A Multibit Representation of Bloom Filter for Simultaneous Acquisition of Membership and Attribute Information

Ying-Chih Tseng[1,2] and Heng Ma[3]
(Corresponding author: Heng Ma)
Department of Obstetrics and Gynecology, Hsinchu Cathay General Hospital, Hsinchu, Taiwan[1]
Department of Information Management, Yuanpei University of Medical Technology, Hsinchu, Taiwan[2]
Department of Industrial Management, Chung Hua University, Hsinchu, Taiwan[3]
707, Sec.2, WuFu Rd., Hsinchu, 30012 Taiwan
(Email: hengma@chu.edu.tw)
(Received Sept. 10, 2018; Revised and Accepted Apr. 13, 2019; First Online Aug. 10, 2019)

Abstract

As dataflow on Internet is growing exponentially, processes that can efficiently extract meaningful information have become a crucial factor for many successful applications. For example, the purpose of membership determination is to discriminate whether a fragment of the dataflow is an element of a specific dataset. The Bloom filter has been well recognized for dealing with such a problem, but it can only provide the membership information. Recently, membership determination that can accompany with additional attribute information has becoming increasingly important, because it could save considerable time for the secondary querying once the membership is confirmed. Therefore, this study proposes a multibit representation of the original Bloom filter by encoding the attribute codes, instead of binary counterpart, for resolving such a situation. Simulation results show that the querying efficiency and false-positive ratios are fairly competitive with reasonable memory-space usages.

Keywords: Attribute Information; Bloom Filter; Membership Determination

1 Introduction

Emerging business models on the Internet, such as blogs, social groups, instant messaging and on-line shopping, have brought the world to a completely different look in recent years. These models usually require unique processes for efficiently manipulating information in real time, such as keyword searching, frequency measuring and account-password matching for accommodating the growing Internet speed. Membership determination could be accounted for a branch of such processes, which could be characterized as: Given a large dataset composed of elements of a certain feature, *e.g.*, login accounts or email addresses, the purpose is to discriminate whether a random query is an element of the dataset. The size of the dataset could grow even larger as time proceeds, and the expansion, however, could influence the querying time to some degrees, so a means that can perform such a process without compromising the querying speed is very critical.

The Bloom filter [2] has been widely employed as a core engine for dealing with the membership determination problem, which includes two phases of processes. In the programming phase, a set of hash functions are utilized for mapping each set element to a one-dimensional bit array. Because of the employment of the hashing functions, in the second phase the querying time could achieve nearly constant regardless the size of the dataset. The bit array is considerably large, in which each bit is initially set to zero and turned to one once hit by the hash process. Such a programming behavior guarantees that all the set elements could get all ones in the querying phase to be considered as a member. Such a process, however, could introduce errors called "false positives", where all the cells hit by a non-member element are all ones. Consequently, the element is falsely determined as a member, which is also referred to as false-positive (FP) errors.

The FP errors could be critical for some applications that requires stringent membership discrimination, which must be within an acceptable level. In [9], the authors show that it is important how such a matter could influence cloud computing, in which they point out that attribute-based encryption is a promising cryptographic solution in cloud environment because access control is important as far as data security is concerned. For example, Thiyagarajan and Ganesan [14] proposes an architecture of multiple keyword search by building index using Bloom filter, taking advantage of its consistent pro-

cessing time for querying through hash functions. The engagements of hash functions, due to their uniqueness have also given themselves a crucial role for defending malicious intrusions, also referred to as intrusion detection systems (IDS), as in [1,12]. For our proposed method, we employ a special yet simple hash function, designed for elements of texts and symbols with various lengths. Such a design was intended to accommodate packet headers on the network, making recognition of known malicious packets with certain characteristics possible.

The two-phase mechanism of Bloom filter is very similar to the process of artificial neural network (ANN), where a certain large amount of known data are iteratively presented to a specific paradigm, and the results are utilized as an on-line component for determining the outcome of inquiring data. In our previous work, an ANN paradigm (CMAC) was employed for resolving the issue of membership determination with multiple attributes [11]. Being a supervised type of ANN, a number of target values of CMAC must be specified for the paradigm to operate properly. In our implementation, each target value designated a specific attribute code. When the training process is complete, all the set data belonging to an identical attribute code would approach that target value, where a recognition zone could be formed for a querying element to be considered as member if they pass through the zones. Furthermore, the associated attribute code could be immediately identified. The scheme works well in simultaneously obtaining membership and the associated attribute information, while the FP errors could also be kept at an acceptable level; however, ANN requires extensively computational time for converging the recognition zones to a sufficiently small width for low FP errors, which is not suitable for dynamic membership insertion. Furthermore, ANN uses floating-point numbers to represent the content of cells in the array, which could consume considerably large memory in real time. In this paper, we change the contents of the array to integer numbers as the attribute codes. Although multiple bits are still required for each cell, the memory space is far less than the CMAC-based approach. Our objective is to demonstrate a feasible approach for applications that require membership determination with simultaneous attribute information. We focus on the computational efficiency instead of hardware implementations because hardware technology is progressing from time to time. Nevertheless, we still take the memory overhead as a crucial factor in the proposed approach. In the following, we present recent efforts in the literature for the addressed problem.

