Single Hash: use one hash function to build faster hash based data structures

Xiangyang Gou[1], Chenxingyu Zhao[1], Tong Yang[1,2], Lei Zou[1], Yang Zhou[1], Yibo Yan[1], Xiaoming Li[1], Bin Cui[1]
Peking University, China[1]
Collaborative Innovation Center of High Performance Computing, NUDT, China[2]

***Abstract*—With the scale of data to store or monitor in nowadays network constantly increasing, hash based data structures are more and more widely used because of their high memory efficiency and high speed. Most of them, like Bloom filters, sketches and d-left hash tables use more than one hash function. Furthermore, in order to achieve good randomicity, the hash functions used, like MD5 and SHA1, are very complicated and consume a lot of CPU cycles to carry out. As a consequence, the implementation of these hash functions will be time-consuming. In order to address this issue, we propose Single Hash technique in this paper. It is based on the observation that the hash functions we use produce 32-bit or 64-bit values which have much bigger value ranges than that we need in practice. We usually have to carry out modular operation to map the hash results into a smaller range in the data structures listed above. In this procedure, information carried by the high bits may be discarded. For example, if in a Bloom filter the length of the bit array is 2^{20} while the hash functions we use are 32-bit hash functions, there are 12 bits in the results of the hash functions discarded in the procedure of modular. We can use these bits to produce more hash values. Therefore, we propose to use a few bit operations to make full use of the information produced by one hash function and generate multiple hash values which can be used in these data structures. Single Hash technique can be applied to most of the hash based data structures. It can significantly improve their speed, because instead of carrying out multiple hash functions, we only need to compute one hash function and a few simple operations (*e.g.*, bit shift and XOR). Other aspects of performance, like memory efficiency and accuracy of these data structures will not be influenced by Single Hash technique. In this paper, we apply it to three kinds of classic hash based data structures, *i.e.*, Bloom filters, CM sketches and d-left hash tables as case studies, and evaluate their performance with both mathematical analysis and extensive experiments. We make all our codes open source on Github.**

I. Introduction

A. Background and Motivation

Nowadays with the rapid development of network, the demand for fast storage, query and monitoring of big data is constantly increasing. Therefore, hash based data structures are being more and more widely used due to their high speed and high memory efficiency. These data structures can be divided into two kinds. The first kind supports accurate queries, and the best known one is the hash table. Hash tables use hash functions to map items into different addresses in the table and use techniques like linked lists to solve hash collisions (the hash collision means different items are mapped into the same address). They do not have errors but as a cost consume a lot of memory when the scale of data is large, and the time overhead in the worst case is O(n). Hash tables have been widely used in social network [1], database indexing [2], packet forwarding [3], [4] and so on. Based on classic hash tables, d-left hash technique [5] is applied to make the items distributed more uniformly. It can make more items stay in the table rather than stored in the linked lists. It can also decrement the length of the linked lists. As a result, d-left hash technique improves both speed and memory efficiency of hash tables. The d-left hash table divides the table into k sections where k is a parameter, and maps an item into k positions with k independent hash functions, one in each section, and stores it in the left most empty position. If all the positions have been occupied by other items, it stores the item in the left most position with the smallest depth, which means the number of items already stored in the table or in the linked list at this position. The second kind of the hash based data structures are the approximate query data structures such as Bloom filters [6] and sketches. These data structures achieve a much smaller memory footprint and constant O(1) time overhead at the cost of a little error rate. A bloom filter is made up of a bit array and several hash functions. It can determine whether an item belongs to a set. It maps each item into several bits and uses these bits to represent the presence of the item. Bloom filters are widely used in several fields of computer science, such as web caching [7], IP lookup [8], P2P networks [9], *etc*. Sketches extend the bits in Bloom filters into counters and use counters to record the frequencies of items in a set. There are many different schemes of sketches, including some well known ones like CM sketches [10] and CU sketches [11]. Sketches have been playing an important role in data stream processing like finding frequent items [12], identifying heavy hitters [13], tracking flows in network traffic [14], [15] and other fields [16], [17].

Despite of their advantages, Bloom filters, sketches, and d-left hash tables all have high hash computation overhead, as multiple hash functions are needed in these data structures, and hash functions with good performance are usually very complicated. The computation of hash functions can consume

∗Corresponding author: Tong Yang (Email: yang.tong@pku.edu.cn). This work is done by Xiangyang Gou, Chenxingyu Zhao, Yang Zhou and Yibo Yan under the guidance of their mentor: Tong Yang. This work is partially supported by Primary Research & Development Plan of China (2016YFB1000304), National Basic Research Program of China (2014CB340400), NSFC (61472009, 61672061), the Open Project Funding of CAS Key Lab of Network Data Science and Technology, Institute of Computing Technology, Chinese Academy of Science.

a lot of CPU cycles and become the bottleneck of the system performance. Therefore, we propose Single Hash technique, which reduces the number of hash functions we need to one. It significantly improves the speed of these hash based data structures, while keeping the accuracy unchanged. It can be applied to most of the data structures using multiple hash functions and improve their performance.

