information

MDPI

Article

A Bloom Filter for High Dimensional Vectors

Chunyan Shuai [1], Hengcheng Yang [2], Xin Ouyang [3,*] and Zeweiyi Gong [2]

[1] Faculty of Transportation Engineering, Kunming University of Science and Technology, Kunming 650221, China; earth0806@kmust.edu.cn or earth0806@sina.com
[2] Faculty of Electric Power Engineering, Kunming University of Science and Technology, Kunming 650221, China; yanghengcheng112@163.com (H.Y.); earth_0806@163.com (Z.G.)
[3] Faculty of Information Engineering and Automation, Kunming University of Science and Technology, Kunming 650221, China
* Correspondence: oyx@kmust.edu.cn or kmoyx@hotmail.com; Tel.: +86-871-6530-0096

Received: 25 April 2018; Accepted: 20 June 2018; Published: 2 July 2018

Abstract: Regardless of the type of data, traditional Bloom filters treat each element of a set as a string, and by iterating every character of the string, they discretize all data randomly and uniformly. However, with the data size and dimension increases, these variants are inefficient. To better discretize vectors with high numerical dimensions, this paper improves the string hashes to integer hashes. Based on the integer hashes and a counter array, we propose a new variant—high-dimensional Bloom filter (HDBF)—to extend the Bloom filter into high-dimensional spaces, which can represent and query numerical vectors of a big set with a low false positive probability. This paper theoretically analyzes the feasibility of the integer hashes on discretizing data and discusses the relationship of parameters of the HDBF. The experiments illustrate that, in high-dimensional numerical spaces, the HDBF shows better randomness on distribution and entropy than that of the counting Bloom filter. Compared with the parallel Bloom filters, for a fixed false positive probability, the HDBF displays time-space overheads, and is more suitable to deal with the numerical vectors with high dimensions.

Keywords: Bloom filter; high-dimensional numerical vector; high-dimensional Bloom filter; integer hash functions

1. Introduction

In high-dimensional spaces, exact search methods, such as kd-tree approaches [1] and Q-gram [2], are only suitable for small size vectors due to the very large computational resources. With the development of sensor technology, communication technology and storage technology, big data with high dimensions have brought a new challenge to the retrieval and storage of the data. There is a compromise between efficiency and accuracy. A Bloom filter (BF) [3] and its variants [4,5], as a type of space-efficient and constant query delay random data structure, have been applied to represent a big set and retrieve memberships broadly [4,5], including IP address lookup [6,7], routing-table lookup [8], hardware String Matching [9,10], cooperative Web caching [11], intrusion detection [12–15], and so on.

The standard BF [3] includes a bit (or counter) array and multiple string hash functions, which stores all elements of a given set into the bit-array using these multiple string hash functions. During the past 30 years, different variants of the BF have optimized and extended BF from different perspectives, which make BFs more suitable for different circumstances and requirements [4]. The performances are comprehensively discussed in References [16,17], including the memory cost, size of the set, false positive probability (FPP) and number of the hash functions. Reference [18] analyzes the conditions under which the paradox occurs in the BF and demonstrates that it highly depends on the prior probability that a given element belongs to the represented set. Complement BF [19]

proposes to identify the trueness of BF positives, which effectively solves the FPPs produced by the BF. The modifications mainly concentrated on the following three aspects.

1. Improvements on the array. For most of the applications, Counting BF (CBF) [20] proves that 4-bit counter array is enough to defend FPPs brought by elements deletion. The Variable-Increment CBF (VI-CBF) [21] increases the counter at a variable step instead of 1 to further reduce the FPPs of the CBF. By combination with VI-CBF, fingerprint CBF (FP-CBF) adds fingerprints to the elements stored in the CBF to reduce the FPP of the VI-CBF. These two improvements on the FPP are at the cost of memory, VI-CBF needs more bits to store the variable steps and another one requires extra spaces to store the fingerprints. Improved associative deletion BF (IABF) [22], adopts multiple CBF to store multiple attributes, and a check CBF to keep the association information on these attributes of items in the given data set. By operation of these CBF, the IABF can support association attribute deletion. The Shifting Bloom filter (ShBF) [23] can quickly process membership, association, and multiplicity queries of sets, using a small amount of memory.
2. Modifications on the structure. For a static set, it is possible to know the whole set in advance and design a perfect hash function to avoid hash collisions. However, for the set with uncertain cardinality, BF and CBF are unsuitable. Scalable BF (SBF) [24] and Dynamic BFs (DBF) [25] propose to represent a dynamic set instead of rehashing the dynamic set into a new filter as the set size changes. If the original DBF is full, the DBF adds a new CBF dynamically and merges it into the DBF. Let the DBF own s sub-CBFs and a sub-CBF corresponds k hash functions, time complexities of membership query (or deletion) for sub-CBF and DBF are $O(k)$ and $O(k \times s)$, respectively. However, in the DBF, there is no mechanism to control the overall FPP, although sub-CBF makes the upper bound of FPP converge, it does not support useful bit vector-based algebra operations between sub-CBFs and may cause unacceptable space waste; in addition, the DBF has to query all sub-CBFs one by one to discover the query. Partitioned BF (Par-BF) [26] partitions the whole required memory space into disjointed sub-CBF lists. Each sub-CBF list can index maximal n_c elements. Each sub-CBF list can contain s homogeneous CBFs at most. A thread is assigned into each sub-CBF list to do the matching, via parallel computing, the Par-BF gets query time optimization. PBF [27], PBF-BF and PBF-HT [28] extend the BF to represent and query elements with multi-dimensional text attributes. The PBF allocates d dimensions into d parallel BFs, but the integrity of an element is destroyed, which results in a high FPP. PBF-BF and PBF-HT also introduce d parallel CBF (PBF) arrays to store d dimensions, to keep these d dimensions integrity, a check BF (PBF-BF) or a hash table (PBF-HT) is added. However, the space occupations increase linearly, not only with the rising cardinality, but also the dimensions, which lead to huge memory wastes and large query delays. All of them make parallel BFs not suitable for the big set in a high-dimensional space.
3. Different hash functions. LshBF [29] schemes adopt local sensitive hash functions (LSH) instead of string hash functions to solve the Approximate Membership Query (AMQ). The LSH function follows a P-stable ($p \in (0, 2]$) distribution, only when $p = 1$ and 2, the probability density functions can be written, and they are a Cauchy distribution ($p = 1$) and Gaussian (normal) distribution ($p = 2$), respectively. The LSH function maps the neighbors in Euclidean spaces to nearby locations, and directly changes high-dimensional vectors into real numbers. To reduce the FPP, the LshBF-BF [30] adds a verification BF to further disperse points in the LshBF. Multi-Granularity LshBF-BF (MLBF) [31] develops multiple granularities search instead of one step in the LshBF-BF to further improve the search accuracy. However, MLBF is designed to only filter (query) objects with multiple logarithmic distance granularities. The integer-granularity locality-sensitive Bloom filter (ILBF) [32] filters objects with multiple integer distance granularities to shrink the distances and to reduce the FPP in MLBF. All these schemes are based on the LSH, according to the central limited theorem, after mapping, the LSH shrinks most of elements of the set around the mean, which results in a high FPP in member query, especially around the