The difficulty of the addressed problem lies in that additional structures or computational components could be involved in the determination process for providing the attribution information at the same time. Intuitively, using a set of parallel Bloom filters could solve the problem, where each Bloom filter could represent an attribute code. The main drawback of such an approach has been that the size of each Bloom filter is difficult to decide because the number of elements for each attribute code could vary to some extent. Using the same size of the parallel Bloom filters could cause dramatic memory waste. Furthermore, a querying element must go through all the filters, which is not only a time-consuming process but could also lead to extra errors when multiple filters respond true membership. In [15], variants of the original Bloom filter were proposed for dealing with multi-attribute membership problem. In this approach, PBF (Parallel-Bloom filter) is responsible for defining attributes for a number of counting parallel Bloom filters when an element could be associated with multiple attributes. PBF-HT (PBF with a Hash Table) and PBF-BF (PBF with a Bloom Filter) were designed for verification purposes, whose objective was to compensate the extra false positive errors that could be introduced by PBF. Consequently, the number of attribute codes could become very large in particle applications. Although PBF-HT and PBF-BF could compensate time and space losses, the memory usage and additional false-positive errors could lead PBF to an unfit situation. In 2012, the concept of approximate membership query (AMQ) [8] was raised based on its previous work [15]. The AMQ is an approach referred to as that the degree of an element is within a certain range of a member's boundary. This approach could be meaningful for some prediction models especially in networking, however, the addressed problem could not be resolved by this approach because each element is associated with only one attribute, and the membership allows no ambiguous regions.

Due to the success of the original Bloom filter for a variety of applications, a number of approaches tackled the addressed problem with modified architectures. For example, the invertible Bloom lookup table proposed in [6] is to provide key-value pairs upon querying. The data structure was designed to accommodate both keys and values of integers, where inserting an element would always be successful because distinct keys are used. In the querying mode, an inquiring element x would receive a value y that composes a pair with x, and then information could be obtained given that y is not null, which is an extra step with conditions in obtaining the attribute information. As it becomes increasingly important to acquire the membership with its additional attribute information at the same time, especially in the networking applications, the following work had been done in an effort to resolve this problem under a packet routing situation. The routing mechanisms have been a key factor for keeping the Internet not only fluent but healthy, which requires accurate and timely dispatching of millions of arriving packets in a fraction of second. In [7], the authors proposed combinatorial Bloom filters, in which a considerably large group of hashing functions were employed. They defined a unique binary vector code for each hashing group for differentiating the attributes; therefore, an inquiring element must go through all the hashing groups before being sent to the general Bloom filter for the membership determination. In this approach, different hashing group combinations represent a certain attribute, but the size

of the binary vector for hashing could be very long to accommodate a large number of attributes. Qiao *et al.* [13] use two data structures for determining membership and the ID information. The first one, index filter, is a general Bloom filter, whose purpose is solely for membership determining, while the second one, the set-id table, stores the ID information for each member using multiple hash functions. The process proceeds when one of the multiple hash functions hit a zero in the set-id table, the ID information is then inserted into that cell and the identical hash function is used for coding the index filter. In the querying phase, the membership with the ID information could be obtained if one hash function reports one on the index filter and an ID on the set-id table. In this method, the lengths of both structures must be sufficiently large to accommodate a good portion of zeros for better performances.

More recently, using multiple bits instead of a single bit for each cell of the Bloom filter has been proposed to incorporate the set IDs. Xu *et al.* [16] suggested encoding both information in the same data structure could provide more efficient query processing speed. Therefore, they proposed multi-bit array, where insertion and lookup procedures could be achieved by bitwise operations including union and intersection. The approach, however, could result in additional false-positive errors when the set number is fairly large. Although several remedy techniques were proposed, they still compromise the lookup speed by dividing the original structure into several levels. Dai *et al.* [4] also proposed multi-bit structure for encoding the set IDs, where bitwise operations were employed for the insertion and lookup purposes. This approach, unlike [16], was dealing with determining membership of multiple disjoint sets; however, it also suffered from errors at the lookup phase because any query was given a set ID response, which could result in additional false-positive errors. The ID Bloom filter [10] shares the same bitwise operations for element insertion and lookup procedures as in [4], so it could also inherit the chances of misjudgment in the lookup phase. The approach designated the set IDs with decimal numbers instead of bit streams, whose purpose was to save the number of memory accesses in the querying phase; however, using the bitwise operations seems to lack scalability because it is difficult to encode a large number of sets into the array structure.

