2023 IEEE International Conference on Data Mining (ICDM) | 979-8-3503-0788-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICDM58522.2023.00015

Variable-length Encoding Framework: A Generic Framework for Enhancing the Accuracy of Approximate Membership Queries

Haipeng Dai, Hancheng Wang, Zhipeng Chen, Jiaqi Zheng, Meng Li, Rong Gu, Chen Tian, Wanchun Dou
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu 210023, CHINA
haipengdai@nju.edu.cn, {hanchengwang, zhipengchen}@smail.nju.edu.cn,
{jzheng, meng, gurong, tianchen, douwc}@nju.edu.cn

***Abstract*—Approximate membership query (AMQ) data structures can efficiently indicate whether an element exists in a data set. Therefore, they are widely used in data mining applications such as IoT streaming data mining, anomaly detection, duplicate detection, record linkage, and community discovery. The data amount to be processed in real-world applications often changes frequently and dynamically. Thus, before using the AMQ data structures, it is necessary to configure their capacity to the maximum number of elements that will be stored during runtime. We observe that when the number of elements stored in an AMQ data structure is lower than its capacity, a significant amount of space is wasted, making the false positive rate much higher than expected. To tackle this problem, we propose the variable-length encoding framework. It dynamically adjusts the encoding length of each element according to the number of elements stored in the AMQ data structure. Based on this design, the variable-length encoding framework can make full use of the memory space allocated to AMQ data structures, thereby improving the space efficiency and reducing the false positive rate. In addition, as a general encoding scheme, the variable-length encoding framework can be widely used in different types of AMQ data structures. Theoretical analysis and evaluation results show that AMQ data structures using the variable-length encoding framework have significantly lower false positive rates compared with state-of-the-art AMQ data structures. For example, when the load factor is 25%, the variable-length encoding framework can reduce the false positive rate of AMQ data structures by 88.15% on average (up to 99.40%).**

***Index Terms*—probabilistic data structure, approximate membership query, false positive rate, cuckoo filter**

I. Introduction

A. Motivation and Problem Statement

AMQ data structures are ubiquitous. Approximate membership query (AMQ) data structures (*e.g.*, Bloom filters [1], cuckoo filters [2], quotient filters [3], and their variants [4]–[16]) are a type of compact probabilistic data structures. AMQ data structures can approximately indicate whether an element exists in a set. Specifically, if the element e exists in the set, the lookup operation of the AMQ data structures always returns that the element e exists. Conversely, if element e does not exist in the set, AMQ data structures' lookup operations return with probability ε that element e exists. By allowing one-sided errors (*i.e.*, only false positives and no false negatives), AMQ data structures store the encoding of the partial information (*e.g.*, fingerprints) instead of the raw value of the element. This approach can achieve a trade-off between the space consumption and the false positive rate, that is, the average space consumption per element is $O(\log(1/\varepsilon))$ bits. In this sense, compared with directly storing original data, AMQ data structures are more space-saving. Therefore, AMQ data structures are widely used in applications that need to process massive data, such as anomaly detection [17], [18], duplicate detection [19], [20], record linkage [21], community discovery [22], entity linking [23], streaming data mining [19], [24]–[27], web data mining [28], big data mining [29]–[31], and spatio-temporal data mining [21], [32], [33].

Problem statement. We observe that for the above applications, the data amount that needs to be processed by applications often changes dynamically. Therefore, before using the AMQ data structures, it is necessary to configure their capacity to the maximum number of elements that will be stored during runtime. This inevitably results in much space unused in the AMQ data structure when the number of stored elements is less than the capacity. Specifically, assuming that the space allocated to an AMQ data structure is M, when the number of inserted elements n is much smaller than the capacity of the AMQ data structure, the space occupied by each element (denoted as l_1) is much smaller than its allocated space $l_2 = M/n$. This problem leads to the underutilization of the allocated space, making the false positive rate of AMQ data structures much higher than that when the l_2 bits are fully utilized for each element. In summary, when the number of elements stored in the AMQ data structure is less than the capacity, the space efficiency is low, resulting in a high false positive rate.

Tackling this problem is important. Tackling this problem can reduce the false positive rate of AMQ data structures and make approximate membership queries more accurate. Furthermore, tackling this problem can reduce the overhead of eliminating false positives in systems using AMQ data structures, which is very important for improving the performance of data mining systems using AMQ data structures [12], [16], [30], [32], [34]–[36].

B. Limitations of Prior Art

The existing AMQ data structures mainly use a fixed-length encoding scheme to store partial information. For example, cuckoo filters encode each element as an l_f-bit fingerprint.

979-8-3503-0788-7/23/$31.00 ©2023 IEEE
DOI 10.1109/ICDM58522.2023.00015

Bloom filters generate k hash values for each element. Such AMQ data structures gradually consume the memory space allocated to them as the elements are inserted. Obviously, for an AMQ data structure using a fixed-length encoding scheme, when the number of elements stored in the data structure is significantly lower than its capacity, a large amount of space is unutilized in the data structure. Note that although we can use resizing technique to avoid allocating too much space for AMQ data structures at one time and improve space efficiency, frequent expansion and contraction will reduce the throughput and increase the false positive rate. To alleviate this problem, we propose a light-weight solution, *i.e.*, by adjusting the encoding length of each element according to the number of inserted elements, the AMQ data structures can make full use of all the memory space allocated.

C. Proposed Approach

We propose a framework that can improve the space efficiency and reduce the false positive rate of AMQ data structures. This encoding framework stores as much encoding of the partial information as possible for each element by making full use of all the memory space allocated. This design enhances the space efficiency and reduces the false positive rate of AMQ data structures. We name this encoding framework as the variable-length encoding framework. For AMQ data structures with a dynamic number of elements, using this framework can significantly enhance their accuracy.

Specifically, the core mechanism for reducing the false positive rate of AMQ data structures by using the variable-length encoding framework is as follows: (i) When the number of elements stored in the AMQ data structure is small, the variable-length encoding framework stores additional encoding of the partial information for each element by using an encoding length longer than that used in the fixed-length encoding scheme. Therefore, even when the AMQ data structure contains a small number of elements, the variable-length encoding framework can fully utilize the allocated memory space. (ii) As the number of elements in the AMQ data structure increases, to accommodate newly added elements, the variable-length encoding framework can gradually compress previously inserted elements by shortening them to the encoding length used in the fixed-length encoding scheme. This design can reduce the false positive rate without reducing the total capacity of the AMQ data structure.

The encoding length for each element stored in AMQ data structures using the variable-length encoding framework is always not less than that in AMQ data structures using the fixed-length encoding scheme. Therefore, the variable-length encoding framework can store the encoding of additional partial information for each element, thereby reducing the false positive rate of the AMQ data structure. We will theoretically analyze the impact of the variable-length encoding framework on reducing the false positive rate in Section V.

D. Key Technical Challenges

The first technical challenge is how to implement the variable-length encoding framework without incurring a large computational overhead. AMQ data structures are a type of lightweight data structures. Usually, AMQ data structures perform millions to tens of millions of insertion, lookup, and deletion operations per second [2], [37]. For such high-throughput data structures, the additional computational overhead will significantly affect their throughput. Thus, it is difficult for the variable-length encoding framework to reduce the false positive rate of AMQ data structures without affecting the throughput. We mitigate the impact of the variable-length encoding framework on the throughput of AMQ data structures by using bit manipulation instructions to accelerate the variable-length encoding and decoding process.