mean. For example, approximately 68.5% elements are projected to the locations between the negative and positive variance under Gaussian distribution. Through LSH mapping, the LshBFs transform high-dimensional vectors into real numbers, which avoids dimension disasters and brings computing and space overheads, but the aggregation of neighbors causes high FPPs. This, LshBFs are only suitable for AMQ not membership query.

Whatever the types of elements may be, the BF and the variants in above 1 and 2 view every element of a set as a flat one-dimensional string. By iteratively computing the characters of each input, they yield a random hash value and project it in a fixed array. For elements with all dimensions being numerical, such as pictures, the LshBFs [29–32] P-stable-function-based will provide a high FPPs for membership query. To answer the membership lookup of a large set with high-dimensional vectors, this paper modifies the string hash functions of BF and implements a new BF structure, denoted as HDBF. The experiments demonstrate that the HDBF has the same performances as CBF as regards data discretization, which can efficiently deal with the vectors in high numerical dimensional spaces. The main contributions of this paper are as follows.

1. The modified hash functions can effectively discretize vectors with numerical high dimensions, uniformly and randomly. Based on the modified hash functions, HDBF extends the Bloom filters to represent and query numerical vectors in high-dimensional spaces.
2. Compared with CBF, HDBF is more efficient in dealing with numerical dimensions, and can be a replacement for CBF in numerical high-dimensional spaces
3. HDBF outperforms CBF in false positive probability, query delay, memory costs, and especially in numerical high-dimensional spaces.
4. Different from parallel BFs (PBFs), HDBF will not bring dimension disaster. Moreover, it has memory and query overheads compared with PBFs.

2. Work Mechanism and Structure

2.1. Bloom Filter

Definition 1. *A standard BF [3] applies an array of m bits, initially all are set to 0, and k independent hash functions h_i to represent a set $S = \{a_1, \ldots, a_n\}$ of n elements, as shown in Figure 1a. If an element is mapped into the BF by h_i, the corresponding bit $h_i(a_j)\%m$ is set to 1. Given a query q, by k hash functions $h_i(q)\%m$ mapping, the BF answers whether the q is a member of S with a false positive probability (FPP). CBF [20] replaces the bit-array with the 4-bit counter array to support element deletion, as shown in Figure 1b.*

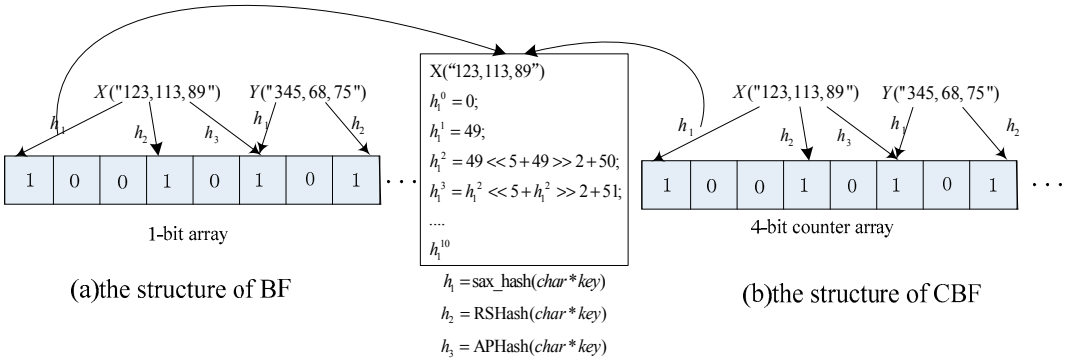

Figure 1. Structure of high-dimensional vector Bloom filter.

A BF includes hash functions and the bit array, this paper takes sax_hash, a classical string hash function [33] used by the BF and most of the generations, as an example to illustrate the work mechanism of string hash functions, shown in Figure 1 and Algorithm 1. Given a three-dimensional

vector x with numeral dimensions 123, 113, and 89, if the vector is inputted into the sax_hash, the sax_hash regards it as a string key = "123, 213, 89". By determining every character's ASCII code {"49", "50", "51", "44", ... } and bitwise operating, the sax_hash gets a hash value ranged in $[0 - 2^{31}]$. The iterating process is

$$\begin{aligned} h_1^1 &= 49 \\ h_1^2 &= h_1^1 \ll 5 + h_1^1 \gg 2 + 50 \\ h_1^3 &= h_1^2 \ll 5 + h_1^2 \gg 2 + 51 \\ &\cdots \end{aligned} \tag{1}$$

as shown in Figure 1 and line 3 and 4 of Algorithm 1. The operations of other string hashes are similar to the sax_hash, such as RSHash and APHash. Through k different string hash computing and MOD function $h_i(x)\%m$ mapping, the input is stored in the bit (or counter) array.

