Jiang et al. EURASIP Journal on Image and Video Processing (2018) 2018:139
https://doi.org/10.1186/s13640-018-0375-6

EURASIP Journal on Image and Video Processing

RESEARCH Open Access

An improved algorithm based on Bloom filter and its application in bar code recognition and processing

Mai Jiang*, Chunsheng Zhao, Zaifeng Mo and Jing Wen

Abstract

In many cases, databases are incompetent to meet the requirement of the quick query identification and processing of bar codes, such as the automatic sorting system of giant logistics warehouse. Bloom filter can be faster than databases, but its high false positive rate may seriously affect the efficiency of work. Although increasing the width of bit vector and the number of hash functions can reduce the false positive rate, the effect will be not significant after a certain threshold value, and this approach will increase the cost of processing time. So, it could not be increased indefinitely. This paper presents an improved algorithm based on Bloom filter and its application in bar code recognition and processing. The bit vector of Bloom filter is divided into two parts. Every element a_i could be mapped to a part of the bit vector by some hash functions. For each element to amplify the difference by g (), which makes g (a_i) = a^*_i, the a^*_i is mapped to another part of the bit vector by some hash functions too. This algorithm can significantly reduce the false positive rate of the Bloom filter, but does not increase much time and space costs.

Keywords: Bloom filter, Hash function, False positive rate

1 Introduction

In many cases, the bar codes need to be quickly identified and processed in a timely manner. The traditional way is to scan the bar codes and then read the database to identify the bar codes; but on some occasions, such modern logistics warehouse of giant automatic sorting system using databases often cannot satisfy the requirement of bar codes recognition speed. Especially nowadays, with the increasing popularity of online shopping, many logistics systems are processing more and more data, which puts a lot of pressure on the database. The rapid identification of bar codes using Bloom filter is a good option. However, Bloom filter has a defect that there is a false alarm rate, which is a bad situation for the bar code's recognition system that needs to be quickly identified and processed. Even sometimes false positives can lead to a lot of later processing time costs. Although the false positive rate could be reduced by increasing the length of the bit vector of the Bloom filter and adding the number of hash functions, the cost of time and space will also be increased. However, in systems that require quick recognition, the increasing of time and space is often restricted. In addition, the algorithm implementation of various classical hash functions of Bloom filter can easily result in the ignoring of the slight difference of similar inputs, which leads to the improvement of false positive rate. The bar code itself is often very different. In this paper, an improved Bloom filter algorithm is proposed to reduce the false positive rate and ensure the speed of bar code recognition. The algorithm proposed in this paper is to reduce the false positive rate of Bloom filter.

This paper is in the same direction with the author's another paper [1]. However, this paper differs from the reference [1] in the following aspects and improvements. Firstly, this paper analyzes the reasons why the classical hash functions in practical applications tend to ignore the differences of similar elements, which is why similar elements are more likely to cause conflicts. The arguments of previous articles have been based on non-existent perfect hashing functions. Secondly, a new idea is proposed in this paper, that is, the hashing process is carried out after the element has been transformed to solve the problem of high conflict

* Correspondence: 214694345@qq.com
School of Computer Science, Sichuan University of Science and Engineering, Zigong, Sichuan, People's Republic of China

Springer Open
© The Author(s). 2018 Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

rate of Bloom filter caused by approximate input elements. This transformation is the differential amplification transformation. Finally, based on the proposed algorithm, this paper presents a detailed scheme for bar code recognition and processing.

We briefly introduce the results and discussion in Section 2. In this section, the research results are briefly introduced. Bloom filter and its current common usage are discussed. We describe Bloom filters in detail, and we give a hopefully precise picture of space/computing time/error rate trade-offs. An improved algorithm is introduced. In section 3, an application scheme of the algorithm in bar code recognition is introduced. The last section is the conclusion.

2 Results and discussion

An improved Bloom filter is proposed in this paper. This algorithm can effectively reduce the false positive rate of Bloom filter, but it does not increase much space and time cost. Moreover, the algorithm is relatively easy to implement. This paper applies the algorithm to bar code recognition and processing and designs a specific scheme. According to the statistical effect of the test, the algorithm reduces the false positive rate and achieves better test results. At the same time, the bar code's automatic identification scheme designed in this paper has a favorable processing speed.

2.1 Bloom filters

Bloom filter [2] was proposed by Burton Howard Bloom in 1970. It is a compact data structures for probabilistic representation of a set in order to support membership queries (i.e., queries that ask: "Is element X in set Y?"). This compact representation is the payoff for allowing a small rate of false positives in membership queries. That is, queries might incorrectly recognize an element as a member of the set.

If it is needed to determine whether an element is in a collection, the traditional solution is to save all the elements and then compare them. The data structures such as linked lists, trees, and so on, are the same idea. But with the increasing of the elements in the collection, the storage space that they need becomes larger and larger, and the retrieval speed becomes slower and slower (O(n),O(logn)).

Bloom filter has the advantage that space efficiency and query time efficiency are much better than ordinary algorithms. A hash table can be used to determine whether an element is in a collection or not, and the retrieval is very efficient. Nevertheless, Bloom filter does the same thing with only one fourth or one eighth or even lower of the space complexity of the hash table approach. Bloom filter can insert elements, but non-existing elements can be deleted. Bloom filter is impossible to have a false negative, as long as the elements exist in Bloom filter. However, Bloom filter has one major shortcoming that it can produce false positive, and the more similar elements there are, the greater the false positive rate will be.