In this paper, we adopt the concept that both membership and attribute information are coded in the same array to avoid additional structures that consume memory usage, remain the querying speed as in the original Bloom filter, and keep the false-positive ratio within an acceptable level. The multibit representation allows each cell in the array to designate a number of attribute codes. Programming for both membership and attribute information in the same array is a better approach because it does not require a great number of hash functions. Secondly, fast querying could be achieved because the number of memory accesses is mainly dependent on the hash number. Furthermore, the false-positive rate is anticipated to be acceptable because only a single array is addressed by the hashing process. The multibit representation suggests that the attributes would be coded as integer numbers, so the input data in the programming phase include member elements and their associated attribute codes.

2 Methodology

In the proposed approach, since each cell of the array is represented by multiple bits for designating various attribute codes, we must first determine the number of bits for each cell to best conserve the memory space. Of the integer numbers that the multiple bits could represent, we reserve 0 and the largest one for the use of the programming as well as querying phases. For example, if 4 bits are used in each cell, there are 16 integer numbers ranged from 0 to 15, in which 0 and 15 would be reserved, and the remaining 1 14 could be used as attribute codes. Consequently, the relationship between the number of bits in a cell and the number of attribute codes could be characterized as in Equation (1).

$$b = \lceil \log_2(a+2) \rceil \tag{1}$$

where b is the number of bits in each cell, a is the total amount of attribute codes, $\lceil \cdot \rceil$ is the ceiling operator that finds the smallest integer greater or equal to the content.

2.1 The Proposed Approach

The original Bloom filter turns a cell's value from 0 to 1 for those hash hit by all the member elements, which is an important process for expediting membership determination in the querying mode. In the proposed approach, however, since the array is composed of attribute codes instead of binary ones, a specific procedure is employed for programming to achieve simultaneous retrievals of membership and attribute information in the querying phase. However, it is not an easy task because if an element's attribute code is assigned to all the hash hit cells, "collision" could happen, where multiple elements with different attribute codes hit the same cell. Therefore, we introduce two techniques for resolving such a situation: (1) the universal code U using the reserved largest number to represents all the attribute codes, and (2) the threshold rate T to pass for being considered as a member with the associated attribute code. In a sense, (1) is to ensure that the programming process could successfully proceed, while (2) is to save as much memory space as possible. We use examples in the following to further explain how these two techniques work.

As U designates all the attribute codes, the T is the hurdle to pass for being considered as a member, the hash number is h and the total attribute number is a, we describe the membership discrimination rules in the querying phase. The rules are designed to make the querying phase as fast as possible, including:

1) If zero is encountered, it is not a member;

2) If there are all U's, it is not a member;

3) Find the largest count c of the hashed cells of the same attribute code a plus the number of U, and if $c/h \geq T$, the element is recognized as a member of the attribute code a; otherwise, it is not a member.

We further deliberate the procedure with the following four examples with $h = 5$, $a = 8$ and $T = 0.5$, where the sequence is the attribute codes of the hash hit cells by an element is:

1) 2, 0, 3, 5, 6. The sequence includes zero, so the element is immediately rejected for being a member;

2) U, U, U, U, U. The sequence contains all U's, so it is not a member;

3) 4, 2, U, 4, 8. The largest count $c = 3$ with $a = 4$, and element is considered as a member with the attribute code of 4 because $3/5 \geq T$;

4) 3, 5, 6, 8, 6. $c = 2$ with $a = 6$, but $2/5 < T$, so the element is rejected;

5) 3, 4, 4, 3, U. There is a tie between $a = 3$ and 4, and both surpass T, so it's ambiguous and no conclusion is drawn.

With the discrimination rules, we now describe the programming process in the proposed approach.

It's important to describe the hash functions employed in the proposed approach before we go to the programming process. We call it "sequential hashing", which was inspired by [5] that suggests a simple logarithm function disregarding the integer and the first few digits of the decimal parts could form an effective hash function. In our implementation, we employ such a concept with some specific factors including the sum and the accumulated sum the element, the array size m and a position indicator for the array. When the position indicator exceeds m during the hashing process, it is set to the remainder position divided by m. The proposed hashing function is easy to implement and with operational efficiency, and could produce any specified h hash numbers.

