Computer Life (CPL) ISSN: 1819-4818
DELIVERING QUALITY SCIENCE TO THE WORLD

A Study of URL Deduplication Policy Based on Eigenvalue Extension Bloom Filter

Mai Jiang [a], Xiang Gao [b] and Zaifeng Mo [c]

Sichuan University of Science and Engineering, Zigong 64300, China;

[a]214694345@qq.com, [b]1033716335@qq.com, [c]1033716335@qq.com

---

Abstract: Lots of applications of the big data always demand the web crawler tools to obtain useful information from the Internet. The web crawler tool needs to avoid the repeated access to the same URL. In this paper, a URL deduplication strategy based on an improved Bloom Filter is proposed. The improved Bloom Filter used in this strategy is the eigenvalue extension Bloom Filter mentioned in this paper [1]. Eigenvalue extension Bloom Filter maintains both high-density compression storage and high-speed lookup performance. It also greatly reduces the false positive rate of Bloom Filter. The implementation algorithm of eigenvalue extension Bloom Filter is simple and easy to operate, and therefore, the URL deduplication strategy on the basis of eigenvalue extension Bloom Filter is very simple and high-efficient.

Keywords: Bloom Filter; Hash function; false positive rate; Web crawler.

---

1. INTRODUCTION

Bloom Filter is a spatially efficient random data structure that applies the bit arrays to represent a set very concisely and to determine whether an element belongs to it or not. The high-efficiency of Bloom Filter comes at a price, that is, when judging whether an element belongs to a set or not, it may mistake elements that do not belong to the set for belong to the set (false positive). Therefore, Bloom Filter is not suitable for those "zero error" applications. In applications that tolerate low error rates, Bloom Filter gets a huge savings in storage space with very few errors.

Bloom Filter represents a set by a bit array. In its initial state, Bloom Filter is a bit array containing m bits, each is set to 0, as shown in Fig.1.

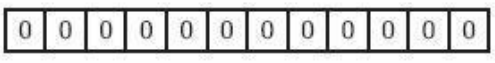

Fig.1. Bit Array

To express S = {x_1, x_2,…,x_n} a set of n elements such as x n}, Bloom Filter uses k independent Hash functions that map each element in the set to the scope of {1,...,m}. For any element x, $h_i(x)$, the position mapped by the i hash function, is set as 1 ($1 \le i \le k$). Note that if a position is set to 1 for multiple times, then it only works in the first time and there will be no effects in the subsequent times. In Fig. 2, k = 3, and two hash functions select the same position (the fifth from the left).

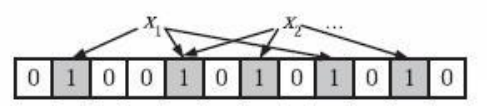

Fig. 2. Elements Are Stored in Bloom Filter

When determining if y belongs to this set, we apply the k hash function to y. If the positions of all $h_i(y)$ are 1 ($1 \le i \le k$), then we think y is the element in the set, otherwise we think y is not the element in the set. In Fig. 3, y_1 is not an element in the set. However, y_2 either belongs to this set, or just happens to be a false positive.

Fig. 3. Look up Elements in Bloom Filter

2. EIGENVALUE EXTENSION BLOOM FILTER AND ITS ANALYSIS

2.1 Description of classical Bloom Filter

An empty bloom filter is a bit array with m binary bits, and each bit array is initialized to 0 and called V of this bit array. Assume set $S = \{x_1, x_2, ..., x_n\}$, and define there are k different hash functions $h_1, h_2, ..., h_k$. A value greater than or equal to 1 and less than or equal to m is calculated after each element is substituted into a hash function, that is, $h_i(x) \to \{1..m\}$, and each element can get k value by substituting k hash functions. Of course, the perfect hash function is one of those hash functions that can randomly and uniformly hash all the elements to n different positions.

To add an element into a bloom filter, namely, to get k value by substituting the element in k hash function, and set the bit corresponding to the k value in the bit array to 1.

To query an element, namely, to determine whether it is in a set, get k value by substituting the element in k hash function. If the bits in this bit array corresponding by k value are all 1, then the element is in the set; if neither one is 1, then the element is not in the set.

Deleting an element is not allowed because the corresponding k bits will be set to 0 in that case, and there may be bits corresponding by other elements, and therefore, this is absolutely forbidden.