Algorithm 1

```
unsigned int sax_hash(char *key)
{ 1. unsigned int h = 0;
2. while(*key)
3. h ^ = (h << 5) + (h >> 2) + (unsigned char)*key++;
4. return (h& 0x7FFFFFFF);
}
```

2.2. High Dimensional Bloom Filter (HDBF)

BF assumes that all the elements of a set can be randomly and uniformly scattered into a range of integers. The string hash functions should satisfy: (1) Different vectors being projected to different values by the same hash function; (2) same vector being projected to different values by different hash functions; and (3) the avalanche effect [34]. The change of single character will bring big change of the hash value.

The Sax hash regards the input parameter "key" as a one-dimensional string, and every character in the "key" will be computed. In fact, the characters the sax_hash operated are the ASCII codes, which are a series of integers. This implies that the input can be modified to an integer array. Thus, by modification of the input, we can expand BF into high-dimensional spaces. Except for replacing the input string (char *key) with an integer array (int *key), the modified sax_hash, called HDsax_hash, as shown in Algorithm 2. The HDsax_hash regards the input as a three-dimensional integer vector. By bitwise operating on integers 123, 213 and 89, as shown in lines 3 and 4 of Algorithm 2 and Figure 2, the HDsax_hash computes a hash value of the three-dimensinal vector. The operation process is

$$\begin{aligned} HD\_h_1^1 &= 123 \\ HD\_h_1^2 &= HD\_h_1^1 \ll 5 + HD\_h_1^1 \gg 2 + 113 \\ HD\_h_1^3 &= HD\_h_1^2 \ll 5 + HD\_h_1^2 \gg 2 + 89 \end{aligned} \tag{2}$$

Algorithm 2

```
unsigned int HDsax_hash(int *key)
{ 1. unsigned int h=0;
2. while(*key)
3. h ^= (h<<5) + (h>>2) + (unsigned int)*key++;
4. return (h& 0x7FFFFFFF);
}
```

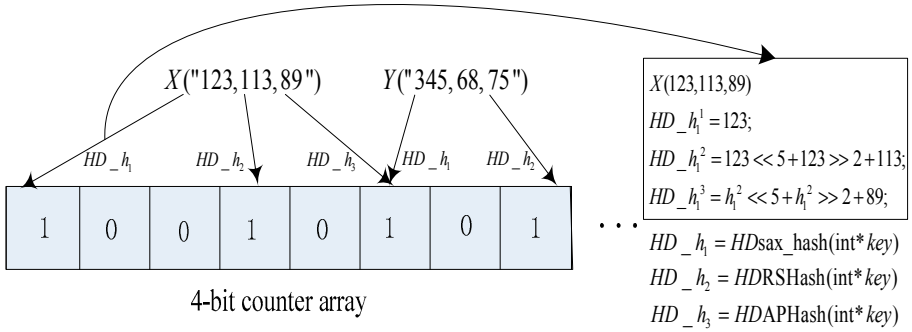

Figure 2. Structure of high dimension vector Bloom filter.

The similar modifications are applied on other string hash functions and the high-dimensional integer hash function (HDIH) family is obtained. Based on the HDIH and a counter array, a new BF structure, denoted as HDBF, is constructed to store and query the vectors with high numerical dimensions in a large set.

Definition 2. *A high-dimensional integer BF (HDBF) applies an array of m counters. Initially all are set to 0, and k independent HDIH functions HD_h_i to represent a set $S = \{V_1, V_2, \ldots, V_n\}$ of n vectors, where any vector with d numerical dimensions $V_j(v_{j1}, \ldots, v_{jd}), v_{jl} \in I$, as shown in Figure 2. If vector V_j is mapped into the HDBF by HD_h_i, the corresponding counter $HD\_h_i(V_j)\%m$ is increased by 1. Given query q, if all k HDIH functions $HD\_h_i(q)\%m$ are bigger than 1, the HDBF regards q as a member of S with a FPP; if not, the query is certainly not in set S.*

3. Performances

Due to the same data process, counter array and data type of CBF [20] and HDBF, they have the same performance, which can map a vector into an integer, ranged in $[0 - 2^{31}]$, randomly and uniformly. After n vectors are mapped into the counter array with m size by k HDIH functions, the probability of any one counter still being 0 is:

$$p' = (1 - \frac{1}{m})^{kn} \approx e^{\frac{-kn}{m}} = p \tag{3}$$

If a false positive occurs, the corresponding counter must be 1, so the false positive probability (FPP) is:

$$f_{HDBF} = (1 - p)^k = (1 - e^{\frac{-kn}{m}})^k = \exp(k \ln(1 - e^{\frac{-kn}{m}})) \tag{4}$$

From Equation (4), the memory required by HDBF is:

$$m = -\frac{kn}{\ln(1 - (f_{HDBF})^{\frac{1}{k}})} \tag{5}$$

Let the upper limit of the FPP of the HDBF be f_0. For fixed m and k, from Equation (4), the maximum number of vectors the HDBF can represent is n_0, and

$$n_0 = -\frac{\ln(1 - e^{\ln \frac{f_0}{k}}) \cdot m}{k} \tag{6}$$

In terms of Equation (4), given that $g(k) = k\ln(1 - e^{-\frac{kn}{m}})$, then $f_{HDBF} = e^{g(k)}$. To get the minimum value of f_{HDBF}, function $g(k)$ is derivative using k,

$$\frac{dg(k)}{dk} = \ln(1 - e^{-\frac{kn}{m}}) + \frac{kn}{m}\frac{e^{\frac{-kn}{m}}}{1 - e^{\frac{-kn}{m}}} \tag{7}$$

When $\frac{dg(k)}{dk} = 0$, the minimum number of the hash functions is obtained, and:

$$k_{\min} = (\ln 2)(\frac{m}{n}) \tag{8}$$

Since HDBF only needs k hash computing for a query/deletion/insertion, the query time complexity is $o(k)$.

