Towards In-Network Compact Representation: Mergeable Counting Bloom Filter Vis Cuckoo Scheduling

WENJING LIU\textsuperscript{1}, ZHIWEI XU\textsuperscript{1,2}, (Member, IEEE), JIE TIAN\textsuperscript{3}, AND YUJUN ZHANG\textsuperscript{2}, (Member, IEEE)

\textsuperscript{1}College of Data Science and Application, Inner Mongolia University of Technology, Hohhot 010080, China
\textsuperscript{2}Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China
\textsuperscript{3}Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102, USA

Corresponding author: Zhiwei Xu (xuzhiwei2001@ict.ac.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1800403 and Grant 2016YFE0121500, in part by the National Science Foundation of China under Grant 61902382, Grant 61972381, Grant 61672500, Grant 61962045, Grant 61502255, and Grant 61650205; in part by the Strategic Priority Research Program of Chinese Academy of Sciences under Grant XDC02030500; in part by the Natural Science Foundation of Inner Mongolia Autonomous Region under Grant 2017MS(LH)0601 and Grant 2018MS06003; in part by the Science and Technology Planning Project of Inner Mongolia Autonomous Region under Grant 2019GG372; in part by the Key Technologies RD Program of Inner Mongolia Autonomous Region under Grant 2020GG0094, in part by the Science Research Project of Inner Mongolia University of Technology under Grant BS201934, and in part by the Visiting Scholar Project of China Scholarship Council under Grant 201908150030.

ABSTRACT With the breakthrough of edge intelligence, we are witnessing a booming increase in distributed applications on edge nodes. These distributed applications need to apply a novel data representation algorithm to support data-information exchanging and data-information based decision among different edge nodes. As the most efficient data compact representation algorithm, Counting Bloom Filter (CBF) is an extension of Bloom filter, which enables updating data representation as well as inserting data into a representation. To facilitate distributed applications on edge nodes, edge nodes need to exchange and summarize the information of the data collected from different edge nodes. Impossible to merge with other CBFs, the existing CBF and its variants thus cannot be used for representing and exchanging data information among edge nodes. To handle this problem, we design a novel mergeable CBF, mergeCBF. Based on an insight about the counting processing of a CBF, we unfold the counter array of the conventional CBF to a group of bit arrays, and in order to support merging multiple filters, map each inputted item to the cells in this group of cuckoo-scheduled bit arrays instead of the counters in CBF. Experiments on real-world datasets demonstrate that mergeCBF can support conventional operations and merging operations in an efficient way without degrading the quality of the representation results.

INDEX TERMS Edge computing, in-network data representation, compact representation, mergeable counting bloom filter, cuckoo-based bit array scheduling.

I. INTRODUCTION

According to a survey of Cisco, there will be 850 ZB of data generated by mobile users and IoT devices by 2021 [1]. Edge computing is fastest and easiest way to analyze vast amounts of geographically distributed data. Data generated by IoT devices, other forms of data benefit from edge computing architectures to address low data transmission latency, overhead and privacy concerns. Edge computing [2], [3] is proposed on a set of connected systems and devices for data collection, caching, processing, and analysis at edge close to data source. To achieve efficient and lightweight data pre-processing and analyzing at the resource-limited edge nodes [4]–[6], a compact data representation [7], [8] for exchanging the information of data among edge nodes is highly desired.

Bloom filter [9] is a space-efficient structure for compact data representation, and the most popular data summarization method. Among variants of Bloom filter, Counting bloom filter (CBF) is a group of variants [12]–[14], [16]–[18] that support deleting operations as well as updating operations.
TABLE 1. Comparison of compact representation algorithms.

|                | Query | Insert | Delete (Update) | Merge | Required Resource |
|----------------|-------|--------|-----------------|-------|-------------------|
| BF & others without counting [9]–[11] | ✓     | ✓      | ✓               | ✓     | RAM               |
| CBF & its variants [12]–[17]          | ✓     | ✓      | ✓               | ✓     | RAM, SSD          |
| QF & its variants [18], [19]         | ✓     | ✓      | ✓               | ✓     | RAM               |
| Our mergeCBF                            | ✓     | ✓      | ✓               | ✓     | RAM               |

on the representation results. To accurately summarize data, the stale data should be removed from the data representation results periodically. Therefore, CBF has been the primary tool used in Internet to achieve efficient data representation.