Because collisions could happen at any cell through the hashing process, we investigate the magnitude of the collision for each cell, where a pilot run is executed for all the member elements with their attribute codes. The information is recorded in a "hit table" as in Table 1, which is particularly useful for allocating appropriate attribute codes for the array. The first column of the table is the sequential cell number up to the array size m, the second one is the element(s) that hit the cell with the attribute code in the parentheses, and the last column shows the hit frequency of the cell by elements with different attribute codes. Although the table claims memory space, it is only required in the programming phase.

The hit table could somehow depict our programming procedure whose purpose is to determine the arrangement

Table 1: The hit table in the programming phase

C_1	$e_1(5)$	1
C_2		0
C_3	$e_2(8)$	1
C_4	$e_1(5)$	1
C_5	$e_1(5)$	1
C_6	$e_1(5), e_2(8)$	2
C_7	$e_2(8), e_3(2)$	2
C_8	$e_2(8)$	1
C_9	$e_1(5), e_3(2)$	2
C_{10}	$e_2(8), e_3(2)$	2
C_{11}	$e_3(2)$	1
C_{12}	$e_3(2)$	1
⋮	⋮	⋮

of the attribute codes in the array, allowing all the elements to pass the threshold rate T with the smallest number of U. Therefore, we start with cells with the lowest hit frequency, and move the way up by excluding those elements that are already qualified as a member. We demonstrate this concept with a simplified example in Figure 1.

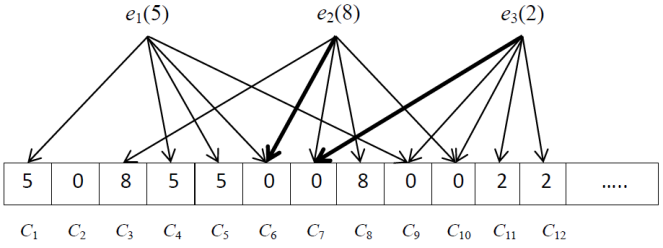

Figure 1: A simplified example for the proposed programming procedure

In Figure 1, like the original Bloom filter, all the cells in the array are initially set to zeros, designating the cell is unused. In the example, the hash number h is 5, meaning each element hits 5 cells after the hashing process. Supposing the threshold rate T is 0.5, at least 3 cells with the same attribute code or the universal code must be hit (3/5=0.6) for the element to be admitted as a member with that attribute code. Consulting the hit table in Table 1, the second column shows the elements that hit this cell with different attribute codes. Therefore, the first step of the programming procedure is to fill the attribute code to those cells with a single hit. For example, C1 is only hit by e1, whose attribute code 5 is then settled in C1, so are the other cells with only one hit. When a cell is hit by multiple elements, *e.g.* C6, since e1 is already qualified as a member but e2 is not, the attribute code 8 of e2 would be issued for C6. The thick arrow in Figure 1 represents the winner of the cell. After C6 is settled with the attribute code 8, the next cell with multiple hits is C7,

which would be set to the attribute code of e3 because e2 has become a member. For both C9 and C10 cells, since both hit elements are members, a randomly selected attribute code of the hit elements would be assigned to the cell. When no more attribute code is inserted into the array after an iteration, the uncertain cells are stored with the universal code U.

2.2 Theoretical Analysis

In this section, we provide theoretical analysis for the proposed approach, especially on the universal code U and the threshold rate T because they play key roles for programming effectiveness and memory conservation to the addressed problem. More specifically, the universal code U could solve collisions in the programming process, and thus keep the array at a reasonable size. However, it is conceivable that a large number of U could greatly increase the false-positive errors. The threshold rate T is to conserve the memory space because it represents the fraction of the hash cells of an element to be considered as a member. The fraction must contain either the element's attribute code or U. we suggest that it should be set at least 0.5 to avoid excess false-positive errors. Although T is pre-specified, its value could substantially affect the number of U, meaning a large T could dramatically increase the number of U. Therefore, we investigate the occurrence of U with regard to T as well as other related factors. The notations of these related factors are given as follows: the array size m, the number of attribute codes a, the number of elements with each attribute n, and the hash number h. We assume the number of elements n is the same for all attribute codes for the preliminary analysis.

The universal code U is only engaged when a cell is hit by multiple elements, and the probability of a cell hit by multiple elements is as in Equation (2).

$$P_x = 1 - P_0 - P_1, \tag{2}$$

where P_x is the probability of multiple hits, P_0 and P_1 are that of zero and single hit respectively.