B. Proposed Approach

We observe that the hash functions produce 32-bit or 64-bit results, in other words, results with maximum values up to 2^{32} or 2^{64}. However, the range of random values we actually need when implementing hash based data structures, which is usually based on the length of the data structure or the length of each section, is much smaller. When we map the results of hash functions into this smaller range with modular operation, the information carried by the high bits is discarded. Based on this observation, we propose to make full use of the information produced by one hash function with bit operations to generate multiple hash values. The formula to produce the i_{th} hash value for key x is $H_i(x) = ((h(x) >> 16) \oplus (h(x) << i))\%w$, where $h(.)$ is the hash function we use, and the range we need is $0 \sim w-1$. In this way, we only need to compute one hash function and a few simple operations to get the multiple hash values we need instead of computing multiple complicated hash functions. Therefore, the speed is significantly improved. Notice that the operations we use in Single Hash are bit operations like shifting and XOR, which are much faster than addition or multiplication. This is also meant to further improve the speed.

Single Hash technique can be applied to most of the data structures that use multiple hash functions, like Bloom filters [6], COMB [18], CM sketches [10], CU sketches [11], TCM [19], d-left hash tables [5] and so on. In this paper, we apply it to three most famous and representative data structures as case studies, *i.e.*, Bloom filters, CM sketches and d-left hash tables. We carry out both mathematical analysis and extensive experiments to evaluate the performance of our technique. Experimental results show that Single Hash outperforms prior art significantly in speed while keeping the same accuracy.

C. Key Contributions

The key contributions of this paper are as following:

1. We propose Single Hash technique, which uses one hash function to produce multiple hash values that can be used in most of the hash based data structures. It improves the speed of these data structures significantly while keeping the accuracy unchanged.

2. We apply Single Hash technique to Bloom filters, CM sketches and d-left hash tables as case studies, and carry out mathematical analysis and extensive experiments to show that our Single Hash technique outperforms the state-of-the-art.

II. Related Work and Backgrounds

A. Less Hashing

Less Hashing [20] is a well-known technique to improve the Bloom filter and its variants. It uses two hash functions $h_1(.)$ and $h_2(.)$ to simulate k hash values by computing $g_i(x) = h_1(x) + i \times h_2(x)$ $(1 \leqslant i \leqslant k)$. Notice that $g_i(x)$ also has to be mapped to range $0 \sim m-1$ to be used as an address, where m is the length of the bit array. Thus one additional modular operation will be needed in implementation. Less Hashing reduces the number of hash functions needed in implementing a Bloom filter from a variant k which can be larger than 10 into a constant value 2, improving the speed while without influencing its accuracy. It has proved that a Bloom filter does not necessarily need k independent hash functions. Based on this conclusion, we notice the waste of information in the modular operation and propose Single Hash technique. Compared to Less Hashing, Single Hash reduces the number of hash functions to one and uses fast bit operations to replace the time consuming addition and multiplication operations, leading to further improvements in speed. It can be applied to most of the hash based data structures. In this paper, we apply it to the following three kinds of data structures as case studies.

B. Bloom filters

A Bloom filter [6] represents a set S of n items from a universe U using an array of m bits, denoted by $B[1]...B[m]$, which are all set to 0 initially. The filter uses k independent hash functions $h_1, h_2...h_k$ that maps each item to a random number uniformly in the range $[0, m-1]$. (This assumption is needed for the false positive rate derivation of Bloom filters). For each item $x \in S$, the bits $B[h_i(x)]$ are set to 1 for $1 \leqslant i \leqslant k$ (a bit can be set to 1 multiple times). To answer a query to check if y is in S, we check whether all $B[h_i(y)]$ $(1 \leqslant i \leqslant k)$ are 1. If not, y is definitely not a member of S. If all $B[h_i(y)]$ are set to 1, y is probably in S, because a Bloom filter may yield false positives. The probability of a false positive has been well studied in the literature [6], and thus we only show the formula of false positive rate f below.

$$\begin{aligned} f &= \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k \\ &= \left(1 - e^{-\frac{nk}{m}}\right)^k \end{aligned}$$

The value of f is minimized as $\left(\frac{1}{2}\right)^k$ when $k = ln2 \times \left(\frac{m}{n}\right)$. In practice, k must be an integer.

The Bloom filter also has partition schemes. The partition version of a Bloom filter is to divide the bit array into k sections if there are k hash functions. The i_{th} $(1 \leqslant i \leqslant k)$ hash function maps an incoming item to a value h_i in range $0 \sim \frac{m}{k}$ where m is the total number of bits in the bit array. Then we set the h_{ith} bit in the i_th section to 1. When querying, we check all the k corresponding bits for an item in the k sections, and if they are all 1 we report yes otherwise we give a negative report. The partition version of the Bloom filter has the same accuracy as the non-partition version when the number of inserting items is big enough: if we represent the number of inserted items with n, the probability that an item z which has not been inserted has hash collisions in a section is

$1-\left(1-\frac{1}{m/k}\right)^n$, and the false positive rate is the probability that z has hash collisions in all the k independent sections $\left(1-\left(1-\frac{1}{m/k}\right)^n\right)^k \approx \left(1-e^{\frac{kn}{m}}\right)^k$, exactly the same as the false positive rate of the non-partition Bloom filters stated above.

Bloom filters are originally designed for membership queries in a set. Recent years many variants of Bloom filters have been proposed to meet the requirements in different applications. Like Bloomier filters [21], Dynamic count filters [22], and those support multiple sets membership queries including COMB [18], shifting Bloom filter [23], [24] and iSet [25]. They also use multiple hash functions, and some of them even use much more hash functions than Bloom filters. Our single Hash technique can also be applied to them, but in this paper we only use the most representative one, Bloom filter as an example to keep the paper reasonably concise.