4. Experimental Section

4.1. Dataset and Settings

Since there are no benchmark datasets for BFs, here Color [35], Sift and Gist [36], used in most experiments, are adopted to compare and test the performances of different variants. The Color dataset includes 70 K vectors with 32 dimensions, and values of the dimensions are all less than 1, we expanded all values into integers. Sift and Gist contain 100 K vectors with 128 and 300 dimensions, respectively, and the values of dimensions are positive integers. All query vectors are different from the samples and are set to 10 K. The experiments ran on a computer with Intel Xeon E5-2603 v3 and 16 GB RAM. The schemes used to compare contain CBF [20], PBF-HT and PBF-BF [28], in which all counters of the arrays took up 4 bits.

4.2. Distribution and Entropy

Since the distribution and entropy reflect the discrete state of data, to check whether HDBF can scatter the high-dimensional vectors into different integers, randomly and uniformly, this paper firstly compares the distribution and entropy of HDBF with CBF on 3 datasets. Let v be the value of a counter after n vectors are projected by k hash functions, and $p = v/kn$ be the selected frequencies of a counter. The entropy of the counter array is defined as

$$E = -p\log p \tag{9}$$

Given $m = 25n$. Figure 3 shows the distributed situations of CBF and HDBF after different datasets are projected into counter arrays by 6 different hash functions. Figure 3a–c shows the distributions of the CBF, where the maximum values of the counters are 6, 6 and 12 on Color, Sift and Gist, respectively. Figure 3e–g demonstrates the distributions of HDBF, in which the maximum values are 6, 6 and 7, respectively. This illustrates that the HDIH functions almost possess the same discrete ability as the string hashes of CBF.

Figure 4a–c displays the increase of entropies of HDBF and CBF with samples increases under d, being 32, 128 and 300. For fixed n, Figure 4d,e shows the changes of entropies with the dimension increase. In Figure 4a,d, HDBF and CBF almost have similar entropies in low-dimensional spaces ($d \leq 32$). With the increase in dimension (Figure 4b,c,e,g), the entropies of HDBF are slightly larger than those of CBF, obviously, for Sift and Gist where $d > 32$. This means that HDBF is superior to the CBF on the data discretization, especially in high-dimensional spaces.

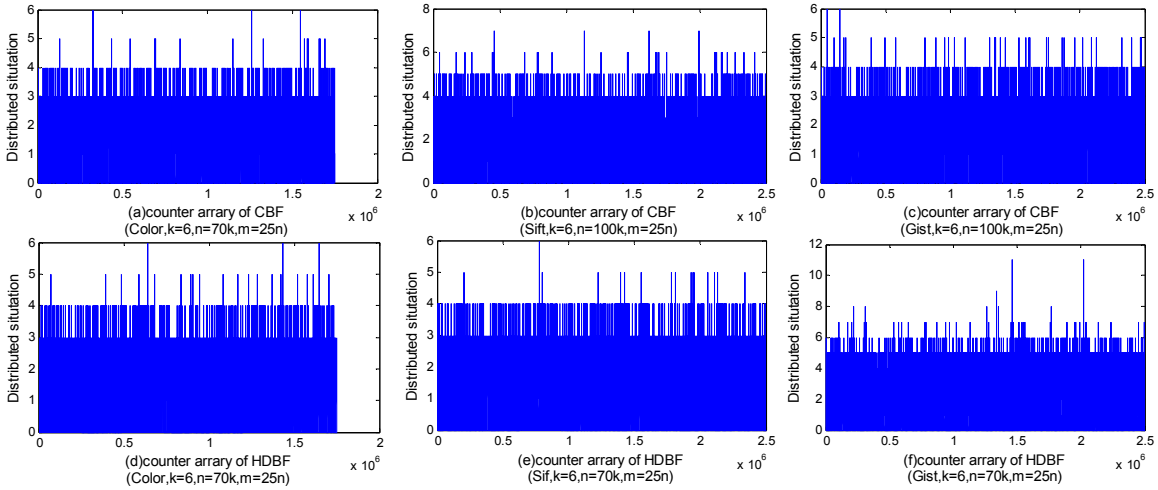

Figure 3. The distributions of CBF and HDBF for different samples.

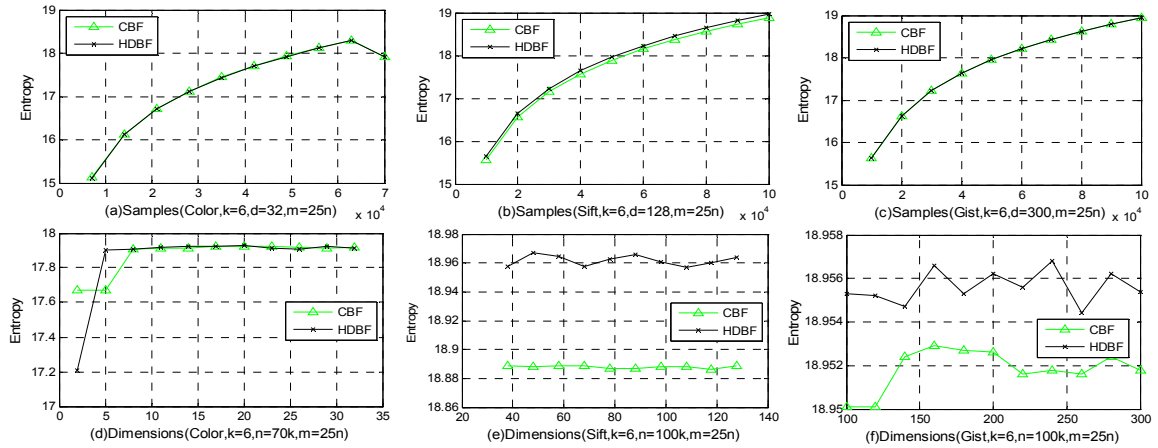

Figure 4. Entropies of CBF and HDBF for different samples.