The second technical challenge is how to extend the variable-length encoding framework to various AMQ data structures. Different AMQ data structures have different designs. Thus, it is challenging to propose a framework that can commonly improve the performance of different AMQ data structures. The design of the variable-length encoding framework can be widely applied to various AMQ data structures because it is based on the common mechanism of different AMQ data structures. In Section II, we divide existing AMQ data structures into two categories and illustrate how these two categories of AMQ data structures use the variable-length encoding framework.

E. Key Contributions

We have four main contributions:

(1) We have an observation that when the number of inserted elements in the AMQ data structure is lower than its capacity, there is a lot of space waste, which in turn increases the false positive rate. Furthermore, we also analyze the constraints of the existing work.

(2) We propose a variable-length encoding framework that can commonly improve the space efficiency of different types of AMQ data structures, reducing the false positive rate of AMQ data structures. The variable-length encoding framework adjusts the encoding length according to the number of inserted elements in the AMQ data structures, making full use of the memory space allocated to the AMQ data structures.

(3) We theoretically analyze the false positive rate of AMQ data structures using the variable-length encoding framework. Furthermore, we set the parameters of the variable-length encoding framework by using the conclusions drawn from the theoretical analysis.

(4) Experimental results show that the AMQ data structures using the variable-length encoding framework have significantly lower false positive rates compared with state-of-the-art AMQ data structures. For example, when the load factor is 25%, the variable-length encoding framework can reduce the false positive rate of AMQ data structures by 88.15% on average (up to 99.40%).

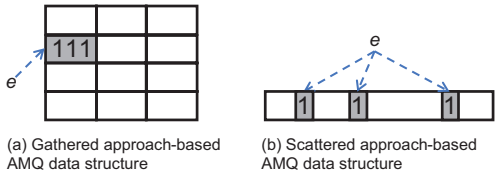


Fig. 1. Example of inserting element e using two different types of AMQ data structures.

II. Variable-Length Encoding Framework

In this section, we introduce the overall scheme of the variable-length encoding framework and describe how to apply it to different types of AMQ data structures.

The variable-length encoding framework is a generic framework that can improve space efficiency and reduce the false positive rate of AMQ data structures. By adjusting the encoding length of elements based on the number of inserted elements, the variable-length encoding framework can better utilize the allocated memory space to reduce the false positive rate. Specifically, (i) When the number of elements stored in the AMQ data structure is small, the variable-length encoding framework stores additional encoding of the partial information for each element by using a longer encoding length for each element. Therefore, the allocated memory space can be effectively utilized even if only a small number of elements are inserted. (ii) As the number of elements in the AMQ data structure increases, the variable-length encoding framework compresses previously inserted elements to accommodate newly added elements. This design avoids reducing the total capacity of the AMQ data structure.

We will describe how to apply the variable-length encoding framework to different types of AMQ data structures in Sections III and IV in detail. Specifically, as shown in Figure 1, the existing AMQ data structures can be divided into two categories according to how partial information is stored: gathered approach-based AMQ data structures and scattered approach-based AMQ data structures. (i) The gathered approach-based AMQ data structures (*e.g.*, cuckoo filters and their variants) encode elements into fingerprints. The partial information for each element is gathered in memory. (ii) The scattered approach-based AMQ data structures (*e.g.*, Bloom filters) map each element to k different memory addresses. The partial information for each element is scattered in memory. The two types of AMQ data structures have different ways of storing the encoding information. Therefore, they have differences in how the variable-length encoding framework is used. The variable-length encoding framework can be employed in both types of the above AMQ data structures, because the variable-length encoding framework leverages the common features of the AMQ data structures to reduce their false positive rate.

III. Gathered Approach-Based AMQ Data Structures

Gathered approach-based AMQ data structures mainly include cuckoo filters and their variants. This type of AMQ data structures approximately represents elements by encoding

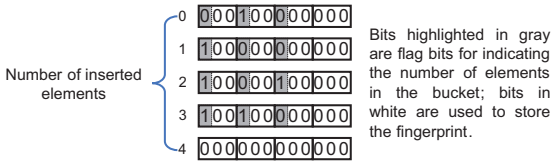


Fig. 2. An example of the encoding scheme when inserting a varying number of elements into a bucket. Each bucket has four slots, and each slot has three bits. The highest bit (highlighted in gray) of the first three buckets is used to indicate the number of elements in the current bucket, and the other bits (white part) are used to store the fingerprints of inserted elements. For example, when there is only one element in the bucket, a 9-bit fingerprint can be stored. In addition, when there are four elements in the bucket, the first three fingerprints are arranged in an increasing order to avoid encoding conflicts when the number of elements in the bucket is less than 4.

elements into fixed-length fingerprints and storing them in a hash table. As shown in Figure 1, unlike Bloom filters, which store each element in k different locations, this type of AMQ data structures stores each element in a slot. Therefore, we name this type of AMQ data structures gathered approach-based AMQ data structures.

Gathered approach-based AMQ data structures have a three-level storage structure of hash tables, buckets, and slots. A hash table consists of a series of buckets, where elements sharing the same hash index are stored in the same bucket. A bucket consists of a series of slots, and each slot can store a fingerprint. For gathered approach-based AMQ data structures, when the number of fingerprints stored in the bucket is less than the bucket capacity, there are unutilized empty slots within the bucket. After using the variable-length encoding framework for this type of AMQ data structures, even if only one element is inserted to the bucket, all the slots in the bucket will be occupied. Therefore, the variable-length encoding framework makes full use of the space in the bucket to store more information for each element. To achieve the above goals, firstly, the variable-length encoding framework needs to support encoding and decoding fingerprints with different lengths. Then, the variable-length encoding framework needs to employ the above encoding and decoding schemes in the insertion, lookup, and deletion operations of AMQ data structures. Next, we will introduce the above designs in turn.

Encoding and decoding schemes. For gathered approach-based AMQ data structures, if there are b slots in each bucket and each slot has l_f bits, each bucket has $b \cdot l_f$ bits. For simplicity, we will take four slots per bucket as an example to illustrate our design (Figure 2). Note that the variable-length encoding framework also works for buckets with a different number of slots. We use four slots per bucket as an example because four slots per bucket is a typical setup for such AMQ data structures [2], [37], [38]. Next, we illustrate the encoding and decoding schemes when inserting different numbers of elements into buckets with four slots.

(1) When there is no fingerprint inserted to the bucket, we set the highest bit of the first three slots to 0, 1, and 0. Therefore, if the highest bit of the first three slots are 0, 1, and 0 during decoding, it means that there is no fingerprint in the bucket.

(2) When there is one fingerprint inserted to the bucket, we

set the highest bit of the first three slots to 1, 0, and 0. The remaining $bf-3$ bits in the bucket can be used to store this fingerprint. Therefore, if the highest bit of the first three slots are 1, 0, and 0 during decoding, it means that there is one fingerprint inserted to the bucket, and this fingerprint is stored in the remaining $bf-3$ bits in the bucket.