2.2 Usage

Since Bloom proposed this theory in reference [2], Bloom filter has been used for various purposes:

A. *Blacklist*

One of the most typical applications is the blacklist. Bloom filter is used to filter usernames or IP addresses, e-mails, etc. If the record is not on the blacklist, it can be passed; otherwise, it is not allowed to pass. If miscarriage of justice occurs, normal users will be judged as blacklisted users. A whitelist can be created to eliminate misjudgments.

B. *Duplicate URL detection for web crawler*

When a web crawler gets a URL, it needs to determine whether the URL has been accessed. The number of URL records acquired by web crawlers is often huge, which makes the storage and processing of these data more difficult. If Bloom filters are adopted in the web crawler, it can greatly reduce the storage capacity and improve the processing efficiency. If there is a misjudgment, the URL that has not been accessed will be wrongly judged to have been accessed [3].

C. *File or record lookup*

The files on disks or records in databases need to be stored in Bloom filter as keys. When accessing disk files or database records, there is no need to access them directly from disks or databases. Bloom filter can be used to detect the existence of these data. If the data exist, access is initiated. It can avoid empty queries of disks or databases. If a misjudgment occurs, files or records that do not exist are misjudged to be existent. The negative effects of the misjudgment are negligible.

D. *CDN (content delivery network)*

CDN adopts proxy caching technology, and its proxy cache server adopts distributed cloud storage. When the CDN server is accessed, firstly, the local server will be looked for whether it has a cache or not. If not, other sibling servers will be checked. Other sibling servers' caches are stored as keywords

in a Bloom filter. To avoid an empty query, the Bloom filter of the local server should be queried before going to another server. If a misjudgment occurs, a cache that does not exist will be misjudged to be existent. This has little negative impact too.

E. *Others*

Bloom filter also has some other common uses, such as "Web Cache Sharing," [4] "Query Filtering and Routing," [5–8] "Compact Representation of a Differential File," [9] "Free Text Searching," [10] "OceanStore"[8, 11]. In summary, Bloom filter is very versatile. Over time, more new applications for Bloom filter will be developed. With the arrival of the big data era, Bloom filter will surely exert more value.

2.3 Details of Bloom filters

2.3.1 Constructing Bloom filters

There's a set of n elements. Bloom filter uses the bit vector V with length m to describe the membership information of A. Therefore, k hash functions, $h_1, h_2, \ldots, h_k$ with $\forall a_i \in A$, $h_i(a_i) \in \{1..m1\}$, are used as described below:

The following program builds a Bloom filter corresponding to set A. The bit vector V of the Bloom filter has m bits, using k hash functions $h_1, h_2, \ldots, h_k$:

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

So, if a_i is a member of set A, all bits corresponding to the hash value of a_i are set to 1 in the resulting Bloom filter V. The following program tests whether an element belongs to set A:

```
Boolean membershipTest(elm, BitVector V, hash functions) {
    for(i=0;i<k;i++)
        if V[hi (elm)] != 1
            return false;
    return ture;
}
```

When new elements are added to a collection, their corresponding locations are evaluated by the hash function, and the bits are set in the filter:

```
BitVector addElement(elm, BitVector V){
    for(int i=0;i<k;i++)
            V[hi (elm)] = 1;
        return V;
    }
```

If two Bloom filters are to be merged, simple bitwise OR manipulation is performed between two-bit vectors:

```
BitVector filterUnion(BitVector V1, BitVector V2){
    return V=V1|V2; // bitwise-or
}
```

2.3.2 Bloom filters—the math

One striking feature of Bloom filters is that there is a clear trade-off between filter size and error rate. It is observed that after n keys are inserted into an m-sized filter using the k hash function, the probability of a bit still being zero is:

$$P_0 = \left(1-\frac{1}{m}\right)^{kn} \approx 1-e^{-\frac{kn}{m}} \tag{1}$$

It is assumed that all hash functions are perfect, and they evenly distribute the elements of set A throughout the space {1..M}. In practice, good results are obtained using MD5 and other hash functions [12].

Therefore, the probability of a false positive (the probability that all k bits have been previously set) is:

$$\begin{aligned} P_{\mathrm{err}} &= (1-p_0)^k = \left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^k \\ &\approx \left(1-e^{-\frac{kn}{m}}\right)^k = e^{k1n\left(1-e^{-kn/m}\right)} \end{aligned} \tag{2}$$

In Eq. (2), P_{err} is minimized for $k = \frac{m}{n} 1\text{n } 2$ hash functions.

The number of hash functions used in practice is not the more the better. The computational overhead of each hash function is constant. Initially increasing the number of hash functions reduces the false positive rate, but the incremental benefits of adding hash functions decrease after a certain threshold value (see Fig. 1).

In Fig. 1, the Bloom filter is 32 bits per item (m/n = 32). At this point, 22 hash functions are used to minimize the false positive rate. However, adding hash functions does not significantly reduce the error rate when more than 10 hash functions have been used.