With the breakthrough of edge computing, we are witnessing a booming increase in distributed in-network applications. These applications leverage distributed edge nodes instead of data center servers or terminal devices to handle massive data. To facilitate these applications to analyze data in a distributing way, novel general data representation schemes are highly desired for identifying and summarizing data on different edge nodes, which poses a new challenge on whether the data representation results can be efficiently merged with each other. Fulfilling the challenges is difficult for almost all of the existing compact data representation approaches, including CBF and its variants. Among other types of data compact representation algorithms, only quotient filter [18] and its variant [19] can support the merging operation, which however facilitates the hash table-based data representation on solid-state drive (SSD) better than only a general BF-like hyper-compact representation on high-speed random access memory (RAM). The detailed comparison of the most popular compact representation algorithms is present in Table 1.

To provide a mergeable counting bloom filter for common high-speed random access memory (RAM), we study the counting processing of a CBF, and obtain an insight that the reason why CBF cannot support merging operations is that CBF does not record any other information of the inputted data items but its hit count on some specific counters in CBF, and thus CBF cannot identify the repeated data items inserted in the different candidate filters to be merged. According to this insight, we propose a novel design by unfolding the counter array of CBF to a group of bit arrays, and cuckoo schedule these bit arrays in a fixed sequence to be used. In this way, we can find the replicated items that have been inserted into different candidates, and only keep one of them into the merging result. The operations on basic bit arrays are simple enough to make our design more efficient that the other existing approaches.

**Contributions:**

1) We study the counting processing of a CBF, and address the reason why CBF cannot support merging operations. The reason is that CBF does not record any other information of the inputted data items but its hit count on some specific counters in CBF. Thus, CBF cannot identify the repeated data items inserted in the different candidate filters to be merged, and these replicated items cause the overflow of the counters of CBF.

2) A novel design of mergeable counting bloom filter, mergeCBF, is proposed by unfolding the counter array of CBF to a group of bit arrays, and cuckoo schedule these bit arrays in a fixed sequence to be used. Ultimately, the information of the inserted items are recorded in different bit arrays, and thus the information of the replicated items will be aggregated into the existing record, which enables the merging operations on multiple filters.

3) We implement extensive experiences based on real-world data. The experimental results demonstrates our mergeCBF can achieve efficient merging operations among multiple filters as well as accurate query and updating operations.

We provide the preliminary of compact representation and approximate membership query, and define the problem in Section II. In Section III, a mergeable counting bloom filter is proposed according to an insight about the counting process of CBF. Based on the insight, we take multiple bit arrays to record the repeated counting times when items are inserted. The performance of our design is evaluated in Section V. We study the related work and conclude our work in Section VI and VII.

II. PRELIMINARY AND PROBLEM STATEMENT

A. COMPACT REPRESENTATION AND APPROXIMATE MEMBERSHIP QUERY

The huge amount of existing data in Internet makes data representation and exact matching data queries costly. The brute-force searching to respond the exact-matching query can cause extremely high cost on computational and storage resources. In contrast, compact data representation and approximate membership queries can relax constraints on user requests to take much shorter time for users to get satisfactory results. In fact, the data representation and membership queries are not a new problem but become crucial to existing and possible future in-network data analysis at edge. Considering the limitation of resources at edge nodes, the data representation and membership query should be lightweight and efficient enough to avoid overwhelming edge nodes.