(3) When there are two fingerprints inserted to the bucket, we set the highest bit of the first three slots to 1, 0, and 1. The two fingerprints in the bucket are sequentially stored in the remaining $bf-3$ bits in the bucket, that is, each fingerprint occupies $(bf-3)/2$ bits. Therefore, if the highest bit of the first three slots are 1, 0, and 1 during decoding, it means that there are two fingerprints inserted to the bucket, and each fingerprint occupies $(bf-3)/2$ bits.

(4) When there are three fingerprints inserted to the bucket, we set the highest bit of the first three slots to 1, 1, and 0. The three fingerprints in the bucket are sequentially stored in the remaining $bf-3$ bits in the bucket, that is, each fingerprint occupies $(bf-3)/3$ bits. Therefore, if the highest bit of the first three slots are 1, 1, and 0 during decoding, it means that there are three fingerprints inserted to the bucket, and each fingerprint occupies $(bf-3)/3$ bits.

(5) When there are four fingerprints inserted to the bucket, the number of fingerprints in the bucket reaches its capacity. We store the four fingerprints in the four slots respectively, and then adjust the order of the fingerprints in the first three slots according to the fingerprint value to ensure that the fingerprints in the three slots are arranged in ascending order. Obviously, the highest bit of the first three slots can only be 000, 001, 011, or 111. These four values do not conflict with any previous encodings. Therefore, if the highest bit of the first three slots is one of 000, 001, 011, and 111 during decoding, it means that there are four fingerprints inserted to the bucket, and each slot is an l_f-bit fingerprint.

To sum up, when the number of elements in the bucket is less than 4, we set the highest bit of the first three slots to special values to identify the number of fingerprints in the current bucket. When the number of elements in the bucket is 4, by adjusting the order of the fingerprints, we ensure that the highest bit of the first three slots do not conflict with the special values when the number of elements is less than 4. The above encoding scheme only uses the bits in the buckets, and can distinguish buckets with different numbers of fingerprints without requiring additional memory space. Furthermore, the above encoding scheme can be extended to cases where the number of slots in each bucket is not four. Specifically, we set the highest bit of a slot to a special value when the number of fingerprints in the bucket is less than the number of slots. When the number of fingerprints is equal to the number of slots, we can obtain the number of fingerprints in the bucket by adjusting the order of the fingerprints to ensure that they do not conflict with the above special values. Note that using the above encoding and decoding schemes to improve accuracy will incur additional computational overhead. We use PEDP and PEXT bit manipulation instructions to speed up the reading and writing of fingerprints, avoiding excessive performance degradation (Section VI).

Next, we will take cuckoo filters as an example to illustrate how to apply the above design to the existing AMQ data structures. Note that our encoding-decoding scheme is orthogonal to cuckoo filters. Therefore, the variable-length encoding framework can also be applied to other gathered approach-based AMQ data structures. In addition, the variable-length encoding framework does not require modifications to the insertion, lookup, and deletion mechanisms of the AMQ data structures. Therefore, the variable-length encoding framework can be easily applied to existing AMQ data structures.

Insertion. For standard cuckoo filters, each element has two candidate buckets, *i.e.*, i_1 and i_2. For an element e, if the bit length of its fingerprint is l_f, the fingerprint of the element (denoted as f) and two candidate buckets (denoted as i_1 and i_2) can be calculated as follows.

$$f = \text{fingerprint}\,(e) \bmod 2^{l_f}. \tag{1}$$

$$i_1 = hash\,(e)\,; i_2 = i_1 \oplus hash\,(f)\,. \tag{2}$$

In the cuckoo filters using the variable-length encoding framework, the bit length of the fingerprints changes dynamically. To ensure that the two candidate bucket indexes of an element do not change with the bit length of the fingerprint, we use the low l_f bits of the variable-length fingerprint when calculating the candidate bucket indexes. Note that l_f is the shortest bit length to which variable-length fingerprints can be compressed. Therefore, for an element e, if its variable-length fingerprint is f', and the bit length of f' is $l_{f'}$ ($l_f \leq l_{f'}$), then the element's fingerprint (denoted as f') and two candidate buckets (denoted as i'_1 and i'_2) can be calculated as follows.

$$f' = \text{fingerprint}\,(e) \bmod 2^{l_{f'}}. \tag{3}$$

$$i'_1 = hash\,(e)\,; i'_2 = i'_1 \oplus hash\left(f' \bmod 2^{l_f}\right). \tag{4}$$

The subsequent insertion operations of the cuckoo filters using the variable-length encoding framework are the same as the standard cuckoo filters at the bucket level. They both insert fingerprints using the cuckoo eviction mechanism. Specifically, if the two candidate buckets are not full yet, we use the encoding and decoding scheme mentioned above to insert the fingerprint f' into a candidate bucket. Otherwise, we randomly kick a fingerprint f'' from the candidate bucket to store the fingerprint f'. If the other candidate bucket of the kicked out fingerprint f'' is not yet full, we store the kicked out fingerprint f'' in another candidate bucket, and return the insertion success. Otherwise, we kick a fingerprint again from another candidate bucket and repeat the above process. If the number of repetitions reaches a certain threshold (typically 500), the insertion fails.

The variable-length encoding framework does not change the insertion mechanism of cuckoo filters. Therefore, cuckoo filters using the variable-length encoding framework have the same time complexity as standard cuckoo filters.

Lookup. The lookup operation of cuckoo filters using the variable-length encoding framework is similar to that of standard cuckoo filters. The lookup operation of cuckoo filters

using the variable-length encoding framework first calculates the indexes of two candidate buckets according to Equation (4). Then, the variable-length encoding framework decodes the fingerprints in the two candidate buckets and checks whether there is a fingerprint matching the query element e. If there is a fingerprint matching the element e, it indicates that the element e exists, otherwise, it does not exist. As the lookup operation of cuckoo filters using the variable-length encoding framework only needs to access two candidate buckets, the time complexity of the lookup operation is $O(1)$.

Deletion. The deletion of cuckoo filters using the variable-length encoding framework first calculates the indexes of two candidate buckets according to Equation (4). Then, unlike standard cuckoo filters that delete any fingerprint that matches element e, cuckoo filters using the variable-length encoding framework delete the longest fingerprint that matches element e in the candidate buckets. This design can avoid false negatives. For example, for two elements e_1 and e_2 that have been inserted to the data structure. If e_1 matches fingerprints f_1 and f_2 and the bit length of f_1 is longer than f_2. e_2 only matches fingerprint f_2 and does not match fingerprint f_1. If the fingerprint f_2 is removed when performing the deletion on element e_1, it will cause a false negative when querying element e_2. Therefore, cuckoo filters using the variable-length encoding framework delete the longest fingerprint matching element e when performing the deletion operation. In addition, as the deletion operation of cuckoo filters using the variable-length encoding framework only needs to access two candidate buckets, the time complexity of the deletion operation is $O(1)$.