4.3. FPP

Figure 5 displays the FPP changes with the increase of k for different memory costs and fixed samples. From Equation (8) in Section 4, for fixed m and n, there are a minimum number of hash functions; the FPPs first decrease to a minimum, then increase with the increase in k. The CBF and HDBF have the same change tendencies.

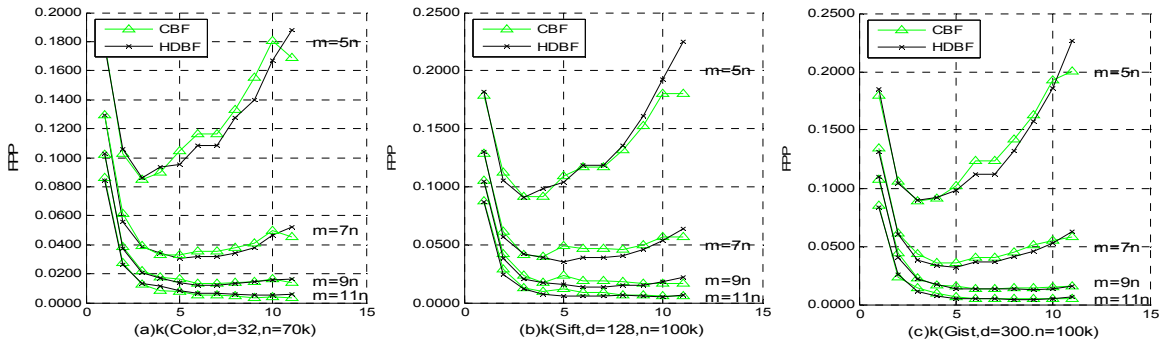

Figure 5. FPPs of the CBF and HDBF for different k and memory cost.

From Equation (8), for a fixed $k = 6$ and memory costs, the FPP will increase with sample growth, even reaching 1, as Figure 6 shows. On the contrary, for a fixed k and n, the FPP will decrease with the increase in memory, as shown in Figure 7. From Figures 5–7, we can clearly see that the FPPs of the CBF and HDBF almost possess the same values, display similar change tendencies, and they are close to meeting the false positive probability requirements.

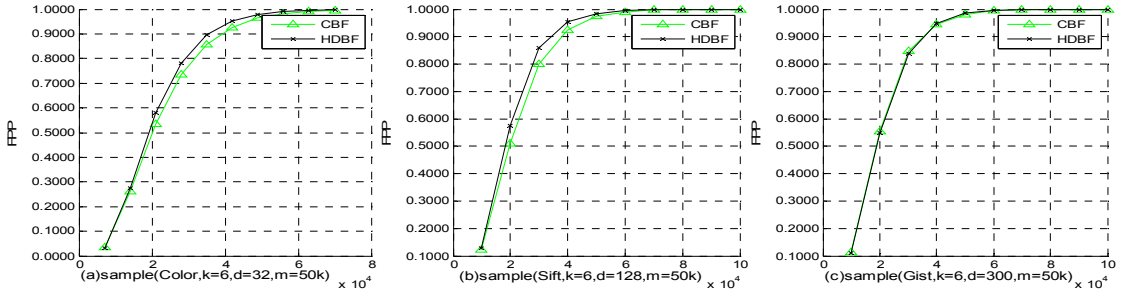

Figure 6. FPPs of the CBF and HDBF for different samples.

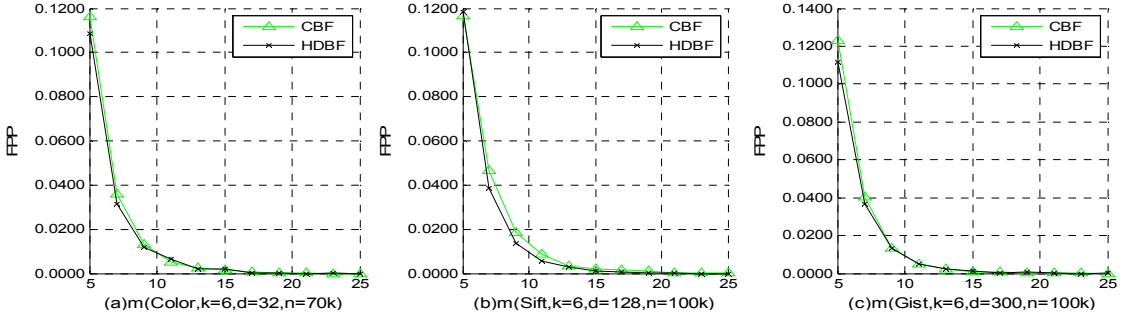

Figure 7. FPPs of the CBF and HDBF for different memory costs.

The above discussions show that HDBF can discretize data with high-dimensions, randomly and uniformly, which can substitute CBF for dealing with vectors with numerical high-dimensions. The following sections will continue to compare HDBF with other schemes based on BFs.