C. Sketches

There are many kinds of sketches. Most of them are designed to store the frequencies of items in a set. Among them the most widely used one is the CM sketch [10]. The CM sketch has d arrays. Each array is relative with an independent hash function $h_j(x)$ $1 \leqslant j \leqslant d$. When inserting an item x, let $\forall j(1 \leqslant j \leqslant d)$ $count[j, h_j(x)] = count[j, h_j(x)] + 1$. Similarly, when deleting the item, let $\forall j(1 \leqslant j \leqslant d)$ $count[j, h_j(x)] = count[j, h_j(x)] - 1$. $count[j, h_j(x)]$ means the counter mapped by $h_j(x)$ in the j_{th} array. When querying an item x, the CM sketch carries out d hash functions and gets all the corresponding counters. Then it selects the minimum one as the frequency. The CM sketch only has over-estimations. Other kinds of sketches usually have similar data structures but different update and query strategies, such as CU sketches [11], Count sketches [26], Pyramid sketches [27] and Cold Filter [28]. Some sketches are specially designed for certain fields, like gSketch [29] and TCM [19] for graph streams. In this paper we focus on CM sketches as a representor of this kind of data structures.

D. D-left Hash tables

The d-left hash table [5] splits the table into k sections, with each section associated with a hash function $h_i(.)$ $(1 \leqslant i \leqslant k)$ and uses linked lists to deal with hash collisions, which means if there are multiple items mapped to the same position in the table, only the first one is stored in the table, while the others will be stored in a linked list at this position. It defines the depth of each position with the number of items that have already been stored in the table or the linked list at this position. When inserting an item x, it computes the k hash functions and finds the $h_i(x)_{th}$ position in the i_{th} section for all i $(1 \leqslant i \leqslant k)$ and check the k positions. It finds the position with the smallest depth and inserts the item in it. If there are several positions with the smallest depth, it inserts the item into the left most one. In this way, it makes the items distributed more uniformly in the hash table and reduces both the length and the number of the linked lists. In other words, compared to classic hash tables, more items will stay in the table rather than the linked lists in d-left hash tables. This leads to better memory efficiency and lower time overhead in the worst case.

III. Algorithm

A. Original Version

We observe that the hash functions we use nowadays generate 32-bit or 64-bit results, in other words, results with a maximum value up to 2^{32} or 2^{64}. However, the range we actually need when implementing hash based data structures is much smaller. Take the CM sketch as an example: consider that the memory we use in the implementation of a CM sketch is $256K$ Bytes, and the size of the counters is no less than 2 Bytes, which means there are about $128K$ counters. Notice that there are several arrays in a CM sketch. Therefore, the length of each array, *i.e.*, the number of counters in each array, is within 2^{18}. This means when we compute modular operations to map the results of hash functions into addresses in the arrays, a lot of information carried by the high bits of the hash results is discarded. Based on this observation, we deem it practical to reduce hash calculation cost in these data structures by making full use of the information produced by hash functions and propose Single Hash technique. It can be applied to most of the data structures which use multiple hash functions and reduce the hash computation cost.

The basic idea of Single Hash technique is to make full use of the randomness of hash computation results by shifting operations. We use $h(.)$ to represent the hash function we use, and use k to represent the number of random values we need to generate. The range we need to map the items into is from 0 to $w-1$. We assume that $h(.)$ generate 32-bit results. In the original version, when inserting an item with key x, after computing hash function $h(x)$, we put it into the following formula to get the i_{th} random value $H_i(x)$:

$$H_i(x) = (((h(x) >> 16)\,\%w) \oplus ((h(x)\,\%w) << i))\,\%w$$

In the formula, we use $h(x) >> 16$ to get the high bits of the result of the hash function, and use the remainder of dividing it with w as the first parameter p_1. $h(x)\,\%w$ is the second parameter p_2. If we use $h(x)_l$ to represent the low 16 bits of $h(x)$, in other words, $h(x)\,\%65536$, we can see that $p_2 = \left((p_1 \times 2^{16}) + h(x)_l\right)\%w$. Because $h(x)_l$ and p_1 are independent with each other, p_2 is always uniformly distributed in range $0 \sim w-1$ whatever p_1 is. Therefore, p_1 and p_2 are also independent. To generate different values, we let $p_{2i} = p_2 << i$ take part in the XOR operation with p_1 to get the i_{th} random value.

When the hash function we use is 64-bit, change 16 in the formula into 32, and the performance will be better as there are more bits to use.