Let H be the total number of hash hits by all the elements, so $H = a \cdot n \cdot h$. Therefore, we could derive P_0 and P_1 as in Equations (3) and (4).

$$P_0 = \frac{\binom{m}{1}}{\binom{H}{0}} = \frac{m}{H} \tag{3}$$

$$P_1 = \frac{\binom{m}{1}}{\binom{H}{1}} = \frac{m}{H}. \tag{4}$$

We then rewrite Equation (2) as in Equation (5).

$$P_x = 1 - \frac{2m}{H}. \tag{5}$$

P_x includes a portion of cells, where only one or none element does not meet T, which must be excluded out because they would not be represented by U. Therefore, we derive the probability of each element that could actually turn to U as in Equation (6).

$$P_y = \sum_{i=\lceil T \cdot h \rceil}^{h} \frac{1}{\binom{h}{i}\binom{a}{1}} = \sum_{i=\lceil T \cdot h \rceil}^{h} \frac{i!(h-i)!}{a \cdot h!}, \tag{6}$$

where P_y is the probability that the result of an element's hash hits meets T.

Let j be the hit number for each cell, and the probability of U could be designated as in Equation (7).

$$P_u = P_x \cdot (1 - \sum_{j=2}^{H} P_y^j). \tag{7}$$

In Equation (7), the second term in the parentheses represents the probability of not getting pass as a member when the number of hit elements is j.

Using Equation (7), when $h = 6$, $a = 100$, $n = 1,000$, $m = 40,000$, $T = 0.5$, P_x would be $2/15 = 0.1333$ according to Equation (5), and P_y would be approximately 0.1283 according to Equation (6). So the probability of U for each cell P_u is approximately 0.13.

3 Experimental Results

Normally, the performance metrics for the original Bloom filter or its variants include querying time, memory space and false-positive rate. For the proposed approach, we replace the bit array with a multibit one associated with proposed procedures for providing simultaneous attribute information when the membership is true Therefore, the time metric would not be a concern because the number of memory accesses is the same with the original Bloom filter, depending on the hash number. Additional operations such as counting the attribute codes and comparing with the threshold rate only accounts for a small fraction of the computational time. Under such a circumstance, we consider the memory space utilized for the array and the threshold rate as indicators for evaluating the false-positive rate in this section. The data element of the experiments was email addresses, which represent text strings with varied lengths. The dataset included 300,000 elements, while 100,000 others (outside the dataset) were used for evaluating the false-positive rates. Each of the set elements was assigned to a random attribute code between 1 and 14 since we used 4 bits (code values $0 \sim 15$) for each cell of the array. The value 0 designated the unused cells and 15 was the universal code that could represent all the attribute codes 1 14.

The memory space was designated by the number of bits utilized for the array. For example, the array would contain 750,000 cells if 3,000 K bits of memory space was utilized, because 3,000Kb divided by 4 bits for each cell equals to 750,000 cells. The hash number h was set to 5 because we investigated the memory space from 1,000 to 5,000K bits with an increment of 500K, whose m/n ratio was about 0.83 (1,000Kb) to 4.17 (5,000Kb), where the

highest ratio was close to h=5. Besides, the setting the threshold rates was from 20% to 100% with an interval of 20%, so there was a distinct count for each rate using h=5. For example, if the threshold rate is 60%, at least 3 out of the 5 hashing cells of the array must be the identical attribute code or the universal one to be recognized as a member with that attribute code.

With the settings of the memory space and the threshold rates, we conducted three experiments, each of which would include serval runs according to these settings. The first one was to investigate the array composition after the programming process, where the false-positive rates of different array sizes were also presented. The array composition could include zeros, attribute codes and the universals. The results are depicted in Figure 2.

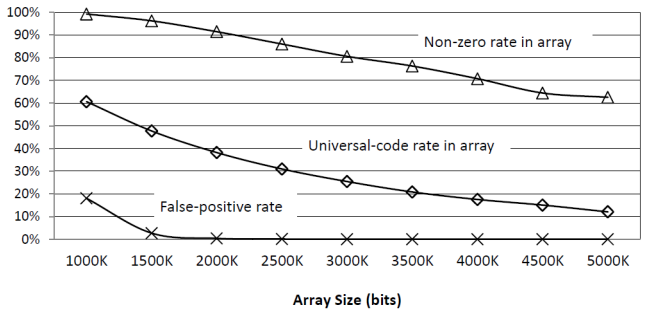

Figure 2: Array composition with varied array sizes after the programming phase

As shown in Figure 2, the non-zero rate started to stabilize when the memory space reached 4500K, while the universal-code rate continued to decrease. The situation was conceivable because the hashing process would address a similar portion of the array cells when memory space was sufficient. Each addressed cell would then be addressed by less set elements, which led to a less chance for issuing the universal code to that cell. As we can see in Figure 2, when the universal-code rate was under 40%, the false-positive rate became very close to zero. It was actually no false-positive errors at all when the rate of the universal codes was under 30% or the memory space was above 3,000K.