As an extension of the most popular compact representation, Bloom filter (BF) [9], Counting Bloom filter (CBF) [12] consists of an array of counters and multiple hash functions. It extend each one-bit cell in Bloom filter to a multiple-bit counter to support deleting data as well as inserting and query data in the representation result. Expect for that, when the expected number of the inserted data items equals to \( n \), CBF is completely equivalent to BF and can be configured...
according to the configuration method of the basic Bloom filter, including the number of cells \((m)\), the number of hash functions \((k)\), etc [9].

\[
  k = \ln 2 \cdot \frac{n}{m} \quad \text{s.t.} \quad \varepsilon \leq (1 - e^{-\frac{k(n+0.5)}{m-1}})^k \quad (1)
\]

In this way, CBF can achieve a compact representation of the inserted data items without involving a large false positive rate \((\varepsilon)\).

The detailed inserting and deleting operations of a CBF are depicted in Figure 1. To insert a data item into a CBF, this item is hashed by function \(h_1\) to \(h_k\) to \(k\) different counters, and these counters will increase by 1 to record this item. When we delete a data item from a CBF, this item is also hashed by function \(h_1\) to \(h_k\) to \(k\) different counters, and these counters will decrease by 1 to remove the information of this item.

There is an upper bounded that the maximum hit times of a counter in CBF is greater than a specific integer [12], and each counter and the size of each cell of CBF (bit string length 1) can be configured according to this upper bound. A 4-bit counter can meet the common needs of a CBF, and with a high probability, it is guaranteed that the counter of any cell will not exceed its maximum counting range. Note that the upper bound is obtained under an assumption that the repeated data items will not be inserted into a CBF.

**B. PROBLEM STATEMENT AND ANALYSIS**

To support in-network data information exchanging and data-information based decision among different edge computing nodes, the promised data compact representation structures should be feasible to be merged with each other as well as being updated. CBF satisfies the requirement of edge nodes for updating their data representation results when these nodes rely on CBF to exchange the information of in-network data. However, a CBF cannot be merged with other CBFs. CBF need to extend functions for merging multiple CBFs to support in-network data information exchanging. It can be concluded that we need to design a novel CBF according to the following guidelines:

1) Support inserting, deleting and query based on CBF.
2) Extend CBF to facilitate merging multiple filters.

The way for the CBF with a counter array to support the merging operation on multiple filters is to add up the value of the counters in the same position of different CBFs. This straightforward way does not work due to the problem illustrated in Figure 2. When a data item is inserted in a CBF, the CBF will increase counters to which the inserted item is mapped, but no other information of this item is recorded. Therefore, there is no means to identify whether an item has been inserted into both CBF1 and CBF2. On an assumption that no replicated item will be inserted in a CBF, the common size of the counters are 4 bits [12], the maximum value of the counters are 15. When CBF1 an CBF2 are merged, \(\forall i\), and the value of the \(i\)–th counter in CBF1 will be add up to that of the \(i\)–th counter in CBF2. The replicated data items inserted in both CBF1 and CBF2 cause that a part of adding results will exceed the maximum value of the counters, thus leading to errors. Therefore, to achieve a mergeable counting bloom filter, we need to handle the problem how to distinguish the records of the same data items inserted into different filters. We should eliminate the repeated records of a data item in...
III. IN-NETWORK COMPACT REPRESENTATION

Towards efficient and lightweight data pre-processing and analyzing at the resource-limited edge nodes, we propose an in-network compact representation scheme for a general purpose, mergeable Counting Bloom Filter (mergeCBF). In detail, we first study the counting process of the conventional CBF and obtain an insight how to eliminate replicated items inserted in different filters. Ultimately, we find a way to design a mergeable counting bloom filters.

A. DESIGN OVERVIEW

As discussed in Section II-B, a conventional CBF cannot merge with other CBFs to obtain a merging result due to the lack of indispensable information to distinguish and eliminate the repeated records of the same data items inserted into different filters. Ultimately, some counters in the merging result will overflow the counting range.