B. Mathematical Analysis

In this section, we give a mathematical analysis of the Single Hash technique. Consider the situation that there are a set S with n items in it, and for each item x we need to

compute k random values $\{H_i(x) \mid 1 \leqslant i \leqslant k\}$ in range $0 \sim w-1$. Given an item z which is not a member of S, we want to calculate the probability of event α that for every i in range $1 \sim k$, there is at least one x_i in set S that satisfies $H_i(x_i) = H_i(z)$. This probability is very important, because in sketches, it is the probability that the item z collides with other items in all the arrays and we can not get its correct frequency, in other words, the error rate; in a d-left hash table, it is the probability that a key-value pair has to be stored in a linked list; and in a partition Bloom filter, it is the false positive rate. It is the indication to show how well the random values we generate work. If we use k independent hash functions to generate k random values, $Pr(\alpha)$ is easy to get. Because each hash function maps the item to range $0 \sim w-1$ uniformly, we can see that the probability that z collides with another item in a hash function is $\frac{1}{w}$. The probability that it collides with at least one of the n items is $1-\left(1-\frac{1}{w}\right)^n$. Because the k hash functions are independent, the probability that z suffers hash collisions in all the k hash functions is

$$Pr(\alpha) = \left(1-\left(1-\frac{1}{w}\right)^n\right)^k = \left(1-e^{-\frac{n}{w}}\right)^k$$

Because in the hash based data structures the length of arrays always has a linear correlation with the number of inserted items n, we can represent $\frac{n}{w}$ with a constant c. Therefore, we have:

$$Pr(\alpha) = \left(1-e^{-c}\right)^k$$

Now we are going to prove that using Single Hash technique $Pr(\alpha)$ will be the same. Here we assume the length of each section in w is a prime, but experimental results show that even if w is not a prime, the result will be nearly the same. We use p_1 to represent $(h(x) >> 16)\,\%w$, and p_2 to represent $h(x)\,\%w$. As stated above, p_1 and p_2 are independent. First we prove the following theorem:

Theorem 1: For two independent variant p_1 and p_2, an equation set with the following form has only one set of solution:

$$\begin{cases} (p_1 << r) \oplus p_2 = A_1 \\ (p_1 << t) \oplus p_2 = A_2 \end{cases} \quad (1)$$

Where A_1, A_2, r and t are all constants and $t < r$.
We let the two equations XOR with each other,and get

$$((p_1 << r) \oplus (p_1 << t)) \oplus (p_2 \oplus p_2) = A_1 \oplus A_2$$

which can be transformed into:

$$p_1 \oplus (p_1 << (r-t)) = (A_1 \oplus A_2) >> t$$

We can get p_1 with the this equation. We assume that there are l bits in the binary form of p_1 with no padding zeros, which means $2^{l-1} \leqslant p_1 < 2^l$. Then the i_{th} $(1 \leqslant i \leqslant l)$ bit b_i can be calculated as follows, where B_i is the i_{th} bit of the constant $(A_1 \oplus A_2) >> t$:

- $b_i = B_i,\ 1 \leqslant i \leqslant (r-t)$.
- $b_i = B_i \oplus b_{i-(r-t)},\ (r-t+1) \leqslant i \leqslant l$

After getting p_1, we put it into the original equation set and calculate p_2. Therefore, this equation set has exactly one set of solution. With theorem 1 we further prove the following theorem:

Theorem 2: For each item x in set S and z, there will be exactly one of the following three cases happening:

1) $H_i(x) = H_i(z)$ for only one i in range $1 \sim k$.
2) $H_i(x) \neq H_i(z)$ for all i in range $1 \sim k$.
3) $H_i(x) = H_i(z)$ for all i in range $1 \sim k$.

To prove this theorem, we need to prove that if there are integers r and t in range $1 \sim k$ which meet the requirements $r \neq t$ and $H_r(x) = H_r(z)$, $H_t(x) = H_t(z)$, then we have $\forall i\ (1 \leqslant i \leqslant k),\ H_i(x) = H_i(z)$. Because p is a very big prime and XOR is the binary form of addition without carries and shares similar characteristic like addition, we can use the properties of prime numbers and get:

$$\begin{aligned} &((h(x) >> 16)\,\%w) \oplus ((h(x)\,\%w) << r) \\ =&((h(z) >> 16)\,\%w) \oplus ((h(z)\,\%w) << r) \\ &((h(x) >> 16)\,\%w) \oplus ((h(x)\,\%w) << t) \\ =&((h(z) >> 16)\,\%w) \oplus ((h(z)\,\%w) << t) \end{aligned}$$

According to theorem 1, we have:

$$\begin{aligned} (h(x) >> 16)\,\%w &= (h(z) >> 16)\,\%w \\ h(x)\,\%w &= h(z)\,\%w \end{aligned}$$

Therefore, for all i in range $1 \sim k$, we have $H_i(x) = H_i(z)$.

Theorem 3: In the Single Hash technique, we still have $Pr(\alpha) = (1-e^{-c})^k$.

We consider the event ϵ that there is at least one x in set S that satisfies the second case in theorem 2 . Because $(h(x) >> 16)\,\%w$ and $h(x)\,\%w$ are independent with each other and uniformly distributed in range $0 \sim w-1$, we can see that:

$$\begin{aligned} Pr(\epsilon) &= 1-\left(1-\frac{1}{w^2}\right)^n \\ &= 1-\sqrt[n]{\left(1-\frac{1}{w^2}\right)^{n^2}} \\ &= 1-\sqrt[n]{e^{\frac{n^2}{w^2}}} \\ &= 1-e^{\frac{c^2}{n}} \end{aligned}$$