The algorithm code is described as follows[2]:

```
BitVector buildBloomFilter(set S, hash functions, integer m){
BitVector V[1..m]=0;
for(i=0;i<n;i++)
for(j=0;j<k;j++)
V[hj(xi)]=1;
return V;
}
```

2.2 The description of eigenvalue extension Bloom Filter algorithm

Although the classical bloom filter has great advantages in space / time complexity, it will inevitably have false positive. In this paper, a new algorithm is proposed to improve the bloom filter so as to reduce the false positive rate.

The algorithm is described as follows[1]:

An empty bloom filter is a bit array with m binary bits, and each bit array is initialized to 0, making bit array *V*, and then a bit array *V* is divided into two sections of {1...m1} and {m1...m}. There is a set $S=\{x_1,x_2,...,x_n\}$ that extracts a number of eigenvalues for each element x in some way, and the function that extracts the eigenvalues is g (), and all selected eigenvalues can be considered as an additional part of the element， and recorded as $x^{'}$. Define k different hash functions $h_1,h_2,...,h_k$ and d different hash functions $f_1,f_2,...,f_d$. For each element x, you can use k hash function $h_1,h_2,...,h_k$ to place the element hash into the k position in the bit array, that is: $h_i(x)\rightarrow\{1..m1\}$. You can also use d difference hash function $f_1,f_2,...,f_d$ and hash $x^{'}$ to d positions, i. e.: $f_j(x^{'})\rightarrow\{m1..m\}$. The storage of the element in the eigenvalue extension Bloom Filter is shown in Fig. 4.

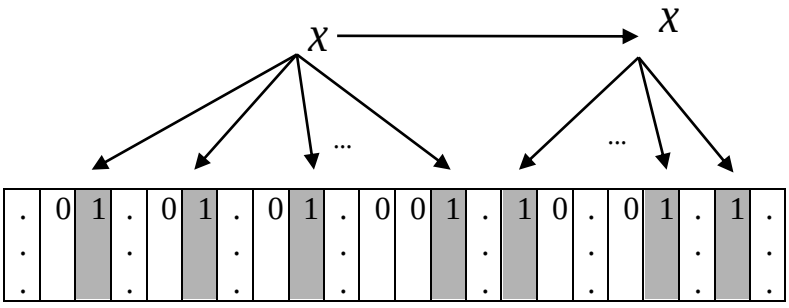

Fig. 4 Diagram of Element Storage Structure of Eigenvalue Extension Bloom Filter

The implementation of the eigenvalue extension Bloom Filter algorithm is described as follows:

BitVector addElementNew(elm, BitVector *V*, hash functions,g()){

for(int i=0;i<k;i++)

V[h_i (elm)] = 1;//0<=h_j(x_i)<=m1

for(int j=0;i<d;j++){

$x_i^{'}$=g (elm) ;

V[f_j ($x_i^{'}$)] = 1;//m1<=f_j($x_i^{'}$)<=m

}

return *V*;

}

2.3 Mathematical analysis

One notable feature of the classic Bloom Filter is that there is an obvious trade-off between the size of the filter and the false positive rate. Obviously, after inserting n elements into a filter with the size of m by using k hash functions, the probability of a particular bit remaining 0 is p_0, as follows:

$$p_0=\left(1-\frac{1}{m}\right)^{kn}\approx 1-e^{-\frac{kn}{m}} \tag{1}$$

Let's assume that the perfect hash function can evenly hash elements in space {1.m}. In practice, good results are achieved in virtue of MD5 and other hash functions[10]. Thus, the probability of false positive (i. e., the probability of all k-bits being placed before 1) is p_{err}, as follows:

$$p_{err} = (1-p_0)^k = \left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^k \approx \left(1-e^{-\frac{kn}{m}}\right)^k = e^{k\ln(1-e^{-kn/m})} \tag{2}$$

Regard the Formula (2) as a function of k $f(k)$, and find the best value, we can get the lowest false positive rate at $k = \ln 2\frac{m}{n}$. When the false positive rate is set, the space-to-element ratio m / n is obtained by Formula (2):

$$\frac{m}{n} = \frac{-k}{\ln\left(1-e^{\frac{\ln p_{err}}{k}}\right)} \tag{3}$$

At the same time, the relationship between Bloom Filter's memory allocation and false positive rate can be obtained by Formulas (2) (3), as shown in Fig. 3. Obviously, as shown in Figs. 5 and 6 above, the better results will not be achieved as the hash functions get more in actual application. Therefore, we can not reduce the false positive rate by increasing the number of hash functions infinitely, and increasing the number of hash functions will not significantly reduce the false positive rate, but increase the overhead of time and space. One feasible method is to reduce the conflict rate by extracting the eigenvalues from the elements and hashing the eigenvalues in virtue of the algorithm which is put forward in this paper.