On the other hand, we notice that a basic Bloom filter (BF) can merge to another by performing bitwise OR on the bit arrays of these two Bloom filters (see Figure 3). The information of the inserted items is represented with the “1” bits in the bit array of a BF. After the merging operation is carried out on multiple BFs, the “1” bits of each BF are aggregated into the result without any interference from the other aggregated BFs. Inspired by this, we try to unfold the counter array of CBF to a group of bit arrays, and schedule these bit arrays in a fixed sequence to record the inserted items. Specifically, when a hashing result of an item is obtained, we select an array, and then set the cell corresponding to the hash result to “1”. We need to repeat this procedure $k$ times by using $k$ hash functions to reduce the false positive ratio of membership query operations (see the configuration of BF [9]). After $k$ cells are set according to $k$ hash results, all “1” bits in different bit arrays of our filter are used as the record of this inserted item. The used bit array should be different in terms of distinct positions where the hash values point to, guaranteeing the record of the inserted items free to collision with a high probability.

To achieve that, we use the bit arrays by following the idea of Cuckoo hash [20]. More specifically, we propose a virtual-Cuckoo schedule algorithm for scheduling bit array usage in the group. Meanwhile we include an additional bit array to record the bitwise OR result of the bit array group of the proposed filter. Ultimately, we can speed up the query operation on our filter at a cost of little longer inserting, deleting and merging latency.

The structure of the proposed mergeable counting bloom filter (mergeCBF) is illustrated in Figure 4:

1) Hash Function Set (i.e., $Hash_1$ to $Hash_k$): A set of hash functions used to map each inserted item into certain array cells at different positions. The number of the hash functions (i.e., $k$) is set up according to the optimized configuration of BF [9], where $k$ is set up according to the size of the bit array (i.e., $m$) and the expected scale of inserted data items (i.e., $n$), $k = \ln 2 \cdot \frac{n}{m}$.

2) Bit Array Group of Size $g$ (i.e., $barr_1$ to $barr_g$): A group of bit arrays is involved in to replace the counter array used in CBF. To record each hash result of an inserted item, we set one of the cells at the position where the hash result indicate to be “1”. In this way, we update the counting result corresponding to the indicated position. Therefore, $g$ is set up according to the maximum value of the counting result, which can be confirmed according to the analysis in [12]. The cell numbers of $barr_1$, $barr_2$, ..., and $barr_g$ are all equal to $m$.

3) Bitwise OR Result of the Array Group (i.e., $orBarr$): A bitwise OR result on the bit array group, used to improve the query efficency of the proposed mergeable bloom filter. That is quite important for edge computing applications to look up specific data among edge nodes and make decisions according to the distribution of these data.

Ultimately, the proposed mergeCBF does not only support the inserting, deleting and querying of data items, but also...
FIGURE 4. Structure of mergeable counting bloom filter.

Algorithm 1 Virtual-Cuckoo

**Input:** $i$;

//The initial index of a cell to store a new item;
{$c_1, \ldots, c_g$};
//The group of $g$ candidate cells;

**Output:** $\hat{i}$;

//The index of the next available cell among $g$ candidate cells;

1: if the $i$-th cell, $c_i$, is occupied then
2: $\hat{i} =$ CuckooHash($i$);
3: $\hat{i} =$ virtual-Cuckoo($\hat{i}$);
4: else
5: $\hat{i} = i$;
6: end if
7: return $\hat{i}$;

The above recursion can converge to guarantee all inserting operations succeed, according to the Cuckoo graph theory of Cuckoo hash. Cuckoo graph is an undirected graph that consists of vertexes for the indices of different hash table cells, and edges with two endpoints indicating two candidate cells for an inserting procedure. When Cuckoo hash insert items into the hash table, all inserting operations can succeed if and only if the corresponding cuckoo graph is a graph with at most one cycle in each of its connected components [20]. Cuckoo hash used in Algorithm 1 can generate random hash results only with a trivial collision probability, that means different inputs are mapped to distinct results. Therefore, each vertex in the Cuckoo graph of Algorithm 1 connect another one by one, and the last vertex will connect to the first vertex finally. The Cuckoo graph of Algorithm 1 is circle. All insert operations of Algorithm 1 will succeed, and Algorithm 1 finally comes to convergence.

As depicted in Figure 4, we apply Algorithm 1 to schedule bit array usage during inserting and deleting items in our filter, when the hash result of an item indicates a specific position of the bit arrays. The inserting procedures are listed as the follows.