Because c is a small constant, while n can be very large in the data structures, we have

$$Pr(\epsilon) = o(1)$$

In other words, in applications ϵ almost does not happen. Because obviously $Pr(\alpha \mid \epsilon) = 1$, we have

$$\begin{aligned} Pr(\alpha) &= Pr(\alpha \mid \epsilon) \times Pr(\epsilon) + Pr(\alpha \mid \neg\epsilon) \times Pr(\neg\epsilon) \\ &= Pr(\epsilon) + Pr(\alpha \mid \neg\epsilon) \times Pr(\neg\epsilon) \\ &= o(1) + Pr(\alpha \mid \neg\epsilon) \times (1-o(1)) \end{aligned}$$

Therefore, we can take $Pr(\alpha \mid \neg\epsilon)$ as the asymptotic value of $Pr(\alpha)$. Conditioned on $\neg\epsilon$, for all the x in set S, pair

$P(x) = ((h(x) >> 16)\,\%w,\ h(x)\,\%w)$ should be uniformly distributed in $G = \{0, 2, ...w-1\}^2 - \{P(z)\}$. When $P(x)$ is in the following set with $w-1$ items:

$$\{(a,b) \mid (a,b) \in G,\ a = H_i(z) \oplus (b << i),\ b \neq h(z)\,\%w\}$$

we have $H_i(x) = H_i(z)$. This becomes a variant of balls and bins problem. There are k bins, and n balls. Each ball has probability $\frac{k(w-1)}{w^2-1} = \frac{k}{w+1}$ to be stored in the bins, otherwise discarded. When stored, the bin to put the ball in is selected randomly. The number of balls that are not discarded has binary distribution $Bin\left(n, \frac{k}{w+1}\right) \approx Po(ck)$ when $Po(.)$ denote Poisson distribution. Because each ball not discarded will be stored in the bins randomly, according to a standard property of Poisson variables, the joint distribution of the number of balls in the k bins is the same as the joint distribution of k independent $Po(c)$ variables. The probability that there is at least one ball in each bin is:

$$Pr(Po(c) > 0)^k = \left(1 - e^{-c}\right)^k$$

This is the asymptotic value of $Pr(\alpha)$.

C. Further Discussions and Improvements

In implementations, we can use a few improvements to make the Single Hash easier to implement. First, because w is usually very large and the influence of congruence will be slight. Therefore, we do not have to use a prime as w. Experiments show that when we use numbers like 100000 or 50000 as w there is still little difference in the performance. Second, we notice that the modular operations computed upon $h(x) >> 16$ and $h(x)$ can also be removed without influence on the randomness of the hash values, and this will improve the speed a lot, as modular operations are very time-consuming. The final version of the formula we use to produce the i_{th} hash value $H_i(x)$ is:

$$H_i(x) = ((h(x) >> 16) \oplus (h(x) << i))\,\%w$$

Although a rigorous mathematical proof of the randomness of this formula is hard to give, we apply it to partition and non-partition Bloom filters, CM sketches and d-left hash tables and carry out extensive experiments to show that it works quite well.

D. Applications

Single Hash technique can be applied to almost all the data structures which use multiple hash functions. In this paper, we apply it to three classic ones as case studies: Bloom filters, CM sketches, and d-left hash tables. We assume that the total length of the data structure is m, and it produces k hash values for each item. When applying Single Hash to the CM sketch, the d-left hash table, or the partition Bloom filter, we just replace the k independent hash functions in the classic version with hash values produced with one hash function $h(.)$: $H_i(x) = ((h(x) >> 16) \oplus (h(x) << i))\,\%\frac{m}{k}$, $(1 \leqslant i \leqslant k)$, and the other operations will be the same. When applying it to a non-partition Bloom filter, the formula to produce the i_{th} hash value for item x will be $H_i(x) = ((h(x) >> 16) \oplus (h(x) << i))\,\%m$, and the other operations are the same as a classic non-partition Bloom filter. the mathematical analysis for Bloom filters without partitions is different from above and controversial because addresses produced by different hash functions are not separated, in other words, $h_i(x)$ and $h_j(y)$ may indicate the same bit in the array ($h_i(.)$ and $h_j(.)$ are different hash functions, and x and y are different items), while in partition Bloom filters, CM sketches and d-left hash tables they are in different sections. But experiments show that the performance of the Single Hash scheme is still out-standing.

IV. Experimental Results

In this section, we apply Single Hash technique to three well known hash based data structures: Bloom filters, CM sketches, and d-left hash tables. For Bloom filters and CM sketches, we carried out extensive experiments to compare the accuracy and speed of the classic scheme, the Less Hashing scheme, and the Single Hash scheme. For the d-left hash table, as it is an accurate data structure with no error, we compare the number and length of linked lists and speed of the three schemes. Note that the Less Hashing has only been applied to Bloom filters in [20], we extend it to the other two data structures. The hash functions we use are 32-bit, and the formula we use in single hash is the final version, namely $H_i = ((h(x) >> 16) \bigoplus (h(x) << i))\,\%w$.

A. Experimental Setup

We carry out experiments with a set of strings generated by Pseudo random number generation algorithm. The length of each string is 20 Bytes. There are totally 1400769 strings in the set, and the number of unique items is 84191, which will be used in the experiments of Bloom filters and d-left hash tables. The data obeys uniform distribution over different keys. The host we implement experiments is a laptop which has an Intel Core i5-6300HQ 2.30 GHZ CPU with 4 cores, and 8 GB memory. It runs Ubuntu 15.04. We implement all the codes with c++ and build them with GCC 5.2.1 and -O3 option.

