networks. 2013; 57: 4047–4064.
3. Mun JH, Lim H. New approach for efficient IP address lookup using a Bloom filter in trie-based algorithms. IEEE Transactions on Computers. 2016; 65: 1558–1565.
4. Antikainen M, Aura T, Särelä M. Denial-of-service attacks in Bloom-filter-based forwarding. IEEE/ACM Transactions on Networking (TON). 2014; 22: 1463–1476.
5. Saravanan K, Senthilkumar A. Security enhancement in distributed networks using link-based mapping scheme for network intrusion detection with enhanced Bloom filter. Wireless Personal Communications. 2015; 84: 821–839.
6. Tarkoma S, Rothenberg CE, Lagerspetz E. Theory and practice of Bloom Filters for distributed systems. IEEE Communications Surveys & Tutorials. 2012; 14: 131–155.
7. Jiang P, Ji Y, Wang X, Zhu J, Cheng Y. Design of a multiple Bloom filter for distributed navigation routing. IEEE Transactions On Systems, Man, And Cybernetics: Systems. 2014; 44: 254–260.
8. Oigawa Y, Sato F. An Improvement in Zone Routing Protocol Using Bloom Filter. 19th International Conference on Network-Based Information Systems (NBiS); 2016 Sept 07–09; Ostrava, Czech Republic.
9. Ko H, Lee G, Pack S, Kweon K. Timer-based Bloom Filter aggregation for reducing signaling overhead in distributed mobility management. IEEE Transactions on Mobile Computing. 2016; 15: 516–529.

10. Zengin S, Schmidt EG. A Fast and Accurate Hardware String Matching Module with Bloom Filters. IEEE Transactions on Parallel and Distributed Systems. 2016; 28: 305–317.

11. Liu J, Chen S, Wang G, Wu T. Page replacement algorithm based on counting Bloom filter for NAND flash memory. IEEE Transactions on Consumer Electronics. 2014; 60: 636–643.

12. Fan L, Cao P, Almeida J, Broder AZ. Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans. Networking. 2000; 8: 281–293.

13. Pontarelli S, Reviriego P, Maestro JA. Improving counting Bloom filter performance with fingerprints. Information Processing Letters. 2016; 116: 304–309.

14. Lim H, Lee J, Byun H, Yim C. Ternary Bloom Filter Replacing Counting Bloom Filter. IEEE Communications Letters. 2017; 21: 278–281.

15. Mitzenmacher M. Compressed Bloom Filters. IEEE/ACM Transactions on Networking. 2002; 10: 604–612.

16. Cohen S, Matias Y. Spectral Bloom Filters. Proceedings of the 2003 ACM SIGMOD international conference on Management of data; 2003 June 09–12; San Diego, California, US.

17. Guo D, Wu J, Chen H, Yuan Y, Luo X. The dynamic Bloom Filters. IEEE Transactions on Knowledge and Data Engineering. 2010; 22: 120–133.

18. Fan B, Andersen DG, Kaminsky M, Mitzenmacher MD. Cuckoo filter: Practically better than bloom. Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies; 2014 December 02–05; Sydney, Australia.

19. Xiao MZ, Dai YF, Li XM. Split Bloom Filters. Acta Electronica Sinica. 2004; 32: 241–245.

20. Xiao B, Hua Y. Using parallel Bloom Filters for multiattribute representation on network services. IEEE Transactions on parallel and distributed systems. 2010; 21: 20–32.

21. Partow A. General Purpose Hash Function Algorithms. 2013. Available from: http://www.partow.ne-t/programming/hashfunctions/DEKHashFunction

22. Kirsch A, Mitzenmacher M. Distance-Sensitive Bloom Filters. In: The Eighth Workshop on Algorithm Engineering and Experiments (ALENEX06). Society for Industrial and Applied Mathematics; 2006. pp. 41–51.

23. Hua Y, Xiao B, Veeravalli B, Feng D. Locality-sensitive Bloom Filter for Approximate Membership Query. IEEE Transactions on Computers. 2012; 61: 817–830.

24. Qian J, Zhu Q, Chen H. Multi-Granularity Locality-Sensitive Bloom Filter. IEEE Transactions on Computers. 2015; 64: 3500–3514.

25. Qian J, Zhu Q, Chen H. Integer-Granularity Locality-Sensitive Bloom Filter. IEEE Communications Letters. 2016; 20: 2125–2128.

26. Datar M, Immorlica N, Indyk P, Mirrokni VS. Locality-Sensitive Hashing Scheme Based on P-Stable Distributions. In: Proceedings of the twentieth annual symposium on Computational geometry. New York: ACM; 2004. pp. 253–262.

27. Kernighan BW, Ritchie D. C Programming Language. 2nd ed. London: Prentice-Hall; 1988.

28. Rottenstreich O, Keslassy I. The Bloom paradox: When not to use a Bloom Filter. IEEE/ACM Transactions on Networking (TON). 2015; 23: 703–716.

29. Bose P, Guo H, Kranakis E, Maheshwari A, Morin P, Morrison J, et al. On the false-positive rate of Bloom filters. Information Processing Letters. 2008; 108: 210–213.

30. Christensen K, Roginsky A, Jimeno M. A new analysis of the false positive rate of a Bloom filter. Information Processing Letters 2010; 110: 944–949.

31. Rottenstreich O, Kanizo Y, Keslassy I. The variable-increment counting Bloom filter. IEEE/ACM Transactions on Networking (TON). 2014; 22: 1092–1105.

32. Qiao Y, Li T, Chen S. Fast Bloom Filters and their generalization. IEEE Transactions on Parallel and Distributed Systems. 2014; 25: 93–103.

33. Reviriego P, Christensen K, Maestro JA. A Comment on "Fast Bloom Filters and Their Generalization". IEEE Transactions on Parallel and Distributed Systems. 2016; 27: 303–304.

34. Crainiceanu A, Lemire D. Bloofi: Multidimensional Bloom Filters. Information Systems. 2015; 54: 311–324.

35. Fu Y, Biersack E. False-Positive Probability and Compression Optimization for Tree-Structured Bloom Filters. ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS). 2016; 1: 19.

36. Heyde CC. Central Limit Theorem. In: Convergence of Stochastic Processes. New York: Springer; 1984. pp. 3244–3248.

37. Katz VJ. Chapter 3: Chinese Mathematics. In: The Mathematics of Egypt, Mesopotamia, China, India and Islam: A Sourcebook. Princeton University Pres; 2007. pp. 187–384.

38. Feistel H. Cryptography and Computer Privacy. Scientific American. 1973; 228:15–23.

39. Fagin R, Kumar R, Sivakumar D. Efficient similarity search and classification via rank aggregation. Proceedings of the 2003 ACM SIGMOD international conference on Management of data; 2003 June 09–12; San Diego, California, US.

40. Amsaleg L. Datasets for approximate nearest neighbor search. 2017. Available from: http://corpus-texmex.irisa.fr/