1) Hash a data item with hash functions, $Hash_1, \ldots, Hash_k$, to get $k$ hash results. Each hash result, $p_j$, indicates a specific position of the bit arrays.

2) Regarding each hash result $p_j$, we use Algorithm 1 to find the first array $barr_i$, the $p_j$-th cell of which contains bit “0”.

3) Set the $p_j$-th cell of $barr_i$ to be “1”. Repeat this procedure until $k$ cells in the positions indicated by the $k$ has results are set to be “1”.

Notice that inserting operation should be performed after a query operation, in order to avoid inserting a data item repeatedly. The detailed inserting process is described in Algorithm 2.

C. DELETING

The deleting operation is similar to the inserting operation. An function like virtual-Cuckoo, virtual-Cuckoo-, is applied to return the last used candidate cell. To be simplicity,
Algorithm 2 mergeCBF.Add

**Input:** An new item, item;

1. barrSet = mergeCBF.GetAllBarr();
   //Get all of the bit arrays;
2. for j = 0 to k - 1 do
3.     p_j = Hash_j(item);
4.     cellSet = mergeCBF.GetAllCell(barrSet, p_j);
   //Get all of the p_j-th cells of the bit arrays;
5.     index = virtual-Cuckoo(1, cellSet);
6.     Barr[index][p_j] = 1;
7. end for
8. Update orBarr()
   //Do bitwise OR on all bit arrays to update orBarr;

we don’t provide the detail of the deleting operation here, but the key steps as the follows:

1) Query the item.
2) If the item is exist, we find k cells by the function virtual-Cuckoo-, and set these cells to be “0”.
3) Finally, update orBarr.

**D. QUERY**

We involve bit array orBarr in our filter to achieve more efficient membership query. The query operation is implemented on orBarr through checking whether the k cells indicated by Hash_1, Hash_2, ..., Hash_k are equal to “1”. The detailed procedure is listed in Algorithm 3.

**E. MERGING**

As discussed in Section II-B, we need to make our mergeCBF feasible to merge with each other to support in-network data information summarizing and exchanging. Compared to the only one CBF that support the merging operation on multiple filters, QF and its variants, our design is more efficient to support the merging procedure by performing bitwise OR on bit arrays (see Figure 5). To avoid the merging result exceeds the optimized capacity of mergeCBF (i.e., n), we need to estimate the total number of the items inserted in the filters that will merge with each other. According the balls and bins theory [22], a bit cell is set to be “1” with a probability under a Poisson distribution P(n). Hence, suppose η cells contains “1” bits, x is the times that a cell has been set to “1”, the estimation value of the number of items inserted in a filter, ˆn, is derived as the following:

\[
P(x > 0) = 1 - P(x = 0) = 1 - e^{-n}
\]

(2)

To select η cells from m cells, there are \((m)_\eta\) combinations.

\[
\hat{n} = \frac{1}{(m)_\eta}(1 - e^{-n})^\eta
\]

(3)

The merging procedure is listed as the following steps.

1) Evaluate whether the total number of items inserted in these filters exceeded the capacity of mergeCBF. If so, stop the merging operation and report an error, otherwise, continue the next step.
2) For ∀j, 1 ≤ j ≤ g, mergeCBF performs bitwise OR on the j – th bit arrays of different filters to obtain a merging result of the j – th bit arrays.
3) Perform bitwise on all orBarr of different filters to obtain a merging result of orBarr. The detailed is present in Algorithm 4.
Algorithm 4 mergeCBF.Combine

Input: mergeCBF1, mergeCBF2;
Output: mergeCBF
1: if mergeCBF1.EstSize() + mergeCBF2.EstSize() ≥ n then
2:    return error;
3: end if
4: barrSet1 = mergeCBF1.GetAllBarr();
5: barrSet2 = mergeCBF2.GetAllBarr();
6: for j = 0 to g do
7:    barr1 = barrSet1.Next();
8:    barr2 = barrSet2.Next();
9:    barr = BitwiseOr(barr1, barr2);
10:   mergeCBF.NextBarr() = barr;
11: end for
12: return mergeCBF, orBarr
   = BitwiseOr(mergeCBF1.orBarr, mergeCBF2.orBarr);

IV. ANALYSIS

In this section, we provide some theoretical analysis about mergeCBF.