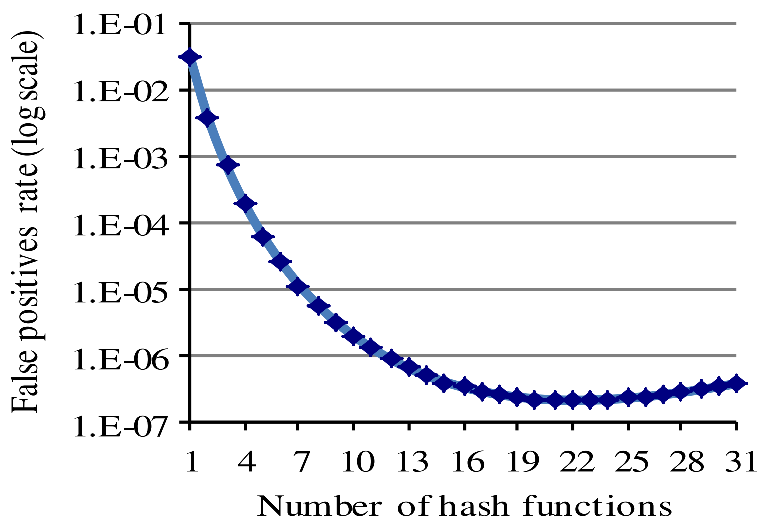

Fig. 1 The false positive rate corresponding to the number of hash functions. The curve shows the relationship between the number of hash functions and the false positive rate. The Bloom filter is 32 bits per item ($m/n = 32$). Initially, the false positive rate decreases as the number of hash functions increases. After reaching the threshold value (22), the number of hash functions increased, while the false positive rate increased slightly

Equation (2) is the basic formula of Bloom filter. Error rate (p_{err}), number of hash functions, and bits per entry (m/n), as long as two of these three parameters are determined, the remaining one is also determined (see Fig. 2).

In Fig. 2, different rows represent different numbers of hash keys. It is worth noting that the error rate of using 32 keys is not significantly an advantage over using only 8 keys.

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

Equation (3) can be derived from Eq. (2). From Eq. (3), the ratio of the size of the bit vector to the number of elements is obtained. Multiply both sides of Eq. (3) by n, then the size m of the bit vector V can be

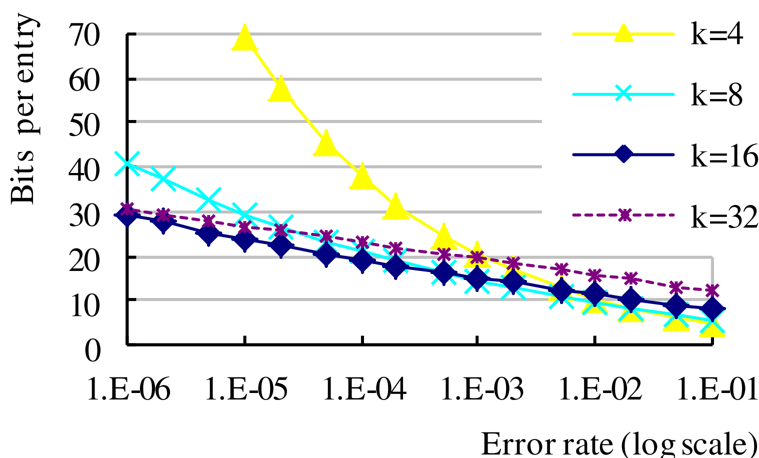

Fig. 2 The error rate desired as a function of the size of the Bloom filter (bits/entry). The number of hash functions is k. The curve represents the relation between error rate and the size of Bloom filter (bits/entry) under the premise of setting the number of hash functions

obtained. In other words, as long as the number of elements, false positive rate, and hash function is determined, the size of the bit vector V can be determined accordingly.

To summarize, Bloom filter is a compact data structure for querying the existence of elements. But it has a false positive rate. To design Bloom filter, we need to weigh the error rate (conflict), the number of hash functions (driver computation overhead), and the size of bit vectors. Formula (2) is the main formula of Bloom filter, which is the analysis basis of Bloom filter.

2.3.3 A modified algorithm

2.3.3.1 Similar elements bring more false positives

Bloom filter produces a hash value by plugging keys into the hash function. In the theoretical analysis of the Bloom filter, it is always assumed that the hash function is perfect with uniform distribution [13]. However, in reality, the perfect hash function does not exist. Whether the hash function is based on addition or multiplication or based on the shift, it is more or less likely to have some clustering of the hashing value [14], which leads to more conflicts, especially when there are more approximations. Now, let us look at a classic hash function in Bloom filter, Bob Jenkins' functions; the algorithm is implemented 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;
}
```

Austin Appleby released a hash function named MurmurHash in 2008. The algorithm is implemented as follows:

```
unsigned long long MurmurHash64B ( const void * key,
  int len, unsigned int seed ){
    const unsigned int m = 0x5bd1e995;
    const int r = 24;
    unsigned int h1 = seed ^ len;
    unsigned int h2 = 0;
    const unsigned int * data = (const unsigned int *)key;
    while(len >= 8){
        unsigned int k1 = *data++;
        k1 *= m; k1 ^= k1 >> r; k1 *= m;
        h1 *= m; h1 ^= k1;
        len -= 4;
        unsigned int k2 = *data++;
        k2 *= m; k2 ^= k2 >> r; k2 *= m;
        h2 *= m; h2 ^= k2;
        len -= 4;
    }
    if(len >= 4){
        unsigned int k1 = *data++;
        k1 *= m; k1 ^= k1 >> r; k1 *= m;
        h1 *= m; h1 ^= k1;
        len -= 4;
    }
    switch(len){
        case 3: h2 ^= ((unsigned char*)data)[2] << 16;
        case 2: h2 ^= ((unsigned char*)data)[1] << 8;
        case 1: h2 ^= ((unsigned char*)data)[0];
        h2 *= m;
    };
    h1 ^= h2 >> 18; h1 *= m;
    h2 ^= h1 >> 22; h2 *= m;
    h1 ^= h2 >> 17; h1 *= m;
    h2 ^= h1 >> 19; h2 *= m;
    unsigned long long h = h1;
    h = (h << 32) | h2;
    return h;
}
```

These are all classic hash functions, and they are all based on shifted hash functions. In the process of shift, the low and high bits are discarded. This must result in a loss of precision. For similar elements, the difference between them is small [15]. The difference bits between similar elements may be discarded by the shift operation. This leads to differences in similar elements being ignored [16]. This is also a source of false positives in the actual use of the Bloom filter. The more numbers of similar elements input, the more likely it is to produce false positives. This is determined by the function. Therefore, Bloom filter with differential amplification and rehash is proposed [17].

2.3.3.2 Constructing new Bloom filters There is a set $A = \{a_1, a_2, \ldots, a_n\}$ of n elements. If a_i is an element of A, mapping a_i to a part of the filter by some hash functions, each element amplifies the difference by function g (), which makes g $(a_i) = a^*_i$. Bloom filters describe membership information of A using a bit vector V of length m. For this, k hash functions, h_1, h_2, …, h_k with $\forall a_i \in A$, $h_i(a_i) \in \{1..m1\}$, and d hash functions, f_1, f_2, …, f_d with $f_j(a^*) \in \{m1..m\}$, are used as described below:

The schematic diagram of the algorithm is shown in Fig. 3.

The following procedure builds an m bits Bloom filter, corresponding to a set A and using h_1, h_2, …, h_k and f_1, f_2, …, f_d hash functions:

```
BitVector buildBloomFilterNew(set A, hash functions,
integer m, g()){
        BitVector V[1..m]=0;
        for(i=0;i<n;i++)
              for(j=0;j<k;j++)
                    V[hj(ai)]=1; //0<=hj(ai)<=m1
        for(i=0;i<n;i++)
              for(j=0;j<d;j++){
                    a*i =g (ai) ;
                    V[fj(a*i)]=1; //m1<=fj(a*i)<=m
              }
        return V;
}
```

Therefore, if a^i is member of a set A, in the resulting Bloom filter V, all bits obtained corresponding to the hashed values of a^i are set to 1. The program to test whether the element elm belongs to Bloom filter is as follows:

```
Boolean membershipTestNew(elm, BitVector V, hash
functions, g()) {
      for(i=0;i<k;i++)
          if V[hi (elm)] != 1
               return false;
      for(i=0;i<d;i++){
          a*i =g (elm) ;
          if V[fi (a*i)] != 1
               return false;
      }
      return ture;
}
```

The algorithm for merging two Bloom filters has not changed. The procedure to add a new element to Bloom filter is as follows:

```
BitVector addElementNew(elm, BitVector V,g()){
      for(int i=0;i<k;i++)
            V[hi (elm)] = 1;//0<=hj(ai)<=m1
      for(int i=0;i<k;i++){
            a*i =g (elm) ;
            V[fi (a*i )] = 1;//m1<=fj(a*i)<=m
      }
            return V;
}
```

2.3.3.3 New Bloom filter—the math In the new Bloom filter, a^*_i is mapped to the latter part (m1..m) of the bit vector by d hash functions. According to the characteristics of Bloom filter, it will inevitably generate false positive rate, which is set as p', then:

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

Assuming that the false positive rate of the new Bloom filter is $p_{\text{New-err}}$, then:

$$p_{\text{New-err}} = \cdot P_{\text{err}} \times p \tag{5}$$

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

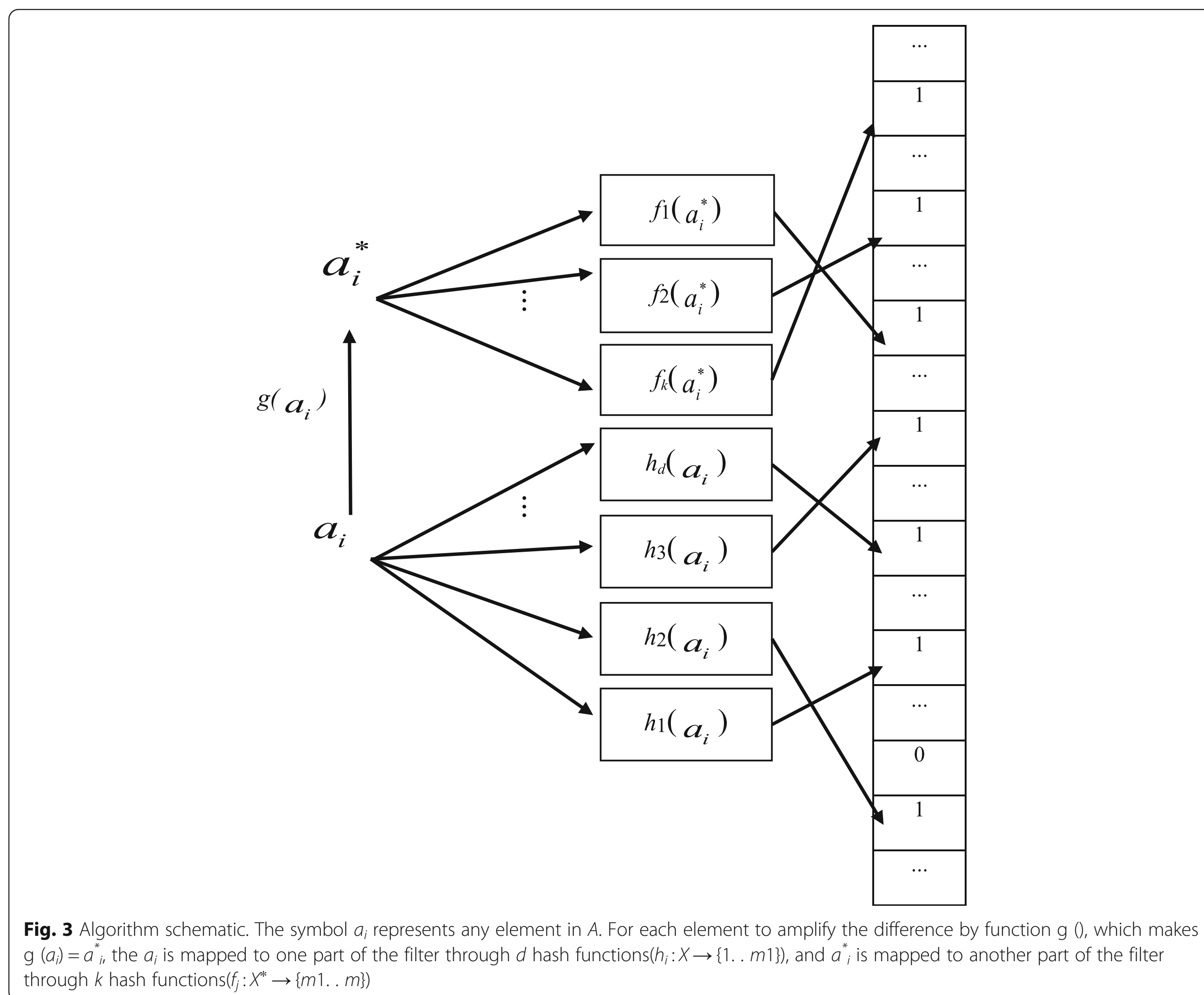

Fig. 3 Algorithm schematic. The symbol a_i represents any element in *A*. For each element to amplify the difference by function g (), which makes g $(a_i) = a^*_i$, the a_i is mapped to one part of the filter through *d* hash functions($h_i : X \rightarrow \{1..m1\}$), and a^*_i is mapped to another part of the filter through *k* hash functions($f_j : X^* \rightarrow \{m1..m\}$)

3 Used in bar code identification and processing

3.1 Bar code processing scheme

Nowadays, bar codes are more and more widely used. In many cases, the speed of bar code processing is increasingly required, such as the sorting system of large logistics companies [18]. This section describes a bar code processing scheme based on the improved Bloom filter introduced in this article. This scheme can not only improve the processing speed of bar codes, but also reduce the false positive rate as much as possible.

A bar code recognition scheme based on improved Bloom filter is presented below(shown in Fig. 4).

In this scheme, we (1) assume the image is in the ideal state of identification and verification, (2) cut out the bar code area from a picture, and (3) properly handle the detected bar codes so that it is as ideal as possible. The ideal state is that the bar code generated by code generator has no defilement of the image, both sides of the bar code are horizontal and vertical, and the bar code is positive from left to right.

In this scenario, the bar code area needs to be cropped from the image that contains the bar code to identify the bar code from the area. The obtained bar code will be the input element of the Bloom filter. Of course, the Bloom filter adopted here is the improved Bloom filter introduced in this paper. This solution takes advantage of the high-speed feature of the Bloom filter to automatically recognize the bar code and quickly retrieve it for further processing.

The bar code area needs to be cropped out of the original image first. The cutting steps of the bar code area in an image are as follows:(1) morphology gradient operation, ignore the Y direction gradient, and focus on the X direction gradient. The grayscale image of the image was calculated separately, and the x-direction gradient was subtracted from the y-direction gradient to retain

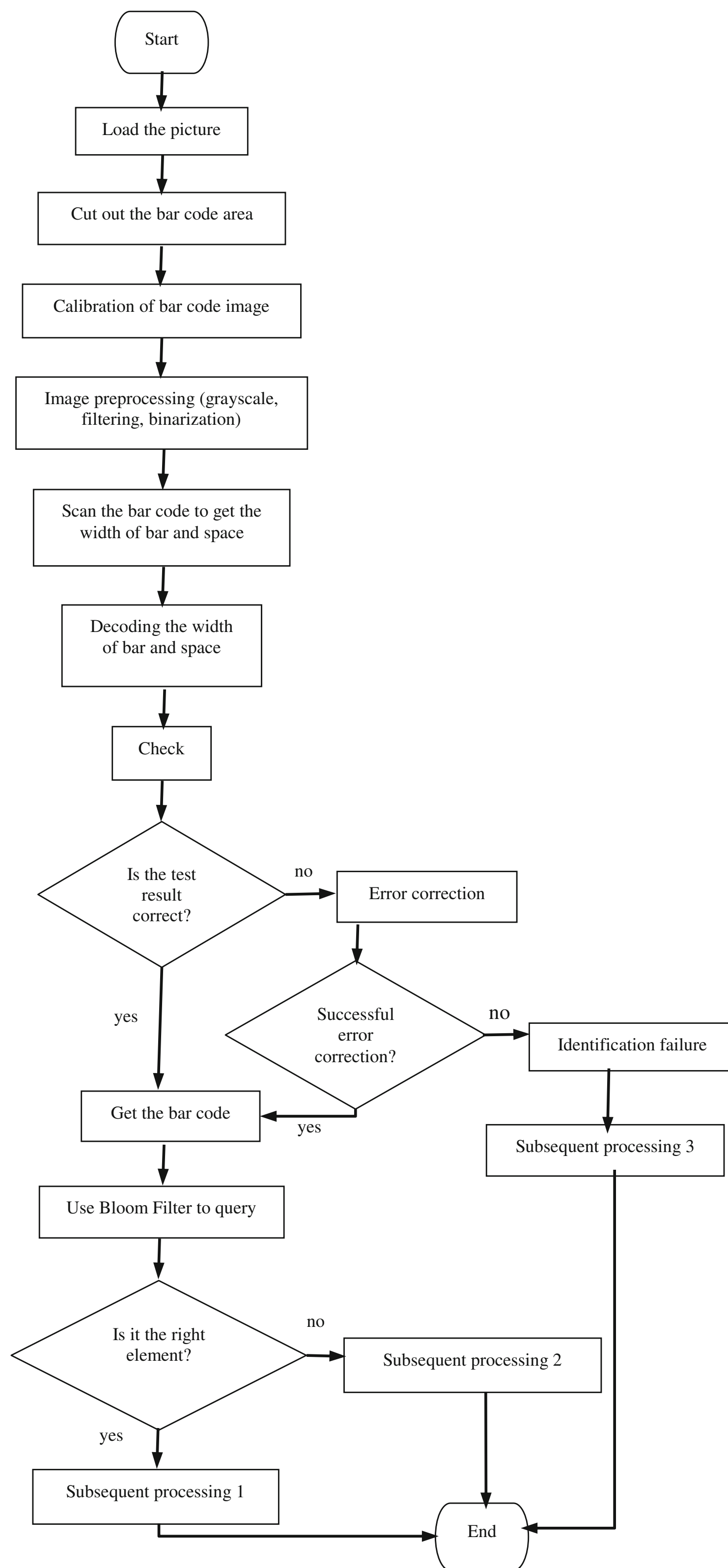

Fig. 4 Bar code recognition processing flow chart. Boxes represent processing steps. The diamond frame represents the selection. The arrow indicates the direction of execution of the process

Fig. 5 The camera captured the original image. The original image of the bar code is captured by the camera

the x-direction feature and remove the y-direction interference. (2) Image blur processing and binarization; at this point, the corresponding fuzzy parameters and threshold parameters need to be adjusted to get the relatively best results. (3) Closed operation; to eliminate the black gap, dilation and erosion can be used at this point to eliminate certain areas connected to the rectangular area of the bar code. (4) Find the contour, calculate the maximum area of the contour, and fit the contour rectangle to get the final result. After cutting successfully, the image containing the bar code area can be obtained (see Figs. 5, 6, 7, 8, and 9).

After the bar code area is cropped, the bar code image needs to be corrected, preprocessed, and scanned to obtain the width of the bar and space. The bar code information is then decoded according to the width of the bar and space. The obtained bar code information can be entered into the improved Bloom filter for retrieval, and then further processing can be conducted according to the retrieval results.