4.4. Memory Costs and Latency

Let the average $FPP \subset [0.0001 - 0.0005]$, $m = 25n$ and $k = 6$. Figure 8 compares the memory use of the PBF-BF, PBF-HT, CBF and HDBF on 3 datasets. For fixed FPPs, the memory costs of PBF-BF and PBF-HT enlarge with the increase in the samples and dimensions, linearly, which is in line with the discussions in Section 2. According to Equation (4), with the n growing, m will enlarge to fit a constant FPP, so the memory usages of HDBF and CBF increase with the number of samples (Figure 8a–c), and will not be affected by the dimensions (Figure 8d–e). Once the dimensions are greater than 1, the memory costs of PBF-BF and PBF-HT are far higher than those of CBF and HDBF, as shown in Figure 8d–f.

Under 10 K query vectors, the average initiation and query time of CBF and HDBF are less than PBF-HT and PBF-BF, as shown in Figures 9 and 10. Since all schemes need to split vectors and project all dimensions into corresponding arrays, the initiation time will continue to increase with the samples and dimensions. However, the increased speeds of CBF and HDBF are far slower than those of PBF-BF and PBF-HT, as shown in Figure 9. Compared with PBF-BF and PBF-HT, CBF and HDBF only require dividing the dimensions and computing the hash values, so their query times will increase slightly with the increased dimensions (Figure 10d–e) but are constant as cardinality increase (Figure 10a–c), which is consistent with Equation (9). Since PBF-HT and PBF-BF contain multiple BFs, once any one BF returns to 0, the query will stop, the query time fluctuates slightly with the increase in dimensions.

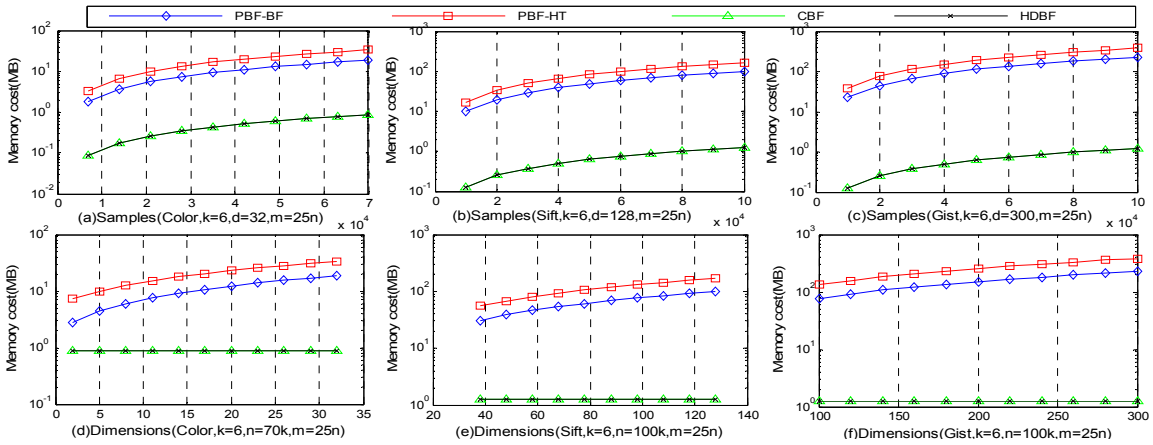

Figure 8. Memory usages of the PBF-BF, PBF-HT, CBF and HDBF with a fixed FPP.

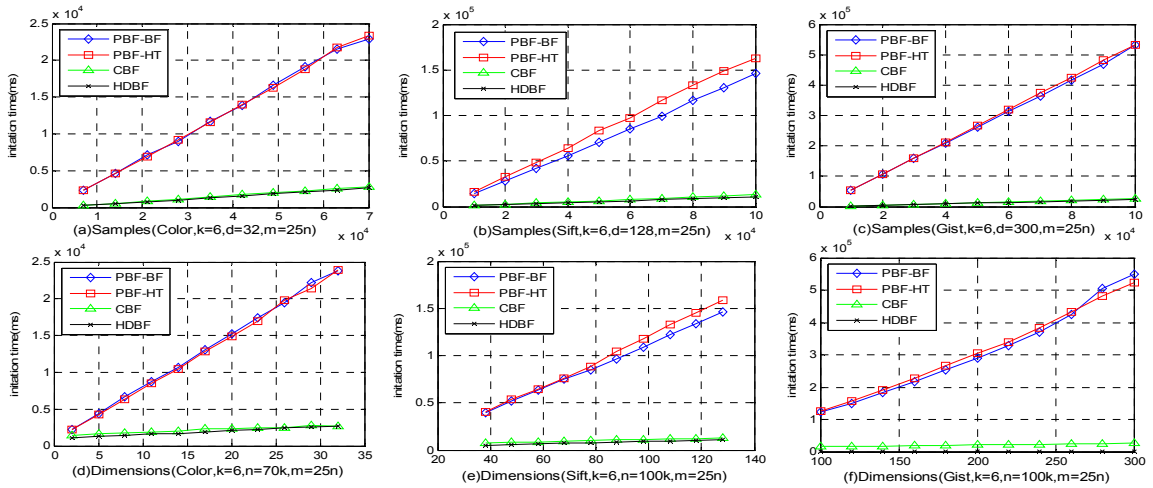

Figure 9. Average initiation time of the PBF-HT, PBF-BF, CBF and HDBF with $FPP \subset [0.0001 \ - \ 0.0005]$, $k = 6$.