Theorem 1: The probability that the number of cells at the same position of bit arrays are set to be “1” is greater than or equal to g has an upper bound m

\[ \Pr(\max(c_j) \geq g) \leq m \left( \frac{nk}{g} \right) \left( \frac{1}{m} \right)^g \]  

Proof: Suppose the maximum capacity of mergeCBF can insert is n, the size of bit arrays is m, the number of hash functions is k, and the number of bit array is g. The probability that the number of cells at the same position of bit arrays have been set to be “1” is greater than or equal to g is:

According to Stirling formulation, the following can be obtained:

\[ \Pr(\max(c_j) \geq g) \leq m \left( \frac{nk}{g} \right) \left( \frac{1}{m} \right)^g \]  

According to m = \[ \frac{nk}{g} \] discussed in [9], it can be found that in the optimal configuration of bloom filter, n \times k has a linear relationship with m. Therefore, n \times k is much larger than g, and the value in the square root of Formula 5 is less than 1. Meanwhile, the second term is enlarged to obtain a relaxed upper bound of Formula 4:

\[ \Pr(\max(c_j) \geq g) \leq m \left( \frac{nk}{g} \right) \left( \frac{1}{m} \right)^g \]  

which is an upper bound of the probability that the number of cells at the same position of bit arrays are set to be “1” is greater than or equal to g.

According to formula (6), when g is 16, the probability of all CCBF bits being set to “1” is 1.37 \times 10^{-15} \times m. In general, this case will come up in an edge computing application.

Theorem 2: The inserting sequence of items has no impact on their compact representation result in the bit arrays of mergeCBF.

Proof: without loss of generality, among a group of data items, item1, . . . , itemi, . . . , itemn, inverted into mergeCBF in a specific sequence, two arbitrary items, itema and itemb, are map to the same position of mergeCBF (i.e., the j − th cells). Suppose α cells at the position j has been set to be “1”. If we insert itema in mergeCBF before itemb, we use virtual-Cuckoo to find an array, arrayda, and set the j − th cell of arrayda to be “1”. Similarly, when we insert itemb into mergeCBF, we use virtual-Cuckoo to find the next available array, arraydb, and set the j − th cell of arraydb to be “1”. Alternatively, we first insert itema in mergeCBF and then perform the inserting operation on itemb. The j − th cell of arraydb and that of arrayda are used in the same sequence, and they are both set to be “1” finally. Therefore, The inserting sequence of different items has no impact on their compact representation result in the bit arrays of mergeCBF.

V. EXPERIMENTAL EVALUATION

In this section, we conduct experimental evaluations to validate the effectiveness and efficiency of the proposed mergeable Counting Bloom filter (mergeCBF). We first explore the false positive evaluations at mergeCBF and three baseline schemes for comparison. We then demonstrate the high-performance of the proposed mergeCBF under different considerations.

A. IMPLEMENTATION AND EXPERIMENT SETUP

All the experiments were run on a Intel(R) Core(TM) i5-5200u CPU @2.20GHz, dual-core and four-threaded with 8 GB of RAM. To evaluate the performance of the proposed mergeCBF, we implement mergeCBF and three of the most popular counting filters, CB, dlCBF and Counting Quotient Filter(CQF). The hash function used in the filters is MurmurHash3 with a hash result of 32 bits, which is an efficient hash function with a low collision rate. Other parameters are listed in Table 2. The detailed implementation of these filters is list as the follows:

1) mergeCBF: We use 16 bit arrays in mergeCBF to replace the counter of 4 bits used in CBF. According to the preset false positive ratio (0.05) and the relations among the number of hash functions(k),
the size of the bit arrays \((m)\), and the estimated capacity\((n_i)\), the \(k\) and \(m\) are set to be 3 and \(m = (capacity \times |log(erreurrate)|)/(k(log(2))^2)\) respectively. A simple one_at_a_time function [21] with a small hashing range \(g\) is used as the Cuckoo hash function.