A bar code is made up of several numbers, letters, or symbols, and the same company's bar codes are often only slightly different [19]. These bar codes, as input elements, are equivalent to a large number of similar keys for Bloom filter. When the hash functions of Bloom filter have some similar input elements, they may ignore the small differences, resulting in the hash function producing the same output on the similar elements and generating false positives [20]. Because a company's bar codes are easy to focus on one section, this will result in more false positives. However, each bar code is unique, and the difference between the two bar codes is certain. Therefore, the characteristic part of the bar code can be extracted for mathematical transformation to indicate the difference between similar elements magnified. Thus, the difference between the keys in the hash function of Bloom filter can be enlarged and the false positive rate can be reduced [21].

The following two tables are illustrating the application of the scheme(see Table 1 and Table 2). The "*a*"

Fig. 6 Morphology gradient operation. Gradient images are calculated respectively. The X-direction gradient is used to subtract the Y-direction gradient, so that the X-direction characteristics can be retained and the interference in the Y-direction can be removed

Fig. 7 Image blur processing and binarization. The image is processed in a fuzzy way and two-valued. Processing should pay attention to adjusting the corresponding fuzzification parameters and threshold parameters to get the best results

is the original input element of the Bloom filter, and the "value of a" is the value of the bar code. The "a^*" is the result of the mathematical transformation of the element "a".

3.2 Test results

A lot of testing has been carried out. According to the statistical effect of the test, the bar code identification error rate is lower when using the improved Bloom filter in the scheme, while the error rate is higher when using the classic Bloom filter in the scheme. If there are more similar bar codes in the test sample, the false positive rate will be greatly reduced. At the same time, the test results show that this scheme is faster than the traditional scheme using databases.

4 Conclusion

The Bloom filter produces hash values by entering keys into hash functions. In the theoretical analysis of Bloom filter, it is usually assumed that the hash functions are perfect with uniform distribution. In reality, however, the perfect hash function never exists. Sometimes, due to the particularity of the data, some hash functions have different uniform distribution of hash values due to input sample. This makes it difficult to evaluate the distribution characteristics of the hash function. So, the evaluation of the uniform distribution characteristics of the hash function is generally carried out by statistical methods. Regardless of the hash function is based on the addition, multiplication, or based on the displacement, there will be more or less hash value clustering, which leads to conflicts, especially when the input keys have more approximation. The classical Bloom filter and most of its variants are all directly hashing the elements, which cannot avoid the higher false positive rate when there are more approximations.

Therefore, an improved Bloom filter is proposed in this paper. The bit vector of Bloom filter is divided into two parts. For each element to amplify the difference by g(), which makes $g(a_i) = a^*_i$, the a^*_i is mapped to another part of the bit vector by some hash functions. When an element is retrieved in both parts of the bit vector, it is shown that the element exists in the Bloom filter.

Fig. 8 Closed operation. The closed operation is used to remove the black gap. The closed operator is set according to the condition of the gap

Fig. 9 Crop to the bar code area. The bar code area is cropped from the original image

Because this algorithm transforms elements through the function g(), the differences between elements are magnified. This reduces the collision rate of the Bloom filter. Especially when the values of input elements are similar, the effect of reducing collision is obvious. This algorithm reduces the collision rate of the Bloom filter, but it does not add much space and time cost. Algorithm implementation is also easy. The results of the statistics are also better. The algorithm is also applied to bar code recognition and processing, and a specific scheme is designed. Because the bar codes of each company are concentrated, the input elements of Bloom filter are similar. According to the results of testing, this algorithm can effectively reduce the conflict rate of bar codes in Bloom filter. At the same time, the bar code automatic identification scheme designed in this paper has an advantage of high processing speed.