B. Bloom Filters

We fix the number of unique items to insert into the Bloom filters as $n = 40k$ and fix the length of bit array as $m = 400K$, and change the number of hash functions we use, which is represented by k. We use the false positive rate to evaluate the accuracy of a Bloom filter, which means the probability of a Bloom filter to report yes when querying an item not in it. To get the estimation of false positive rate, we query $n = 40k$ items not in the Bloom filter, and record the times it reports yes as t. Then we calculate the false positive rate f as $f = \frac{t}{n}$. We tested both partition version and non-partition version. The results are shown in Figure 1 and Figure 2.

Figure 1 and Figure 2 show that the false positive rates of both partition and non-partition versions of Single Hash, Less Hashing and classic Bloom filters are always nearly the

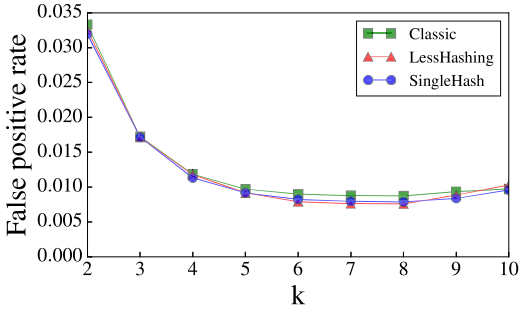

Fig. 1. False positive rate in non-partition Bloom filters.

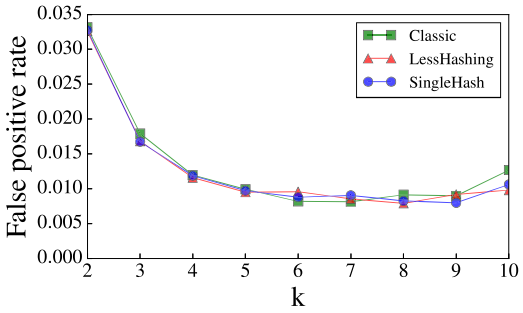

Fig. 2. False positive rate in partition Bloom filters.

same. The false positive rates of the three schemes first drop and then increase when k is constantly increasing. Because as stated in section II, Bloom filters have the best performance when $k = ln2\frac{m}{n}$. The value of the equation is about 6 in our experiments, which can be computed with the value of n and m given above. It is also accordance with the lines in figures. When k is smaller than 6, increasing k will increase the chance that an item does not have hash collisions in at least one of the k positions. But when k becomes too big, the Bloom filters become crowded and the accuracy decreases.

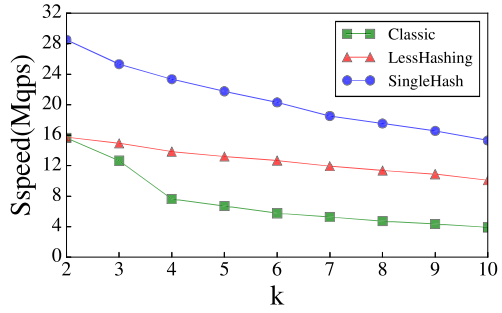

Fig. 3. Speed of different schemes of Bloom filters.

Figure 3 shows that when k changes the speed of Single Hash is always much higher than that of the other two schemes, about $1.5 \sim 1.8$ times of the speed of the Less Hashing, and $1.8 \sim 3.9$ times of the speed of the classic Bloom filter. The measurement we use for speed is *Mqps*, *i.e.*, million queries per second. To evaluate the speed of the three schemes, we query $80k$ items, and 50% of them are in the Bloom filters while the others are not. To make the evaluation more accurate, we repeat the experiment 100 times to get the average speed. When k becomes larger, the difference between Less Hashing and Single Hash will not change, but the gap between the speed of these two and the classic Bloom filter will enlarge as more hash functions need to be computed in the classic Bloom filter. Note that there is no difference in the speed of the partition version and the non-partition version of Bloom filters.

C. Sketches

We insert all the 1400769 strings into the three schemes of CM sketches: the Single Hash scheme, the Less Hashing scheme, and the classic one. As stated above, the number of unique items in these records is $n = 84191$, and we set the total number of counters in each CM sketch fixed to $m = 40K$ and change the number of arrays k. We evaluate the accuracy of the three schemes with average relative error (ARE) and Correct rate (CR). We use f_e to represent the frequency of an item report by a CM sketch, and f_r to report the correct frequency. Because there are only over-estimations in CM sketches, f_e is always no smaller than f_r. We define relative error $RE = \frac{f_e - f_r}{f_r}$. ARE is the average value of the RE of the n items. CR is defined as $CR = \frac{t_c}{n}$, where t_c means the number of items whose reported frequency is equal to the correct one. The following two figures show the ARE and CR of the three schemes of CM sketches, respectively.

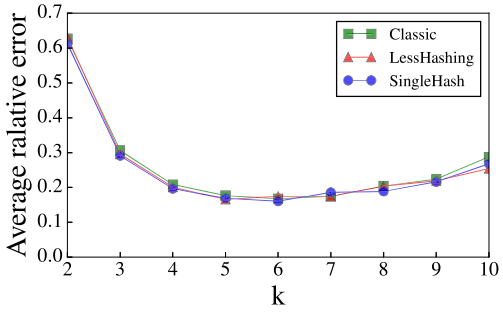

Fig. 4. Average relative error in CM sketches.