Analysis of computational overhead. Using the variable-length encoding framework to reduce the false positive rate brings additional computational overhead. However, the additional overhead is acceptable:

(i) The computational overhead brought by the variable-length encoding framework is limited. We speed up the process of variable-length encoding and decoding by using two bit manipulation instructions including PDEP and PEXT. The above two instructions are now widely supported by a large number of processors including the Intel Haswell line of processors, AMD Excavator, and their subsequent generations [39]. Therefore, it is a generic way to use the above instructions to speed up variable-length encoding and decoding processes. Based on our evaluation, cuckoo filters using the aforementioned instructions exhibit a mere 7.01% reduction in insertion throughput compared with standard cuckoo filters. Thus, the computational overhead brought by the variable-length encoding and decoding process is acceptable.

(ii) It is meaningful to reduce the false positive rate of AMQ data structures, even if it may bring higher computational overhead. Specifically, when false positives occur in the AMQ data structure, applications using this structure may incur even higher computational overheads to eliminate these false positives. For example, log-structured merge-trees use AMQ data structures to determine whether an element is stored on a certain disk block to avoid unnecessary disk block accesses. Log-structured merge-trees can avoid accessing the disk block if the AMQ data structure correctly reports that the element does not exist in the disk block. However, if the AMQ data structure incorrectly reports that the element exists in a certain disk block (*i.e.*, a false positive during the lookup operation), log-structured merge-trees will access a useless disk block, leading to additional overhead. By reducing the false positive rate, the variable-length encoding framework can effectively reduce the additional overhead of the application caused by the false positives of the AMQ data structure. Therefore, it is worthwhile to reduce the false positive rate of AMQ data structures at the cost of increasing the computational overhead.

IV. Scattered Approach-Based AMQ Data Structures

Scattered approach-based AMQ data structures mainly include Bloom filters and their variants. The insertion operation of this type of AMQ data structures obtains k storage positions corresponding to elements in the array by calculating k different hash functions. Note that the optimal number of hash functions k_{opt} is not a fixed value. Taking a Bloom filter as an example, if the length of the array is m, when the number of inserted elements in the Bloom filter is n, the optimal number of hash functions for the insertion operation is $k_{opt} = \frac{m}{n} \ln 2$. In other words, with a fixed array length m, the number of optimal hash functions for Bloom filters' insertion operations decreases as the number of inserted elements increases. However, the insertion operation of Bloom filters always uses a fixed number of k hash functions. k does not change dynamically based on the number of inserted elements during insertion.

The variable-length encoding framework can dynamically adjust the number of hash functions according to the number of inserted elements during insertion. Specifically, for an AMQ data structure with a capacity of N, when the number of elements in the AMQ data structure is less than $\alpha \cdot N$ ($0 < \alpha < 1$), the variable-length encoding framework uses k' hash functions ($k' > k$). Because the above design stores additional information for each element, the accuracy of the AMQ data structure can be enhanced. In Section V, we analyze the effect of the variable-length encoding framework on enhancing accuracy and have a discussion on how to set the values of α and k'. When the number of elements exceeds the predefined threshold $\alpha \cdot N$, the variable-length encoding framework deletes the additional information stored for each element to accommodate newly inserted elements. Then, the variable-length encoding framework reduces the number of hash functions used in the insertion operation to k. Therefore, the variable-length encoding framework can improve the accuracy of the AMQ data structure only when the number of elements is less than the threshold. Because Bloom filters do not support deletion operations, we take counting Bloom filters as an example to introduce the above design in more detail from the three aspects, including insertion, lookup, and deletion.

Insertion. Standard counting Bloom filters consist of an array of length m. Each element in the array is a c-bit counter. We use C to denote the array of counters. In the insertion operation of standard counting Bloom filters, when inserting an element e, the operation increments the counters corresponding to the

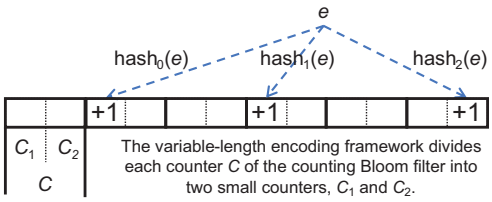


Fig. 3. An example of enhancing a counting Bloom filter using the variable-length encoding framework. In this example, the counting Bloom filter originally uses two hash functions (*i.e.*, $hash_0$ and $hash_1$). An additional hash function, $hash_2$, is added when the variable-length encoding framework is used. When inserting an element to the counting Bloom filter using the variable-length encoding framework, the filter first calculates the index of C_1 counter by using hash functions $hash_0$ and $hash_1$ and increases the value of $C_1[hash_0]$ and $C_1[hash_1]$ counters; then the filter calculates the index of C_2 counter by using hash function $hash_2$ and increases the value of $C_2[hash_2]$ counter. With this design, we can remove all additional hash function increments by directly zeroing out the C_2 of all counters.

k hash functions by one, following Equation (5).

$$C\left[hash_i\left(e\right)\right] = C\left[hash_i\left(e\right)\right] + 1, i \in [0, k). \tag{5}$$

Counting Bloom filters using the variable-length encoding framework also consist of an array of length m. The difference lies in the design of the counters in the array. Specifically, the insertion operation can be divided into two phases:

(i) When the number of inserted elements is less than the threshold $\alpha \cdot N$, each c-bit counter is divided into two separate $c/2$-bit counters with a smaller size, denoted as C_1 and C_2. The C_1 counters store the basic encoding information of the inserted elements (*i.e.*, the encoding information generated by the k hash functions). The C_2 counters store the additional encoding information of the inserted elements (*i.e.*, the encoding information generated by the $k' - k$ hash functions). Specifically, for the element e to be inserted, the counting Bloom filters using the variable-length encoding framework first increase the values of k C_1 counters corresponding to the k hash functions by one according to Equation (6). Then, according to Equation (7), the values of $(k' - k)$ C_2 counters corresponding to the $k' - k$ hash functions are increased by one. Figure 3 shows an example of the above process.

$$C_1\left[hash_i\left(e\right)\right] = C_1\left[hash_i\left(e\right)\right] + 1, i \in [0, k). \tag{6}$$

$$C_2\left[hash_i\left(e\right)\right] = C_2\left[hash_i\left(e\right)\right] + 1, i \in [k, k'). \tag{7}$$

(ii) Once the number of inserted elements is greater than the threshold $\alpha \cdot N$, to accommodate newly inserted elements, we first delete the additional encoding information stored for the previously inserted elements by clearing the values of the C_2 counters. Afterwards, the $c/2$ bits of each C_2 counter are merged into the corresponding C_1 counter to form a c-bit large counter. For newly inserted elements, Equation (5) is used to calculate k counters corresponding to k hash functions. The values of these counters are increased by one. Obviously, when the number of elements exceeds the threshold, the counting Bloom filters using the variable-length encoding framework are converted to standard counting Bloom filters. The accuracy can no longer be improved. Furthermore, this conversion process is irreversible. This is because the values of C_2 counters cannot be recovered after being cleared. Therefore, even if the number of inserted elements is less than the threshold after a number of deletion operations, the counters cannot be split into two counters with a smaller size again.

Lookup. The lookup operation of counting Bloom filters using the variable-length encoding framework can be divided into two phases: (i) When the number of inserted elements is less than the threshold, if the values of k C_1 counters and the values of $(k' - k)$ C_2 counters corresponding to the element e are not 0, the element exists, otherwise the element does not exist. (ii) Once the number of inserted elements is greater than the threshold, if the values of k counters corresponding to element e are not 0, the element exists, otherwise the element does not exist. Obviously, the time complexity of the lookup operation of counting Bloom filters using the variable-length encoding framework is $O(1)$.