5 Methods

The purpose of this study is to find a method to reduce the false positive rate of Bloom filter, especially when the input contains a large number of approximate elements. The same company's bar codes are extractly the data with a small difference. This paper uses this improved Bloom filter to process the bar codes and designs a complete scheme.

An improved Bloom filter is proposed in this paper. We divide the bit vector of Bloom filter into two parts. Consider a set $A = \{a_1, a_2, \dots, a_n\}$ of n elements. If a_i is an element of A, mapping a_i to a part of the filter by some hash functions, each element to amplify the difference of g(), which makes $g(a_i) = a^*_i$. Bloom filters describe membership information of A using a bit vector V of length m. Every element a_i could be mapped to a part of the bit vector by some hash functions. The a^*_i is mapped to another part of the bit vector by some hash functions too. When querying an element, it means that the element belongs to the set as long as it has a corresponding value for both parts of its bit vector.

In many cases, due to the requirement of processing speed, Bloom filter has to be used to deal with bar codes. However, the bar codes of the same company often differ slightly, which easily results in the increasing of the false positive rate of the Bloom filter. A complete bar code processing scheme is designed in this paper. This scheme adopts the improved Bloom filter proposed in this paper to process bar codes effectively. The improved Bloom filter proposed in this paper is used in the scheme. The scheme needs to identify the bar code and get the corresponding data. The data is then inputted as an element into the Bloom filter for looking up. If found, the bar code is the element that exists. For extremely individual elements, if something goes wrong, they could also be queried from the original database.

Table 1 Bar code mapping and differential amplification transformation

a	Value of a	Hash value($h_i : X \to \{1..m1\}$)
a_1	x1x2x3x4x5x6x7x8x9	$h_1(a_1), h_2(a_1), \dots, h_j(a_1), \dots, h_k(a_1)$
a_2	y1y2y3y4y5y6y7y8y9	$h_1(a_2), h_2(a_2), \dots, h_j(a_2), \dots, h_k(a_2)$
a_3	z1z2z3z4z5z6z7z8z9	$h_1(a_3), h_2(a_3), \dots, h_j(a_3), \dots, h_k(a_3)$
a_4	u1u2u3u4u5u6u7u8u9	$h_1(a_4), h_2(a_4), \dots, h_j(a_4), \dots, h_k(a_4)$
a_5	v1v2v3v4v5v6v7v8v9	$h_1(a_5), h_2(a_5), \dots, h_j(a_5), \dots, h_k(a_5)$
…	…	…

Element value is the bar code. A certain rule can be followed to amplify the differences between elements.By the *k* hash functions {*h1, h2, ……, hk*} to obtain the *k* bits are stored in {*1 .. m1*} the corresponding position

Table 2 Mapping after transformation

$a^*_i = g(a_i)$	Value of a^*	Hash value($f_j : X^* \to \{m1..m\}$)
a^*_1	X1X2X3X4X5X6X7X8X9	$f_1(a^*_1), f_2(a^*_1), \dots, f_j(a^*_1), \dots, f_d(a^*_1)$
a^*_2	Y1Y2Y3Y4Y5Y6Y7Y8Y9	$f_1(a^*_2), f_2(a^*_2), \dots, f_j(a^*_2), \dots, f_d(a^*_2)$
a^*_3	Z1Z2Z3Z4Z5Z6Z7Z8Z9	$f_1(a^*_3), f_2(a^*_3), \dots, f_j(a^*_3), \dots, f_d(a^*_3)$
a^*_4	U1U2U3U4U5U6U7U8U9	$f_1(a^*_4), f_2(a^*_4), \dots, f_j(a^*_4), \dots, f_d(a^*_4)$
a^*_5	V1V2V3V4V5V6V7V8V9	$f_1(a^*_5), f_2(a^*_5), \dots, f_j(a^*_5), \dots, f_d(a^*_5)$
...	...	...

The value of *a** is obtained by *g(a)*. Use the *d* hash functions {*f1, f2, ……, fd*} to obtain *d* bits stored inc {*m1 .. m*} in the appropriate location

Acknowledgements
This article was also assisted by Mr. Jun Ye and others, and thanks together. In particular, we would like to thank the reviewers for their valuable comments.

Funding
This work was partially supported by the Sichuan University of Science and Engineering research fund: 2014KY01, Enterprise Informatization and Internet of Things measurement and Control Technology Key Laboratory of the University of Sichuan: 2014WYY04, Support Plan of Sichuan Science and Technology Department: 15ZC0195, and Sichuan Education Department: 17SB0345.

Availability of data and materials
Data sharing is not applicable to this article as no datasets were generated or analyzed during the current study.