Of course, based on Bloom Filter's features, we can see that if eigenvalues are also hashed into Bloom Filter, conflict will not be completely avoided. Set the rate of conflict in the Bloom filter for the eigenvalue portion as p^*, as follows:

$$p^* \approx \left(1-e^{-\frac{dn}{m}}\right)^d = e^{d\ln(1-e^{-dn/m})} \tag{4}$$

The false positive rate of the improved algorithm is $p_{N\text{-}err}$, then

$$p_{N\text{-}err} = p_{err} \times p^* \tag{5}$$

$$p_{N-err} = e^{dk\ln(1-e^{-kn/m})\ln(1-e^{-dn/m})} \tag{6}$$

Because $P^* <= 1$, $P_{N\text{-}err} <= P_{err}$。 And if P_{err} is kept constant, $P_{N\text{-}err}$ will decrease linearly with the decrease of P^*. In practice, good eigenvalue extraction can minimize the correlation between similar elements, thus effectively avoiding the conflict rate of similar elements, which will be more significant than the simple increase of hash functions in the classical Bloom Filter algorithm p^*[1].

The time complexity of the improved algorithm is O (k + d), which is still constant, and there is not much increase compared with the time complexity O (k) in the classical Bloom Filter. However, the cost in space is not increased much, so we can add it properly according to the reality when we use it. Generally speaking, the number of hash function d should be set to be smaller than the number of original hash function k. If d/ k = 1/3, then increase the bit array size by about 1/3. It is necessary to explain that, through the above algorithm description, we can also see that the improved algorithm, although it greatly reduces the false positive rate, does not increase the difficulty of implementation of the algorithm, which shows that the improved algorithm is simple and efficient.

3. THE APPLICATION OF URL DUPLICATION DETERMINATION OF EIGENVALUE EXTENSION BLOOM FILTER IN THE WEB CRAWLER PROGRAM

In a web crawler program, you need to determine the duplication of URL collected by the crawler program so as to avoid storing duplicate URL. Of course, the duplication can also be determined by searching database, but its time efficiency is extremely low, and moreover, since the crawler program will continuously and quickly access the URL, it will cause the huge pressure to the database, the resource occupation will also be extremely high[3]. In practice, the improved Bloom Filter proposed in this paper can be used to determine the duplication of the URL.

Suppose that a crawler program collects several URLs, as shown in Table 1 below, each of these elements can be hashed into the area of {1..m1} of a bit array of m bits using the k hash function, as in the classic Bloom Filter algorithm, and meanwhile, the eigenvalues are simultaneously extracted.

Table 1 Element Hash and Eigenvalue Extraction. (The URL is the element, and the special part of the URL is the eigenvalue)

x	value of x	eigenvalues	hash value ($h_i(x) \to \{1..m1\}$)
x_1	http://www.x1x2x3.x4x5x6.x7x8x9.com	x1,x4,x7	$h_1(x_1), h_2(x_1), \ldots, h_j(x_1), \ldots, h_k(x_1)$
x_2	http://www.y1y2y3.y4y5y6.y7y8y9.net	y1,y4,y7	$h_1(x_2), h_2(x_2), \ldots, h_j(x_2), \ldots, h_k(x_2)$
x_3	http://www.z1z2z3.z4z5z6.z7z8z9.org	z1,z4,z7	$h_1(x_3), h_2(x_3), \ldots, h_j(x_3), \ldots, h_k(x_3)$
x_4	http://www.u1u2u3.u4u5u6.u7u8u9.net	u1,u4,u7	$h_1(x_4), h_2(x_4), \ldots, h_j(x_4), \ldots, h_k(x_4)$
x_5	http://www.v1v2v3.v4v5v6.v7v8v9.com	v1,v4,v7	$h_1(x_5), h_2(x_5), \ldots, h_j(x_5), \ldots, h_k(x_5)$
…	…	…	…

For each element's eigenvalue x', a separate d hash function is used to hash to the area of {m1..m}, as shown in Table 2.

Table 2 Hash Eigenvalue Again

x'	vxlue of x'	hash value($f_j(x') \to \{m1..m\}$)
x'_1	x1x4x7	$f_1(x'_1), f_2(x'_1), \ldots, f_j(x'_1), \ldots, f_d(x'_1)$
x'_2	y1y4y7	$f_1(x'_2), f_2(x'_2), \ldots, f_j(x'_2), \ldots, f_d(x'_2)$
x'_3	z1z4z7	$f_1(x'_3), f_2(x'_3), \ldots, f_j(x'_3), \ldots, f_d(x'_3)$
x'_4	u1u4u7	$f_1(x'_4), f_2(x'_4), \ldots, f_j(x'_4), \ldots, f_d(x'_4)$
x'_5	v1v4v7	$f_1(x'_5), f_2(x'_5), \ldots, f_j(x'_5), \ldots, f_d(x'_5)$
...	...	...