Since the array composition is important for an application to be successful as far as the performance is concerned, it was suggested that a healthy composition should include the portion of zeros near 50% of the array [3]. Therefore, in the second experiment, we further investigated both the non-zero and universal ratios in regard with the threshold rates. The non-zero codes certainly include the universal ones, however, we show separate results in Figures 3 and 4 in order to demonstrate the behavior of the universal-code rate because not only it is an important factor for our proposed approach to ensure the set elements to be successfully coded in the programming phase, but also represent a beacon for the FP error because the more number of universal-coded cells, the more chance of the FP errors.

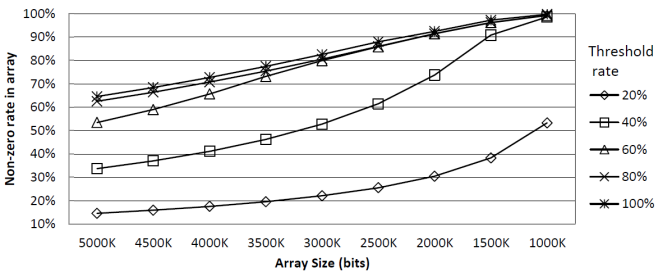

Figure 3: The array utilization rates of varied sizes with different threshold rates

As shown in Figure 3, the threshold rates above 40% (at least 2 out of 5 in the hashing addressed cells) reached a nearly full array utilization situation when the memory space is small. However, as we described that near 50% non-zero rate could be considered as a healthy composition of such a data structure and according to previous experiment that 3,000K of memory space was a suitable for a reasonably low FP error, we can see that at the threshold rate of 60% with 5,000K or 40% with 3,000K were suitable as the appropriate combinations.

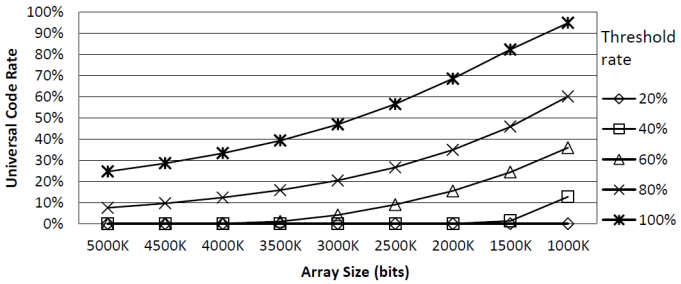

Figure 4: The universal-code rates of varied array sizes with different threshold rates

In Figure 4, it is reasonable that the universal-code rates decreased as the array size increased. For example, when the array size was 3,000Kb or above, the universal rates were all less than 50%, which were even smaller when the threshold rates decreased, *e.g.*, 20%. A small threshold rate was prone to associate with more FP errors because the low standard would easily claim membership for non-member elements. Therefore, it is important to determine suitable array size with the threshold rate for assuring the FP rate an acceptable level. According to the above-mentioned analyses, we recommend the 3,000Kb memory space for the array with the threshold rate of 40% in our specific case, whose m/n ratio is around 2.5. We also recommended the array size of 5,000K with 4.17 m/n ratio and 60% of threshold rate because it could achieve an even lower FP error. In the third experiment, nevertheless, we went through all combinations of these two factors, i.e., the array size and the threshold rate, for evaluating the FP rate as shown in Figure 5.