Authors' contributions
MJ is responsible for the algorithm and scheme design and most of the other work. CSZ participated in the design code. ZFM participated in the scheme design. JW is involved in the implementation of the scheme. All authors read and approved the final manuscript

Competing interests
The authors declare that they have no competing interests.

Publisher's Note
Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 26 July 2018 Accepted: 8 November 2018
Published online: 06 December 2018

References

1. M. Jiang, C. Zhao, G. Xiang, in *6rd International Congress on Image and Signal Processing (CISP 2013)*. A modified algorithm based on the bloom filter (2013), pp. 1087–1091
2. B. Bloom, Space, "time trade-offs in hash coding with allowable errors,". Commun. ACM **13**(7), 422–426 (1970)
3. W.H.A. Yuen, A hybrid Bloom filter location update algorithm for wireless cellular system. IEEE Intl. Conf. on Communications **vol. ICC**(3), 1281–1286 (1997)
4. L. Fan, P. Cao, J. Almeida, A. Broder, *Summary cache: a scalable wide-area web cache sharing protocol* (in Proceedings of ACM SIGCOMM'98, Vancouver, 1998)
5. S.D. Gribble, E.A. Brewer, J.M. Hellerstein, D. Culler, in *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*. Scalable, "Distributed Data Structures for Internet Service Construction" (OSDI 2000, San Diego, 2000)
6. S.D. Gribble, M. Welsh, R.V. Behren, E.A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A.D. Joseph, R.H. Katz, Z. Mao, S. Ross, B. Zhao, The Ninja architecture for robust internet-scale systems and services. Comput. Netw. **35**(4), 473–497 (2001)
7. T.D. Hodes, S.E. Czerwinski, B.Y. Zhao, A.D. Joseph, R.H. Katz, An architecture for secure wide-area service discovery. Wirel. Netw **8**(2–3), 213–230 (2002)
8. J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao, in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. OceanStore: "An Architecture for Global -Scale Persistent Storage" (ASPLOS 2000, Cambridge, 2000)
9. M. Mitzenmacher, in *Twentieth ACM Symposium on Principles of Distributed Computing*. Compressed bloom filters (PODC 2001, Newport, Rhode Island, 2001)
10. J.K. Mullin, A second look at Bloom filters. Commun. ACM **26**(8), 570–571 (1983)
11. S. Rhea, W. Weimer, "Data location in the oceanstore", in unpublished, UC Berkeley. Journal of Nutrition Education & Behavior **44**(4), S25 (1999)
12. M.V. Ramakrishna, Practical performance of Bloom filters and parallel free-text searching. Commun. ACM **32**(10), 1237–1239 (1989)
13. K.W. Choi, D.T. Wiriaatmadja, E. Hossain, Discovering mobile applications in cellular device-to-device communications: Hash function and bloom filter-based approach. IEEE Trans. Mob. Comput. **15**(2), 336–349 (2016)
14. J. Qian, Q. Zhu, H. Chen, Integer-granularity locality-sensitive bloom filter. IEEE Trans. Comput. **20**(11), 2125–2128 (2016)
15. H. Byun, J. Lee, H. Lim. "Ternary Bloom filter replacing counting Bloom filter", IEEE International Conference on Consumer Electronics-asia, 2016, ICCE-Asia: 1-4. https://doi.org/10.1109/ICCE-Asia.2016.7804774
16. J. HM, H. Lim, New approach for efficient IP Address lookup using a Bloom filter in trie-based algorithms. IEEE omputer Society **65**(5), 1558–1565 (2016)
17. D. Pellow, D. Filippova, C. Kingsford, Improving Bloom filter performance on sequence data using k-mer Bloom filters. J. Comput. Biol. **24**(6), 547–557 (2016)
18. P. Jiang, Y. Ji, X. Wang, J. Zhu, Y. Cheng, Design of a multiple Bloom filter for distributed navigation routing. IEEE Transactions on Systems Man & Cybernetics **44**(2), 254–260 (2017)
19. T. Pavlidis, J. Swartz, Y.P. Wang, Fundamentals of Bar code information theory. Computer **23**(4), 74–86 (2002)
20. G. Moualla, P.A. Frangoudis, Y. Hadjadj-Aoul, S. Ait-Chellouche, A bloom-filter-based socially aware scheme for content replication in mobile ad hoc networks. Consumer Communications & Networking Conference, 359–365 (2016) https://doi.org/10.1109/CCNC.2016.7444807
21. H. Ko, G. Lee, S. Pack, K. Kweon, Timer-based bloom filter aggregation for reducing signaling overhead in distributed mobility management. IEEE Trans. Mob. Comput. **15**(2), 516–529 (2016)

Submit your manuscript to a SpringerOpen journal and benefit from:

► Convenient online submission
► Rigorous peer review
► Open access: articles freely available online
► High visibility within the field
► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com