Figure 4 and Figure 5 show that the ARE and CR of the three kinds of CM sketches are nearly the same. In other words, these three schemes have the same accuracy. The CM sketch has the similar characteristic as Bloom filers. When k increases, it first becomes more accurate, and then the accuracy decreases because the length of each array decreases and the CM sketch becomes crowded.

Figure 6 shows that when k changes the speed of Single Hash is always much higher than that of the other two schemes, about $2 \sim 2.6$ times of the speed of the Less

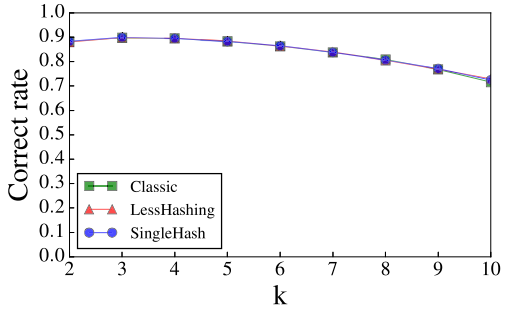

Fig. 5. Correct rate in CM sketches.

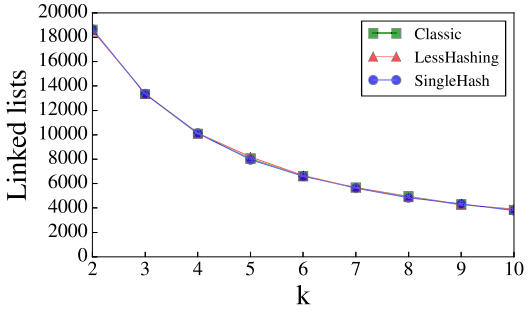

Fig. 7. Number of linked lists in d-left hash tables.

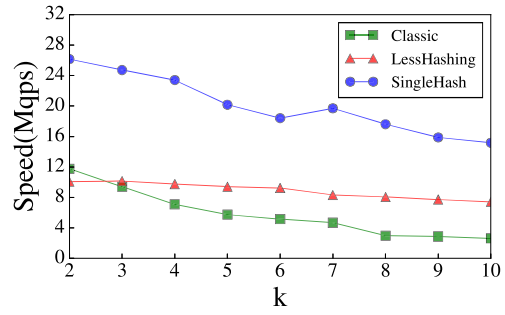

Fig. 6. Speed of different schemes of CM sketches.

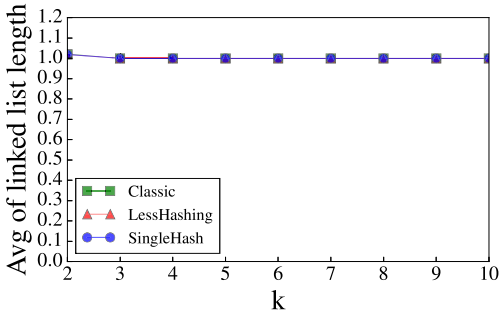

Fig. 8. Average length of linked lists in d-left hash tables.

Hashing, and $2.4 \sim 5$ *times of the speed of the classic CM sketch.* The measurement we use for speed is still *Mqps*. Similar to the experiment in Bloom filters, we repeat the experiment 100 times to get the average speed. When k further increases, the superiority of Single Hash will be more significant.

D. D-left Hash Tables

We insert all the 84191 unique keys with random values into the three schemes of d-left hash tables: Single Hash, Less Hashing, and classic d-left hash tables. The three schemes have the same length $m = 85000$, nearly $1:1$ to the number of items inserted. We change the number of sections k to carry out a series of experiments. Because the d-left hash table is an accurate data structure with no error, we evaluate the performance of different schemes with the number and average length of linked lists, which are important metrics of d-left hash tables. The larger these two values are, the worse a d-left hash table performs. The experimental results are shown in the following figures.

Figure 7 and Figure 8 show that the number of linked lists and their average length in the three kinds of d-left hash tables are nearly the same. Because these three schemes have nearly the same number of linked lists and the average length of linked lists is almost always 1, there are the same number of items inserted into hash tables in the three schemes. In other words, the load factors in those three tables are the same, and the uniformities of the distribution of items in these three schemes are nearly the same. Therefore, we can determine that these three schemes have the same performance.

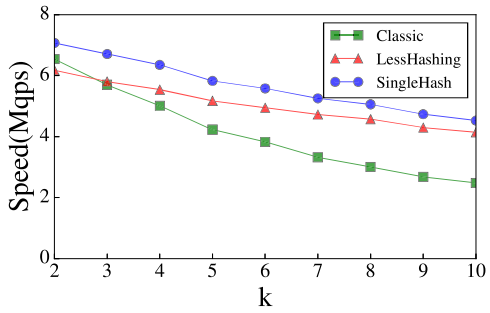

Fig. 9. Speed of different schemes of d-left hash tables.

Figure 6 shows that when k changes the speed of Single Hash is always higher than that of the other two scheme, about 1.1 *times higher than the speed of the Less Hashing, and* $1.1 \sim 1.8$ *times higher than the speed of the classic d-left hash table.* The difference in speed is not that significant because in hash tables we need to traverse linked lists and compare character strings besides computing hash functions. These operations are also time-consuming and they reduce the influence of the decrement of hash computation overhead. But when k becomes larger, the difference in speed between Single Hash scheme and the classic scheme will further

enlarge. The measurement and the method we get the average speed are still the same as the speed experiments above.