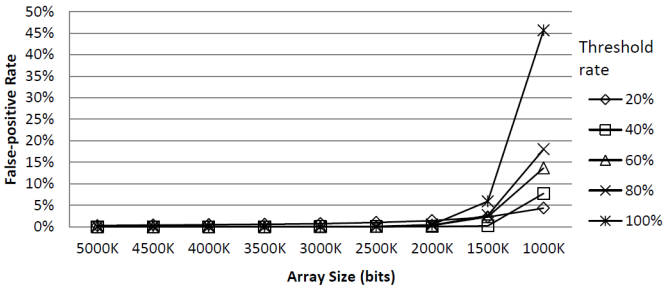

Figure 5: False-positive rates of threshold rate and array size combinations

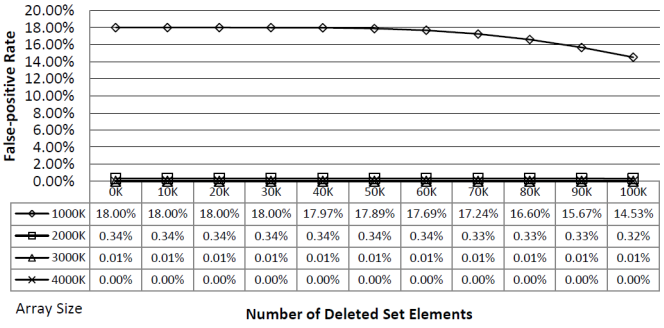

Figure 6: The false-positive rates of varied array sizes and the numbers of deletion

In Figure 5, the FP rates for all of the threshold rates were so close to 0% at the 4,000K memory space level, but the 100% and 80% of the threshold rates started to pick up as the memory space dropped to 3500K or less, which even went up to above 15% as that reduced to 1,000K. In the end, we still consider the memory space of 3,000K $\sim$ 4,000K of memory space, whose m/n ratio was 2.5 3.33 with threshold rates of 40 or 60% could be the best options for the designated case using the proposed approach.

Besides the three experiments, we conducted an extra one regarding dynamic element insertion and deletion of the dataset, which is important for certain applications requiring immediate adjustments of the set members. Therefore, the dynamic element insertion and deletion must include some mechanisms for encoding newly added or deleted members in real time, because the dataset shouldn't be reprogrammed upon slight changes of the dataset. The dynamic insertion for the proposed approach is to encode a new pair of member data (ex, ax) to the array, where ex is the new element and ax is the associated attribute code. The insertion could only be successful when the hash-addressed cells of ex in the array include ax. Once there is at least one ax, the insertion is guaranteed to be successful because the proposed approach uses the universal code U to accommodate all the attribute codes. As far as the dynamic deletion is concerned, we adopted an addition bit as the "sign" bit for each cell, whose values were initially zeros. When a member is determined to be disqualified from its membership, the sign bits of all the hash-addressed cells were turned to 1. With the proposed mechanism, some non-members would be judged as deleted members when all the sign bits of the hash-addressed are one, especially when the array size is insufficiently small, *e.g.*, 1,000Kb. We depicted this situation in Figure 6, where only the FP rate of the 1,000K memory space was shown to decrease as the number of the number of the dynamic deletion elements increased, while others remained low FP rates.

As to other members, only a fraction of cells or none whatsoever whose addressed sign bits are 1 and the attribute code is still in effect without considering the sign bit. However, if a member that was not dynamically deleted but addressed all cells with the sign bits of 1, it would be considered as a deleted member. Such a situation is referred to as a true-negative error, which is far serious than the one presented above because some members would be denied due to the sign-bit mechanism. We show the experimental results in Figure 7, where only an insufficient array size of 1,000K was significant on this situation. The true-negative errors could be a serious matter for some applications, because they represent a security hole where non-members are considered as true ones. Fortunately, the odds of such a situation are relatively slim as far as the array size is sufficient. Figure 7 shows the experimental results on this issue, where the true-negative errors only occurred when the array size was 1,000Kb. We show the details on the bottom of the figure.

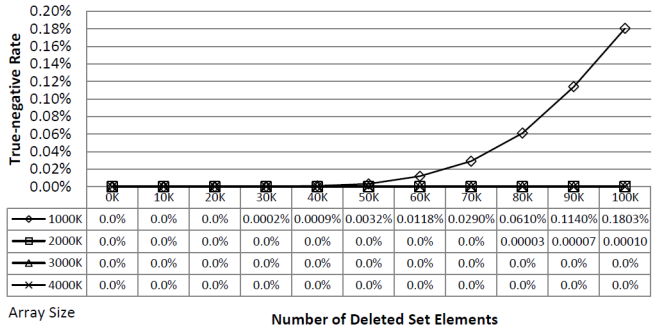

Figure 7: The true-negative rates of threshold rate and array size combinations

4 Conclusion

This study proposes a multibit representation for the original Bloom filter, whose purpose is to simultaneously provide the membership and the associated attribute information. We suggested a programming process that incorporates counting the attribute code and setting a threshold rate for the count percentage to exceed as a member. We also incorporate the universal code that accommodate all the attribute codes for the sake of expediting the programming process. Furthermore, we relieved the all-one

policy of the original Bloom filter by establishing multiple scales of threshold rate as the hurdle for determining a membership of the set elements. As the experimental results showed, we could select proper settings of these factors mentioned above after a pilot run was taken place, and we could then proceed the programming process accordingly until all the set elements are coded in the multi-bit array. Such a process also considered the FP rates as well as the array sizes with certain threshold rates at an acceptable level. The proposed approach would not elevate the computational overhead, neither the FP errors.