Deletion. The deletion operation of counting Bloom filters using the variable-length encoding framework can also be divided into two phases: (i) When the number of inserted elements is less than the threshold, the delete operation reduces the values of k C_1 counters and the values of $(k' - k)$ C_2 counters corresponding to the element e by one; (ii) When the number of inserted elements is greater than the threshold, the deletion operation reduces the values of k counters corresponding to element e by one, just like the standard counting Bloom filter.

V. Theoretical Analysis

We analyze the false positive rate of two types of AMQ data structures using the variable-length encoding framework.

A. Gathered approach-based AMQ data structures

We take the cuckoo filters as an example to analyze the improvement of the false positive rate.

***Theorem** 5.1:* For cuckoo filters using the variable-length encoding framework with m buckets, the expected false positive rate after inserting n elements is given by

$$\varepsilon = 2\sum\nolimits_{s=1}^{b} P\left(X = s\right)\frac{s}{2^{l_s}}, \tag{8}$$

where b represents the number of slots in each bucket, s represents the number of elements stored in each bucket, l_s represents the bit length of the variable-length fingerprint of each element when s elements are stored in the bucket, $P\left(X = s\right) = \begin{cases} e^{\lambda}\lambda^s/s!, & s < b, \\ 1 - \sum_{s=0}^{b-1} P\left(X = s\right), & s = b, \end{cases}$ represents the probability of storing s elements in the bucket, and $\lambda = n/m$ represents the expectation of storing s elements in the bucket.

Proof: For cuckoo filters using the variable-length encoding framework with m buckets, when n elements are inserted, the number of elements inserted in each bucket approximately follows a Poisson distribution with parameter $\lambda = n/m$. Note that although the cuckoo eviction mechanism of cuckoo filters (Section III) makes the number of elements in each bucket slightly deviate from the Poisson distribution. But cuckoo evictions mainly occur at high load factors (>75%).

Therefore, in most cases, the number of elements in each bucket follows the Poisson distribution. In addition, our verification experiments also show that it is accurate to use Poisson distribution to calculate the number of elements in each bucket. Therefore, for a bucket that can store b elements, the probability of storing s elements in the bucket is given by

$$P(X=s)=\begin{cases} e^{\lambda}\lambda^{s}/s!, & s<b, \\ 1-\sum_{s=0}^{b-1}P(X=s), & s=b, \end{cases} \tag{9}$$

where $\lambda = n/m$ represents the expected number of elements stored in each bucket. For a bucket that stores s elements, if the length of the variable-length fingerprint of each element is l_s, when the lookup operation checks the bucket, the probability of matching a wrong fingerprint and thus causing a false positive is $1-\left(1-1/2^{l_s}\right)^{s} \approx s/2^{l_s}$. Therefore, the expected false positive rate caused by each bucket is $\sum_{s=1}^{b}P(X=s)\frac{s}{2^{l_s}}$. Because the lookup of cuckoo filters needs to check the fingerprints in two buckets, the expected false positive rate of cuckoo filters using the variable-length encoding framework is given by

$$\varepsilon = 2\sum\nolimits_{s=1}^{b}P(X=s)\frac{s}{2^{l_s}}. \tag{10}$$

This completes the proof. ■

B. Scattered approach-based AMQ data structures

We take the counting Bloom filters as an example to analyze the improvement of the false positive rate.

***Theorem** 5.2:* For counting Bloom filters using the variable-length encoding framework with m c-bit counters, the expected false positive rate after inserting n elements is given by

$$\varepsilon=\begin{cases} \left(1-e^{-k_1 n/m}\right)^{k_1}, & 0\le n\le \alpha N, \\ \left(1-e^{-k_2 n/m}\right)^{k_2}, & n>\alpha N, \end{cases} \tag{11}$$

where $k_1=\frac{m}{\alpha N}\ln 2$ represents the number of hash functions used in the first phase of the insertion, $k_2=\frac{m}{N}\ln 2$ represents the number of hash functions used in the second phase of the insertion, N represents the capacity of the data structure, and the insertion enters the second phase when the number of elements inserted is greater than αN.

We omit the experimental verification of Theorems 5.1, 5.2, and the proof of Theorem 5.2 due to space limitation. We define the average false positive rate of the data structure as $\sum_{i=0}^{n}\varepsilon(i)$. According to Theorem 5.2, we can calculate the threshold α that minimizes the average false positive rate.

VI. Evaluation

A. Experimental Setup

1) Implementation and Platform: We apply the variable-length encoding framework to five state-of-the-art AMQ data structures and evaluate their performance. We implement all algorithms in C++ and make them publicly available[1]. We perform all experiments on a server equipped with Intel(R) Xeon(R) Gold 5218R CPU (2.10GHz, 27.50MB L3 cache), 64GB RAM, and 1TB SSD running Linux 4.18.0. In addition, all algorithms are compiled with g++ 11.3.0.

[1]https://github.com/wanghanchengchn/variable-encoding-framework

2) Comparison Algorithms: We apply variable-length encoding framework to the Cuckoo Filter (**CF**) [2], the Tagged Cuckoo Filter (**TCF**) [40], the Counting Bloom Filter (**CBF**) [5], the variable-Increment Counting Bloom Filter (**ICBF**) [41], and the Tandem Counting Bloom Filter (**TCBF**) [42]. We named the above algorithms using the variable-length encoding framework as Variable-length encoding Cuckoo Filter (**VCF**), Variable-length encoding Tagged Cuckoo Filter (**VTCF**), Variable-length encoding Counting Bloom Filter (**VCBF**), Variable-length encoding variable-Increment Counting Bloom Filter (**VICBF**), and Variable-length encoding Tandem Counting Bloom Filter (**VTCBF**). The above algorithms include two types of AMQ data structures based on the gathered method and the scattered method. Specifically, CF and TCF belong to gathered approach-based AMQ data structures. CBF, VCBF, and TCBF belong to scattered approach-based AMQ data structures. As far as we know, they are the latest AMQ data structures that reduce the false positive rate of AMQ data structures whose number of elements changes dynamically. Note that adaptive filters and learning filters also claim that they can reduce the false positive rate of AMQ data structures. However, these works are mainly aimed at application scenarios where the number of elements does not change frequently or with some prior knowledge. The variable-length encoding framework proposed in this paper is orthogonal to these algorithms, so we did not compare with them.

3) Metrics: We use the following four metrics in evaluations.

False positive rate. The false positive rate can be obtained by measuring the proportion of erroneous cases where non-existent elements are mistakenly reported as present when querying for elements that are not in the AMQ data structure.

Bits per element. The bits per element can be calculated as the ratio between the number of bits occupied by the AMQ data structure and the total number of inserted elements.

Load factor. Load factor can be calculated as the ratio between the number of inserted elements and the capacity of the AMQ data structure (the expected maximum number of inserted elements).

Throughput. Throughput is measured by the number of operations performed per second. The unit of throughput is MOPS (Million Operations Per Second).