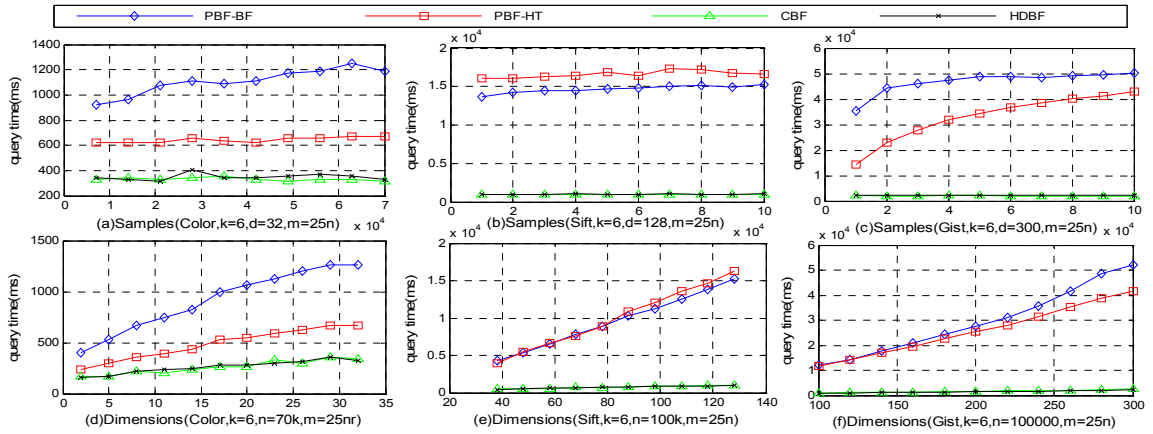

Figure 10. Average query latency of the PBF-HT, PBF-BF, CBF and HDBF with $FPP \subset [0.0001 - 0.0005]$, $k = 6$.

5. Conclusions

With the development of computer technology, data dimensions and sizes increase quickly, and the requirements for tools and methods for dealing with high dimensional data are becoming urgent. Although there are some data structures for high-dimensional data in a number of variants of BF, there are some problems, such as high temporal and spatial costs. In this paper, we proposed a new hash family, called HDIH, to map the vectors with high-dimensions. Based on the HDIH family and a counter array, a new Bloom filter structure, denoted as HDBF, was built to represent and query the vectors with numerical high-dimensions in a large set. The HDBF regards all elements in a set as vectors while not strings. By iteratively operating the dimensions of the input vectors, the HDBF can translate the vectors into a series of integers, randomly and uniformly. This paper theoretically discusses the relationships of false positive probability, memory costs and hash functions of HDBF. The experiments showed that the distribution of HDBF is almost the same as that of CBF, and the entropy of HDBF in high-dimensional spaces is slightly larger than that of CBF. This means that HDBF has a better data discrete ability than CBF, which can replace CBF to deal with vectors with high-dimensions, randomly and uniformly. Compared with PBF-BF and PBF-HT, HDBF has memory and query overheads, and the memory costs and query time will not be affected by the dimensions. Therefore, HDBF, as a substitute for CBF, is suitable for representing and querying numerical vectors in a high-dimensional space.

Author Contributions: C.S. contributes in the conceptualization, project administration, funding acquisition and methodology; X.O. contributes in software, supervision and investigation; H.Y. and Z.G. contributes in validation formal analysis, writing-original draft preparation, writing-review and editing, visualization.

Funding: This work is supported by National key R&D project of ministry of science and technology of China (No. 2017YFB0306400), the National Natural Science Foundation of China (No. 61562056).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Hunt, W.; Mark, W.R.; Stoll, G. Fast kd-tree construction with an adaptive error-bounded heuristic. In Proceedings of the IEEE Symposium on Interactive Ray Tracing, Salt Lake City, UT, USA, 18–20 September 2006; pp. 81–88.
2. Burkhardt, S.; Crauser, A.; Ferragina, P.; Lenhof, H.; Rivals, E.; Vingron, M. q-gram based database searching using a suffix array (QUASAR). In Proceedings of the Third Annual International Conference on Computational Molecular Biology, Lyon, France, 11–14 April 1999; ACM: New York, NY, USA, 1999; pp. 77–83.
3. Burton, H.B. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* **1970**, *13*, 422–426. [CrossRef]
4. Broder, A.; Mitzenmacher, M. Network applications of bloom filters: A survey. *Internet Math.* **2004**, *1*, 485–509. [CrossRef]
5. Tarkoma, S.; Rothenberg, C.E.; Lagerspetz, E. Theory and practice of bloom filters for distributed systems. *IEEE Commun. Surv. Tutor.* **2012**, *14*, 131–155. [CrossRef]
6. Ju, H.M.; Lim, H. New Approach for Efficient IP Address Lookup Using a Bloom filter in Trie-Based Algorithms. *IEEE Trans. Comput.* **2016**, *65*. [CrossRef]
7. Kwon, M.; Reviriego, P.; Pontarelli, S. A length-aware cuckoo filter for faster IP lookup. In Proceedings of the 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), San Francisco, CA, USA, 10–14 April 2016.
8. Nikolaevskiy, I.; Lukyanenko, A.; Polishchukc, T.; Polishchuk, V.; Gurtov, A. isBF: Scalable in-packet bloom filter based multicast. *Comput. Commun.* **2015**, *70*, 79–85. [CrossRef]
9. Zengin, S.; Schmidt, E.G. A fast and accurate hardware string matching module with Bloom filters. *IEEE Trans. Parallel Distrib. Syst.* **2017**, *28*, 305–317. [CrossRef]
10. Lin, P.-C.; Lin, Y.-D.; Lai, Y.-C.; Zheng, Y.-J.; Lee, T.-H. Realizing a sub-linear time string-matching algorithm with a hardware accelerator using bloom filters. *IEEE Trans. Very Large Scale Integr. Syst.* **2009**, *17*, 1008–1020. [CrossRef]