In fact, false positive is mostly caused by similar elements, because the hash function used by Bloom Filter ignores small differences between similar elements. For example, the classic hash algorithm Bob Jenkins' Functions is described as follows:

```
uint32_tjenkins_one_at_a_time_hash(unsigned char *key, size_t key_len){
uint32_t hash = 0;
size_t i;
for (i = 0; i < key_len; i++) {
hash += key;
hash += (hash << 10);
hash ^= (hash >> 6);
}
hash += (hash << 3);
hash ^= (hash >> 11);
hash += (hash << 15);
return hash;
}
```

A number of shift operations are used in Bob Jenkins' Functions. The left shift operation may lose the high position, and the right shift operation may lose the low position. Obviously, the shift operation causes some of the data to lose accuracy, which is likely to cause small differences between similar elements to be ignored.

The number of URL obtained by a web crawler in a web site consecutively is called the crawler depth[4]. After designed with the classical Bloom Filter and the eigenvalue extension Bloom Filter respectively, the web crawler program was tested on different websites. It can be found that the deeper the crawler is in the same website, the more similar the URL is, and the false positive rate will increase accordingly[5]. The eigenvalue extension Bloom Filter has better performance than the web crawler designed by the classical Bloom Filter in the aspect of false positive rate. As shown in Fig. 5, when m / n is 32, the crawler program designed by the classical Bloom Filter and the eigenvalue extension Bloom Filter respectively counts the false positive rate at different crawler depths.

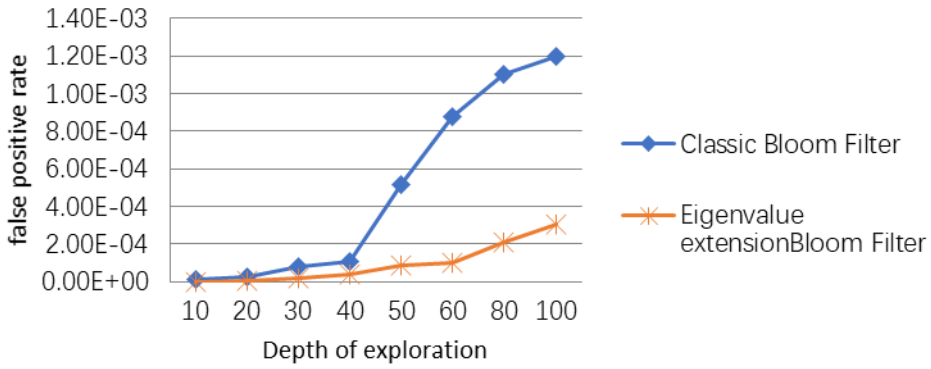

Fig. 5 False Positive Rate Test

The results show that the false positive rate of Bloom Filter is higher when the element similarity is high, and the Bloom Filter can reduce the false positive rate by using eigenvalue extension. In

practice, if the amount of data collected is large, the advantage of the eigenvalue extension Bloom Filter will be more obvious.

4. SUMMARY

In practice, the classical Bloom Filter can't reduce the false positive rate by adding hash function infinitely, but it needs to make appropriate trade-off between space, time and false positive rate. At the same time, increasing the number of hash function largely does not significantly reduce the effect of false positive rate after a certain threshold, but increases the cost of time and space. In fact, too many of the similar elements significantly increases Bloom Filter's false positive rate. The eigenvalue extension Bloom Filter algorithm proposed in this paper significantly reduces the correlation between similar elements. Therefore, this paper proposes the eigenvalue extension Bloom Filter algorithm which can reduce the false positive rate effectively.

At the same time, the improved Bloom Filter algorithm inherits the advantages of time and space efficiency of the classical Bloom Filter. The implementation of the algorithm is also very concise.

Moreover, the application of URL duplication determination of eigenvalue extension Bloom Filter in the web crawler program is also introduced in this paper. Since the crawler programs always go deep on URL in a large Web site, the differences between those URLs tend to be small, and the classic Bloom Filter causes more false positive because the hash function often ignores small errors. In practice, there is a significant effect of the URL duplication determination of eigenvalue extension Bloom Filter in the web crawler program.