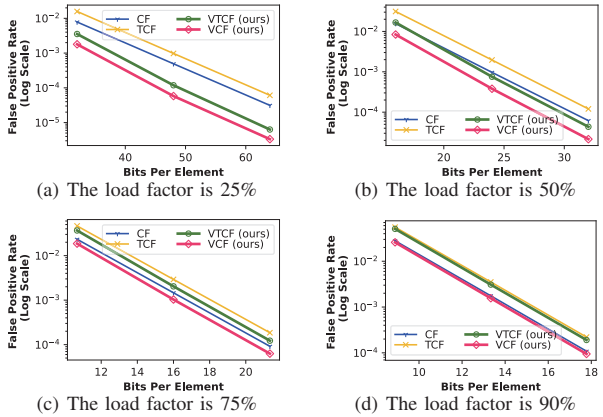
4) Datasets: The datasets are as follows:

Synthetic. We use BobHash [43] to randomly generate 134 million (2^{27}) distinct 64-bit integers for evaluation. Unless otherwise stated, we use this dataset as the default dataset.

WIKI. The WIKI dataset records the timestamps of submission by Wikipedia editors. It contains 90437011 different timestamps [44].

YCSB. We use the Yahoo! Cloud Serving Benchmark (YCSB) generator [45] to generate 75 million distinct elements. The above datasets (or generators) are all publicly available.

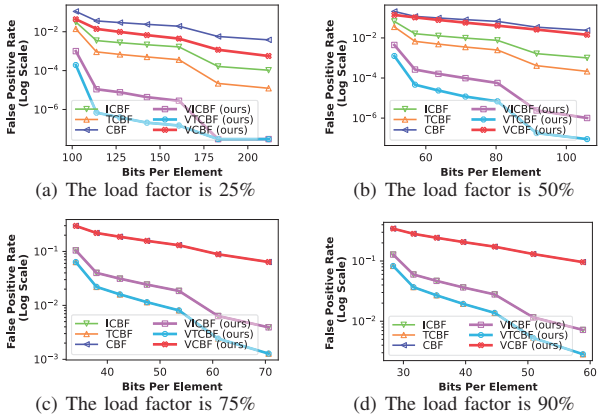
5) Parameter Settings: We compared the false positive rates of different algorithms with the same bit per element. We compared the throughput of different algorithms with the same memory space. Other parameters are set by default and are also publicly accessible. In addition, the performance of the



(a) The load factor is 25% (b) The load factor is 50%

(c) The load factor is 75% (d) The load factor is 90%

Fig. 4. Evaluation on false positive rate of gathered approach-based AMQ data structures (lower is better).



(a) The load factor is 25% (b) The load factor is 50%

(c) The load factor is 75% (d) The load factor is 90%

Fig. 5. Evaluation on scattered approach-based AMQ data structures.

variable-length encoding framework is mainly affected by the threshold triggering encoding length change α and the encoding length. Based on the theoretical analysis, we set α to 0.5, and the encoding length to twice the fixed encoding length used by its modified algorithm. For example, for a counting Bloom filter using 4 hash functions, 8 hash functions are used after using the variable-length encoding framework.

B. False Positive Rate

Our results show that the variable-length encoding framework can significantly reduce the false positive rate. We measure the false positive rate of different AMQ data structures for a given load factor as the bits per element varies. In specific, we configure all AMQ data structures to have the same space consumption and capacity (2^{27}). We insert the same number of elements into each AMQ data structure to ensure that the load factor is the same for all of them. Then, we vary the fingerprint length for gathered approach-based AMQ data structures and the number of hash functions for scattered approach-based AMQ data structures to change the bits per element. Then we measure the false positive rate for different bits per element. Figures 4 and 5 illustrate the false positive rates of different AMQ data structures as the bits per element vary for load factors of 25%, 50%, 75%, and 90%. As shown in Figures 4 and 5, the variable-length encoding framework can achieve an average reduction of 44.98%, 45.53%, 24.94%, 49.41%, 49.73% in the false positive rates of CF, TCF, CBF, ICBF, and TCBF, respectively. In addition, the variable-length encoding framework can reduce the false positive rates by 88.15%, 68.10%, 11.05%, and 4.38% for load factors of 25%, 50%, 75%, and 90%, respectively.

Analysis. When the load factor of an AMQ data structure is low, there is a significant amount of unused space that can be leveraged by the variable-length encoding framework to store longer encoding information for each element. Therefore, when the load factor is low, the variable-length encoding framework can significantly reduce the false positive rate of existing AMQ data structures. As the load factor increases, the amount of unused space in the AMQ data structure decreases, and the additional encoding information stored for each element by the variable-length encoding framework becomes shorter. Consequently, the false positive rate of the AMQ data structure using the variable-length encoding framework gradually approaches that of the original algorithm. Taking cuckoo filters as an example, when the fingerprint length of the standard cuckoo filters is 12 bits and the load factor is 25%, the average length of each fingerprint in the cuckoo filters using the variable-length encoding framework is 34.36 bits. Thus, for low load factors (such as 25% and 50%), the variable-length encoding framework can significantly reduce the false positive rate of AMQ data structures. When the load factor is 90%, there are still 24.5% of buckets in the cuckoo filter using a variable-length encoding framework where the number of stored elements does not reach the capacity of the bucket. These buckets store longer fingerprints than standard cuckoo filters, so for higher load factors, the variable-length encoding framework can still reduce the false positive rate of cuckoo filters.

C. Throughput

Our results show that the variable-length encoding framework reduces the false positive rate of AMQ data structures at the cost of negligible computational overhead. We measure the throughput of different AMQ data structures for insertion, positive lookup (the queried elements all exist in the AMQ data structure), negative lookup (the queried elements do not exist in the AMQ data structure), and deletion. Specifically, we measure the throughput of the AMQ data structures on the Synthetic, WIKI, and YCSB datasets. We insert 90% of the elements in the dataset into the AMQ data structures and measure the insertion throughput. Then, we query all inserted elements and measure the positive lookup throughput. Afterwards, we query the remaining 10% of elements to measure the negative lookup throughput. Finally, we delete all inserted elements and measure the deletion throughput. As shown in Table I, the variable-length encoding framework merely reduces the throughput of insertion, positive lookup, negative lookup, and deletion operations by 19.94%, 2.68%, 11.15%, and 12.89%, respectively. The reason why the variable-length encoding framework has such low

TABLE I
EVALUATION OF THROUGHPUT.