11. Alexander, H.; Khalil, I.; Cameron, C.; Tari, Z.; Zomaya, A. Cooperative Web Caching Using Dynamic Interest-Tagged filtered Bloom filters. *IEEE Trans. Parallel Distrib. Syst.* **2015**, *26*, 2956–2969. [CrossRef]
12. Antikainen, M.; Aura, T.; Särelä, M. Denial-of-service attacks in bloom-filter-based forwarding. *Trans. Netw. IEEE/ACM* **2014**, *22*, 1463–1476. [CrossRef]
13. Parthasarathy, S.; Kundur, D. Bloom filter based intrusion detection for smart grid SCADA. In Proceedings of the IEEE Canadian Conference on Electrical & Computer Engineering (CCECE), Montreal, QC, Canada, 29 April–2 May 2012; pp. 1–6.
14. Meghana, V.; Suresh, M.; Sandhya, S.; Aparna, R.; Gururaj, C. SoC implementation of network intrusion detection using counting bloom filter. In Proceedings of the 2016 IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT), Bangalore, India, 20–21 May 2016; pp. 1846–1850.
15. Aldwairi, M.; Al-Khamaiseh, K. Exhaust: Optimizing Wu-Manber pattern matching for intrusion detection using Bloom filters. In Proceedings of the IEEE Web Applications and Networking, Sousse, Tunisia, 21–23 March 2015; pp. 1–6.
16. Bose, P.; Guo, H.; Kranakis, E.; Maheshwari, A.; Morin, P.; Morrison, J.; Smid, M.; Tang, Y. On the false-positive rate of Bloom filters. *Inf. Process. Lett.* **2008**, *108*, 210–213. [CrossRef]
17. Christensen, K.; Roginsky, A.; Jimeno, M. A new analysis of the false positive rate of a Bloom filter. *Inf. Process. Lett.* **2010**, *110*, 944–949. [CrossRef]
18. Rottenstreich, O.; Keslassy, I. The bloom paradox: When not to use a Bloom filter. *IEEE/ACM Trans. Netw.* **2015**, *23*, 703–716. [CrossRef]
19. Lim, H.; Lee, J.; Yim, C. Complement Bloom filter for Identifying True Positiveness of a Bloom filter. *IEEE Commun. Lett.* **2015**, *19*. [CrossRef]
20. Fan, L.; Cao, P.; Almeida, J. Summary cache: A scalable wide-area Web cache sharing protocol. *Trans. Netw. IEEE/ACM* **2000**, *8*, 281–293. [CrossRef]
21. Rottenstreich, O.; Kanizo, Y.; Keslassy, I. The variable-increment counting Bloom filter. *IEEE/ACM Trans. Netw.* **2014**, *22*, 1092–1105. [CrossRef]
22. Qian, J.; Zhu, Q.; Wang, Y. Bloom filter based associative deletion. *IEEE Trans. Parallel Distrib. Syst.* **2014**, *25*, 1986–1998. [CrossRef]
23. Yang, T.; Liu, A.X.; Shahzad, M.; Zhong, Y.; Fu, Q.; Li, Z.; Xie, G.; Li, X. A shifting bloom filter framework for set queries. *Proc. VLDB Endow.* **2016**, *9*, 408–419. [CrossRef]
24. Almeida, P.S.; Baquero, C.; Preguiça, N.; Hutchison, D. Scalable bloom filters. *Inf. Process. Lett.* **2007**, *101*, 255–261. [CrossRef]
25. Guo, D.; Wu, J.; Chen, H.; Yuan, Y.; Luo, X. The Dynamic Bloom filters. *IEEE Trans. Knowl. Data Eng.* **2010**, *22*, 120–133. [CrossRef]
26. Liu, Y.; Ge, X.; Du, D.H.C.; Huang, X. Par-BF: A Parallel Partitioned Bloom filter for Dynamic Data Sets. *Int. J. High Perform. Comput. Appl.* **2015**, *30*, 1–8. [CrossRef]
27. Xiao, M.Z.; Dai, Y.F.; Li, X.M. Split bloom filter. *Acta Electron. Sin.* **2004**, *32*, 241–245.
28. Xiao, B.; Hua, Y. Using Parallel Bloom filters for Multiattribute Representation on Network Services. *IEEE Trans. Parallel Distrib. Syst.* **2010**, *21*, 20–32. [CrossRef]
29. Kirsch, A.; Mitzenmacher, M. Distance-sensitive bloom filters. In Proceedings of the 8th Workshop on Algorithm Engineering and Experiments/3rd Workshop on Analytic Algorithms and Combinatorics, Miami, FL, USA, 22–26 January 2006; pp. 41–50.
30. Hua, Y.; Xiao, B.; Veeravalli, B.; Feng, D. Locality-Sensitive Bloom filter for Approximate Membership Query. *IEEE Trans. Comput.* **2012**, *61*, 817–830. [CrossRef]
31. Qian, J.; Zhu, Q.; Chen, H. Multi-Granularity Locality-Sensitive Bloom filter. *IEEE Trans. Comput.* **2015**, *64*, 3500–3514. [CrossRef]
32. Qian, J.; Zhu, Q.; Chen, H. Integer-Granularity Locality-Sensitive Bloom filter. *IEEE Trans. Comput.* **2016**, *20*, 2125–2128. [CrossRef]
33. Wu, W.; Wu, S.; Zhang, L.; Zou, J.; Dong, L. LHash: A Light weight Hash Function. In Proceedings of the International Conference on Information Security and Cryptology, Guangzhou, China, 27 November 2013; pp. 291–308.
34. Charles, D.X.; Lauter, K.E.; Goren, E.Z. Cryptographic Hash Functions from Expander Graphs. *J. Cryptol.* **2009**, *22*, 93–113. [CrossRef]

35. Fagin, R.; Kumar, R.; Sivakumar, D. Efficient similarity search and classification via rank aggregation. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, CA, USA, 9–12 June 2003; pp. 301–312.
36. Datasets for Approximate Nearest Neighbor Search. Available online: http://corpus-texmex.irisa.fr/ (accessed on 19 June 2018).