V. Conclusion

In this paper, we propose Single Hash technique, which uses only one hash function and a few bit operations to produce multiple hash values. It can be applied to almost all of the hash based data structures and significantly improve their speed while keeping the other metrics unchanged. We apply it to three kinds of well known data structures as an example in this paper: Bloom filters, CM sketches, and d-left hash tables. We carry out mathematical analysis and extensive experiments, and the results show that our Single Hash technique outperforms the state-of-the-art.

References

[1] S. Marti, P. Ganesan, and H. Garcia-Molina, "Dht routing using social links," in *International Conference on Peer-To-Peer Systems*, pp. 100–111, 2004.

[2] M. S. Jensen and R. Pagh, "Optimality in external memory hashing," *Algorithmica*, vol. 52, no. 3, pp. 403–411, 2008.

[3] F. Pong, "Method and system for hash table based routing via table and prefix aggregation," 2011.

[4] A. Broder and M. Mitzenmacher, "Using multiple hash functions to improve ip lookups," in *INFOCOM 2001. Twentieth Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, pp. 1454–1463 vol.3, 2001.

[5] B. Cking, "How asymmetry helps load balancing," in *Symposium on Foundations of Computer Science*, p. 131, 1999.

[6] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[7] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," in *ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 254–265, 1998.

[8] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," *IEEE/ACM Transactions on Networking*, vol. 14, no. 2, pp. 397–409, 2006.

[9] M. E. Locasto, J. J. Parekh, A. D. Keromytis, and S. J. Stolfo, "Towards collaborative security and p2p intrusion detection," in *Information Assurance Workshop, 2005. Iaw '05. Proceedings From the Sixth IEEE Smc*, pp. 333–339, 2005.

[10] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[11] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," *ACM SIGMCOMM CCR*, vol. 32, no. 4, 2002.

[12] G. Cormode and M. Hadjieleftheriou, "Finding frequent items in data streams," *Proceedings of The VLDB Endowment*, vol. 1, no. 2, pp. 1530–1541, 2008.

[13] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund, "Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications," in *Proc. ACM IMC*, 2004.

[14] A. Chen, Y. Jin, J. Cao, and L. E. Li, "Tracking long duration flows in network traffic," in *Proc. IEEE INFOCOM*, 2010.

[15] G. Cormode and M. Garofalakis, "Sketching streams through the net: Distributed approximate query tracking," in *Proceedings of the 31st international conference on Very large data bases*, pp. 13–24, VLDB Endowment, 2005.

[16] B. V. Durme and A. Lall, "Streaming pointwise mutual information," in *Advances in Neural Information Processing Systems*, pp. 1892–1900, 2009.

[17] D. Talbot and M. Osborne, "Smoothed bloom filter language models: Tera-scale lms on the cheap.," in *EMNLP-CoNLL*, pp. 468–476, 2007.

[18] F. Hao, M. Kodialam, T. V. Lakshman, and H. Song, "Fast dynamic multiple-set membership testing using combinatorial bloom filters," *IEEE/ACM Transactions on Networking*, vol. 20, no. 1, pp. 295–304, 2012.

[19] N. Tang, Q. Chen, and P. Mitra, "Graph stream summarization: From big bang to big crunch," in *International Conference on Management of Data*, pp. 1481–1496, 2016.

[20] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better bloom filter," *Random Structures & Algorithms*, vol. 33, no. 2, p. 187C218, 2008.

[21] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The bloomier filter:an efficient data structure for static support lookup tables," in *Fifteenth Acm-Siam Symposium on Discrete Algorithms*, pp. 30–39, 2004.

[22] J. Aguilar-Saborit, P. Trancoso, V. Muntes-Mulero, and J. L. Larriba-Pey, "Dynamic count filters," *Acm Sigmod Record*, vol. 35, no. 1, pp. 26–32, 2006.

[23] T. Yang, A. X. Liu, M. Shahzad, D. Yang, Q. Fu, G. Xie, and X. Li, "A shifting framework for set queries," *IEEE/ACM Transactions on Networking*, vol. PP, no. 99, pp. 1–16, 2017.

[24] T. Yang, A. X. Liu, M. Shahzad, Y. Zhong, Q. Fu, Z. Li, G. Xie, and X. Li, "A shifting bloom filter framework for set queries," *Proceedings of the Vldb Endowment*, vol. 9, no. 5, pp. 408–419, 2016.

[25] Y. Qiao, S. Chen, Z. Mo, and M. Yoon, "When bloom filters are no longer compact: Multi-set membership lookup for network applications," *IEEE/ACM Transactions on Networking*, vol. 24, no. 6, pp. 3326–3339, 2016.

[26] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Automata, Languages and Programming*, Springer, 2002.

[27] T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li, "Pyramid sketch: a sketch framework for frequency estimation of data streams," *Proceedings of the Vldb Endowment*, vol. 10, no. 11, 2017.

[28] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig, "Cold filter: A meta-framework for faster and more accurate stream processing," *Sigmod*, 2018.

[29] P. Zhao, C. C. Aggarwal, and M. Wang, "gsketch: on query estimation in graph streams," *Proceedings of the Vldb Endowment*, vol. 5, no. 3, pp. 193–204, 2012.