Datasets	Metrics	VCF	CF	VTCF	TCF	VCBF	CBF	VICBF	ICBF	VTCBF	TCBF
Synthetic	insertion throughput	6.282	6.777	6.178	6.594	2.942	4.939	2.883	4.532	1.841	2.384
	positive lookup throughput	18.830	30.881	18.641	17.883	5.584	5.735	5.232	5.262	2.535	2.532
	negative lookup throughput	19.271	29.488	19.194	21.841	8.169	8.462	10.612	10.465	10.077	10.290
	deletion throughput	11.320	14.972	10.072	14.724	4.354	4.952	4.511	4.482	2.554	2.562
WIKI	insertion throughput	19.819	20.947	7.671	8.033	3.563	4.562	3.363	4.380	1.945	2.505
	positive lookup throughput	22.951	30.514	18.661	19.239	6.639	5.416	6.052	5.061	3.013	3.003
	negative lookup throughput	13.171	29.032	17.660	22.696	10.182	8.333	11.968	11.264	11.676	13.823
	deletion throughput	11.761	27.465	10.682	17.037	5.220	4.679	5.265	4.348	3.047	3.079
YCSB	insertion throughput	13.251	14.458	10.570	11.858	3.808	5.617	3.580	5.370	1.962	2.573
	positive lookup throughput	23.129	35.497	21.971	21.579	7.093	6.685	6.561	6.197	2.942	2.830
	negative lookup throughput	20.473	33.656	20.456	25.322	10.349	9.949	12.690	12.534	11.900	11.816
	deletion throughput	12.819	20.019	11.807	17.972	5.566	5.732	5.644	5.277	2.982	2.884

overhead is that we use bit manipulation instructions to speed up the variable-length encoding and decoding processes. Compared with the variable-length encoding framework that does not use bit manipulation instructions, the throughput after using bit manipulation instructions is 1.17× that before optimization. Thus, the variable-length encoding framework reduces the false positive rate of AMQ data structures at the cost of negligible computational overhead.

VII. RELATED WORK

The existing AMQ data structures can be divided into two categories: gathered approach-based AMQ data structures and scattered approach-based AMQ data structures. We review studies closely related to our work in this section.

Scattered approach-based AMQ data structures. Scattered approach-based AMQ data structures mainly include counting Bloom filters [5], and their variants [11], [14], [41], [42]. For this type of AMQ data structures, existing studies mainly focus on improving the space efficiency to reduce the false positive rate. Take VCBF [41] as an example, when performing insertion, different from CBF [5] which increases each counter by one, VCBF increases each counter by a higher variable value (*e.g.*, 2 or 3). For lookup operations in VCBF, if the counter value is less than the variable value added to each counter during insertion, it means that the element does not exist. With this approach, VCBF reduces the false positive rate by fully utilizing the space of each counter. Another example is TCBF [42]. TCBF is an extension of VCBF. If the space of adjacent counters is not fully utilized, TCBF will borrow the space of adjacent counters to further reduce the false positive rate.

Gathered approach-based AMQ data structures. Gathered approach-based AMQ data structures mainly include cuckoo filters, quotient filters, and their variants [40], [46]. For example, CBCF [46] uses a flag bit for each bucket to indicate the type of bucket (*i.e.*, a bucket with three slots or a bucket with four slots). When the number of elements in a bucket is three, the elements can be stored in a bucket with a smaller capacity. For TCF [40], the number of buckets can be a value that is not a power of 2. This design reduces the number of redundant buckets. Therefore, TCF can reduce memory consumption without increasing the false positive rate.

The variable-length encoding framework improves the space efficiency by adjusting the encoding length based on the number of elements stored in the AMQ data structure, thereby achieving a lower false positive rate. Thus, the variable-length encoding framework is orthogonal to the above work and can further reduce the false positive rate of the above work.

VIII. CONCLUSION

In this paper, we propose a general framework named variable-length encoding framework to improve the space efficiency and reduce the false positive rate of AMQ data structures. The variable-length encoding framework can adjust the encoding length of elements based on the number of inserted elements in AMQ data structures. This approach can make full use of the memory space allocated to AMQ data structures to reduce the false positive rate. Then, we divide the existing AMQ data structures into two categories and illustrate how these two types of AMQ data structures use the variable-length encoding framework to reduce the false positive rate. In addition, we provide theoretical guarantees for AMQ data structures using the variable-length encoding framework. Furthermore, We evaluate the performance of AMQ data structures using the proposed variable-length encoding framework. Theoretical analysis and evaluation results show that AMQ data structures using the variable-length encoding framework have significantly lower false positive rates than state-of-the-art AMQ data structures. For future work, we will explore how to further improve the throughput of the variable-length encoding framework by leveraging SIMD techniques.

ACKNOWLEDGMENT

This work was supported in part by the National Natural Science Foundation of China under Grant No. 62272223, U22A2031, 61872178, 61832005, 62172206, 62072230, 62325205, 62072228, and 92267104, in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing University, and in part by the Jiangsu High-level Innovation and Entrepreneurship (Shuangchuang) Program.

REFERENCES

[1] 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.
[2] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of ACM International Conference on Emerging Networking Experiments and Technologies*. ACM, 2014, pp. 75–88.
[3] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok, "Don't