We also carried out an additional experiment concerning the dynamic insertion and deletion of elements of the dataset. The element insertion would be successful when the hash process addressed at least a cell whose attribute code was the same as the one of the inserting element because of the universal code; however, it would be fail when the stated-above condition does not stand. We are currently elaborating work around to establish ground work for such a matter. As far as the dynamic deletion is concerned, we strongly suggested establishing a bit array for recording the status of a deleted member, which worked well in our experiments, but required a minor addition of memory space, i.e., one bit for a cell in the array. The performance, however, was extraordinarily well because all the error rates would stay low as the memory space was relatively sufficient.

References

[1] M. H. Aghdam and P. Kabiri, "Feature selection for intrusion detection system using ant colony optimization," *International Journal of Network Security*, vol. 18, no. 3, pp. 420-432, 2016.

[2] 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.

[3] K. Christensen, A. Roginsky and M. Jimeno, "A new analysis of the false positive rate of a Bloom filter," *Information Processing Letters*, vol. 110, no. 21, pp. 944–949, 2010.

[4] H. Dai, Y. Zhong, A. X. Liu, W. Wang and M. Li, "Noisy Bloom Filters for multi-set membership testing," in *Proceedings of ACM Sigmetrics*, pp. 139-151, 2016.

[5] D. Ellison, "On the convergence of the multidimensional Albus perceptron," *The International Journal of Robotics Research*, vol. 10, pp. 338-357, 1991.

[6] M. Goodrich and M. Mitzenmacher, "Invertible bloom lookup tables," in *Allerton Conference on Communication, Control, and Computing*, pp. 792–799, 2011.

[7] F. Hao, M. Kodialam and 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.

[8] Y. Hua, B. Xiao, B. Veeravalli and D. Feng, "Locality-sensitive bloom filter for approximate membership query," *IEEE Transactions on Computers*, vol. 61, no. 6, pp. 817-830, 2012.

[9] C. W. Liu, W. F. Hsien, C. C. Yang and M. S. Hwang, "A survey of attribute-based access control with user revocation in cloud data storage," *International Journal of Network Security*, vol. 18, no. 5, pp. 900-916, 2016.

[10] P. Liu, H. Wang, S. Gao, T. Yang, L. Zou, L. Uden and X. Li, "ID bloom filter: Achieving faster multi-set membership query in network applications," in *Proceedings of IEEE ICC*, pp. 1-6, 2018.

[11] H. Ma, Y. C. Tseng and L. I. Chen, "A CMAC-based scheme for determining membership with classification of text strings," *Neural Computing with Applications*, vol. 27, pp. 1959-1967, 2016.

[12] Q. S. Qassim, A. M. Zin, and M. J. Ab Aziz, "Anomalies classification approach for network-based intrusion detection system," *International Journal of Network Security*, vol. 18, no. 63, pp. 1159-1972, 2017.

[13] 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.

[14] D. Thiyagarajan and R. Ganesan, "Cryptographically imposed model for efficient multiple keyword-based search over encrypted data in cloud by secure index using bloom

filter and false random bit generator," *International Journal of Network Security*, vol. 19, no. 3, pp. 413-420, 2017.

[15] B. Xiao, Y. Hua, "Using parallel bloom filters for multiattribute representation on network services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 1, pp. 20-32, 2010.

[16] T. Yang T, 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. 25, no. 5, pp. 3116-3131, 2016.

Biography

Ying-Chih Tseng is currently the deputy superintendent of Hsinchu Cathay General Hospital, Taiwan. He received his Ph.D. degree from the Ph.D. Program of Technology Management, Chung-Hua University, Taiwan. He is also an assistant professor in the Department of Information Management, Yuanpei University of Medical Technology, Taiwan. His research interests include medical image recognition and neural networks.

Heng Ma is currently a professor and the department chair of the Department of Industrial Management, Chung-Hua University, Taiwan. He received his Ph.D. degree from the Department of Industrial & Manufacturing

Engineering, The Pennsylvania State University in 1996, MS from the Department of Industrial & System Engineering, Ohio University in 1992, and BS from the Department of Industrial Engineering, National Tsing-Hua University, Taiwan, in 1988. His research interests include robotics, neural networks, image recognition, and network security.