thrash: How to cache your hash on flash," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1627–1637, 2012.
[4] T. M. Graf and D. Lemire, "Xor filters: Faster and smaller than bloom and cuckoo filters," *Journal of Experimental Algorithmics*, vol. 25, no. 1, pp. 1–16, 2020.
[5] L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000.
[6] Y. Peng, J. Guo, F. Li, W. Qian, and A. Zhou, "Persistent Bloom filter: Membership testing for the entire history," in *Proceedings of International Conference on Management of Data*. ACM, 2018, pp. 1037–1052.
[7] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, "A general-purpose counting filter: Making every bit count," in *Proceedings of International Conference on Management of Data*. ACM, 2017, pp. 775–787.
[8] F. Zhang, H. Chen, H. Jin, and P. Reviriego, "The logarithmic dynamic cuckoo filter," in *Proceedings of IEEE International Conference on Data Engineering*. IEEE, 2021, pp. 948–959.
[9] P. Chen, D. Chen, L. Zheng, J. Li, and T. Yang, "Out of many we are one: Measuring item batch with clock-sketch," in *Proceedings of International Conference on Management of Data*. ACM, 2021, pp. 261–273.
[10] J. Liu, H. Dai, R. Xia, M. Li, R. B. Basat, R. Li, and G. Chen, "DUET: A generic framework for finding special quadratic elements in data streams," in *Proceedings of International World Wide Web Conference*. ACM, 2022, pp. 2989–2997.
[11] H. Dai, J. Yu, M. Li, W. Wang, A. X. Liu, J. Ma, L. Qi, and G. Chen, "Bloom filter with noisy coding framework for multi-set membership testing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 7, pp. 6710–6724, 2023.
[12] H. Wang, H. Dai, M. Li, J. Yu, R. Gu, J. Zheng, and G. Chen, "Bamboo filters: Make resizing smooth," in *Proceedings of IEEE International Conference on Data Engineering*. IEEE, 2022, pp. 979–991.
[13] H. McCoy, S. A. Hofmeyr, K. A. Yelick, and P. Pandey, "High-performance filters for GPUs," in *Proceedings of Annual Symposium on Principles and Practice of Parallel Programming*. ACM, 2023, pp. 160–173.
[14] Y. Li, Z. Wang, R. Yang, Y. Zhao, R. Zhou, and K. Zheng, "Learned bloom filter for multi-key membership testing," in *Proceedings of Database Systems for Advanced Applications*. Springer, 2023, pp. 62–79.
[15] X. Wu, H. Huang, Y. Du, Y. Sun, and S. Chen, "Coupon filter: A universal and lightweight filter framework for more accurate data stream processing," *Comput. Networks*, vol. 228, no. 1, pp. 1–13, 2023.
[16] R. Gu, S. Li, H. Dai, H. Wang, Y. Luo, B. Fan, R. B. Basat, K. Wang, Z. Song, S. Chen, B. Wang, Y. Huang, and G. Chen, "Adaptive online cache capacity optimization via lightweight working set size estimation at scale," in *Proceedings of Annual Technical Conference*. USENIX, 2023, pp. 467–484.
[17] Z. Zeng, R. Xiao, X. Lin, T. Luo, and J. Lin, "Double locality sensitive hashing bloom filter for high-dimensional streaming anomaly detection," *Information Processing and Management*, vol. 60, no. 3, pp. 1–18, 2023.
[18] S. Garg, A. Singh, G. S. Aujla, S. Kaur, S. Batra, and N. Kumar, "A probabilistic data structures-based anomaly detection scheme for software-defined internet of vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 6, pp. 1–10, 2021.
[19] A. Singh and S. Batra, "Fingerprint based duplicate detection in streamed data," *Computing and Informatics*, vol. 37, no. 6, pp. 1313–1338, 2018.
[20] S. Che, W. Yang, and W. Wang, "Improved streaming quotient filter: A duplicate detection approach for data streams," *The International Arab Journal of Information Technology*, vol. 17, no. 5, pp. 769–777, 2020.
[21] D. Karapiperis, A. Gkoulalas-Divanis, and V. S. Verykios, "Linkage of spatio-temporal data and trajectories," in *Proceedings of International Smart Cities Conference*. IEEE, 2019, pp. 766–771.
[22] A. Singh and S. Batra, "Ensemble based spam detection in social Iot using probabilistic data structures," *Future Generation Computer Systems*, vol. 81, no. 1, pp. 359–371, 2018.
[23] H. Dai, L. Meng, H. Wang, R. Gu, S. Chen, F. Chen, and W. Hu, "Distantly supervised entity linking with selection consistency constraint," in *Proceedings of Database Systems for Advanced Applications*. Springer, 2023, pp. 784–799.
[24] S. Xiong, Y. Yao, M. W. Berry, H. Qi, and Q. Cao, "Frequent traffic flow identification through probabilistic bloom filter and its GPU-based acceleration," *Journal of Network and Computer Applications*, vol. 87, no. 1, pp. 60–72, 2017.
[25] H. Dai, M. Li, and A. X. Liu, "Finding persistent items in distributed datasets," in *Proceedings of IEEE International Conference on Computer Communications*. IEEE, 2018, pp. 1403–1411.
[26] R. Xie, M. Li, Z. Miao, R. Gu, H. Huang, H. Dai, and G. Chen, "Hash adaptive bloom filter," in *Proceedings of IEEE International Conference on Data Engineering*. IEEE, 2021, pp. 636–647.
[27] H. Dai, M. Shahzad, A. X. Liu, and Y. Zhong, "Finding persistent items in data streams," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 289–300, 2016.
[28] A. Singh, S. Garg, S. Batra, N. Kumar, and J. J. P. C. Rodrigues, "Bloom filter based optimization scheme for massive data handling in IoT environment," *Future Generation Computer Systems*, vol. 82, no. 1, pp. 440–449, 2018.
[29] S. Yu, X. Li, H. Wang, X. Zhang, and S. Chen, "C_CART: An instance confidence-based decision tree algorithm for classification," *Intelligent Data Analysis*, vol. 25, no. 4, pp. 929–948, 2021.
[30] A. Singh, S. Garg, R. Kaur, S. Batra, N. Kumar, and A. Y. Zomaya, "Probabilistic data structures for big data analytics: a comprehensive review," *Knowledge-Based Systems*, vol. 188, no. 1, pp. 1–21, 2020.
[31] S. Yu, X. Li, H. Wang, X. Zhang, and S. Chen, "BIDI: A classification algorithm with instance difficulty invariance," *Expert Systems with Applications*, vol. 165, no. 1, pp. 1–13, 2021.
[32] M. Kumar and A. Singh, "Probabilistic data structures in smart city: Survey, applications, challenges, and research directions," *Journal of Ambient Intelligence and Smart Environments*, vol. 14, no. 4, pp. 229–284, 2022.
[33] H. Cha, X. Hao, T. Wang, H. Zhang, A. Akella, and X. Yu, "Blink-hash: An adaptive hybrid index for in-memory time-series databases," *Proceedings of the VLDB Endowment*, vol. 16, no. 6, pp. 1235–1248, 2023.
[34] C. Wang, X. Huang, J. Qiao, T. Jiang, L. Rui, J. Zhang, R. Kang, J. Feinauer, K. Mcgrail, P. Wang, D. Luo, J. Yuan, J. Wang, and J. Sun, "Apache IoTDB: Time-series database for internet of things," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2901–2904, 2020.
[35] J. Chen, Y. Ding, Y. Liu, F. Li, L. Zhang, M. Zhang, K. Wei, L. Cao, D. Zou, Y. Liu, L. Zhang, R. Shi, W. Ding, K. Wu, S. Luo, J. Sun, and Y. Liang, "Bytehtap: Bytedance's HTAP system with high data freshness and strong data consistency," *Proceedings of the VLDB Endowment*, vol. 15, no. 12, pp. 3411–3424, 2022.
[36] S. Rao, A. K. Verma, and T. Bhatia, "A review on social spam detection: Challenges, open issues, and future directions," *Expert Systems with Applications*, vol. 186, no. 1, pp. 1–31, 2021.
[37] A. Breslow and N. Jayasena, "Morton filters: Faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity," *Proceedings of the VLDB Endowment*, vol. 11, no. 9, pp. 1041–1055, 2018.
[38] M. Wang, M. Zhou, S. Shi, and C. Qian, "Vacuum filters: More space-efficient and faster replacement for bloom and cuckoo filters," *Proceedings of the VLDB Endowment*, vol. 13, no. 2, pp. 197–210, 2019.
[39] B. Koppelmann, P. Adelt, W. Mueller, and C. Scheytt, "RISC-V extensions for bit manipulation instructions," in *Proceedings of International Symposium on Power and Timing Modeling, Optimization and Simulation*. IEEE, 2019, pp. 41–48.
[40] K. Huang and T. Yang, "Tagged cuckoo filters," in *Proceedings of International Conference on Computer Communications and Networks*. IEEE, 2021, pp. 1–10.
[41] O. Rottenstreich, Y. Kanizo, and I. Keslassy, "The variable-increment counting bloom filter," *IEEE/ACM Transactions on Networking*, vol. 22, no. 4, pp. 1092–1105, 2014.
[42] P. Reviriego and O. Rottenstreich, "The tandem counting bloom filter - it takes two counters to tango," *IEEE/ACM Transactions on Networking*, vol. 27, no. 6, pp. 2252–2265, 2019.
[43] Bob Jenkins' hash function web page. [Online]. Available: http://burtleburtle.net/bob/hash/doobs.html
[44] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska, "Benchmarking learned indexes," *Proceedings of the VLDB Endowment*, vol. 14, no. 1, pp. 1–13, 2020.
[45] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of ACM Symposium on Cloud Computing*. ACM, 2010, pp. 143–154.
[46] P. Reviriego, J. Martínez, D. Larrabeiti, and S. Pontarelli, "Cuckoo filters and bloom filters: Comparison and application to packet classification," *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2690–2701, 2020.