2) CBF [12]: To make the counting range of each counter in CBF also be 16, the size of each counter is set to be 4 bits. The preset false positive ratio, the number of hash functions\(k\) and the size of the counter arrays \(m\) are set in accordance to those configured in mergeCBF.

3) diCBF [14]: diCBF divides the storage to \(d\) partitions, and correspondingly, \(d\) hash functions are used to map items into the buckets in these partitions. To make the estimated counting range be 6, the number of the buckets in each partition is set up as \(capacity/(4 \times 6)\), and the size of each bucket is set to be 8 to avoid the inserted items turning out the preset counting range.

4) CQF [19]: CQF use a hash function to generate hash values for different data items. The first \(q\) bits of a hash value are used to indicate the original position of the hashed item (i.e., Quotient value). Ultimately, the capacity of CQF is equal to \(2^q\). The last \(6\) bits are stored in cells as the remainder of this item to eliminate the query false positive. Two bits of each cell is used as Occupieds and Runends, respectively.

A popular real-world open dataset, gas sensor array temperature modulation dataset, is used to evaluate the performance of mergeCBF. The data items in this dataset are collected from a chemical detection platform composed of 14 temperature-modulated metal oxide (MOX) gas sensors to monitor the mixtures of carbon monoxide and humid synthetic air in a gas chamber during 3 weeks. This dataset provides 4,095,000 data items of the sensors and the measured values of CO concentration, humidity and temperature inside the gas chamber. We sample from this dataset to build a serial of experimental datasets with different scales \((10^4, 10^5\) and \(10^6)\).

| TABLE 2. Filter configuration. |
|-------------------------------|
| Scheme           | Data size | \(10^4\) | \(10^5\) | \(10^6\) |
|------------------|-----------|---------|---------|---------|
| mergeCBF         | num_hash(k)  | 3       | 3       | 3       |
|                  | capacity(n) | 10^4    | 10^5    | 10^6    |
|                  | bit_array_size(m) | \(6.4 \times 10^4\) | \(6.4 \times 10^5\) | \(6.4 \times 10^6\) |
|                  | num_barre(g)  | 16      | 16      | 16      |
| CBF              | num_hash(k)  | 3       | 3       | 3       |
|                  | capacity(n)  | 10^4    | 10^5    | 10^6    |
|                  | counter      | \(6.4 \times 10^4\) | \(6.4 \times 10^5\) | \(6.4 \times 10^6\) |
|                  | array_size(m) | 16      | 16      | 16      |
| diCBF            | num_hash(k)  | 4       | 4       | 4       |
|                  | capacity(n)  | 10^4    | 10^5    | 10^6    |
|                  | bit_size(m)  | \(6.4 \times 10^4\) | \(6.4 \times 10^5\) | \(6.4 \times 10^6\) |
|                  | num_childtab(d) | 4       | 4       | 4       |
|                  | num_cell(w)  | 8       | 8       | 8       |
| CQF              | num_hash(k)  | 1       | 1       | 1       |
|                  | capacity(n)  | \(2^2\) | \(2^4\) | \(2^4\) |
|                  | quotient(g)  | 24      | 24      | 24      |
|                  | reminder(r)  | 6       | 6       | 6       |

B. FALSE POSITIVE EVALUATION

Just like other bloom filters, MergeCBF achieves compact data representation at a cost of few false positives in the query process, that means the queries on the items that have not been inserted in mergeCBF will return positive results. Considering the other type of errors, false negatives, will not come up during membership queries on bloom filter and its variants, we only evaluate the false positives for mergeCBF to query data items, as well as those of the other three baselines.

As depicted in Figure 6, we configure the filters for different capacity scales, \(10^4\), \(10^5\) and \(10^6\), and then insert \(10^4\), \(10^5\) and \(10^6\) items into these filters, respectively. Correspondingly, we query \(10^4\), \(10^5\), and \(10^6\) nonexistent items on the these filters. The false positive evaluation results have linear increase in the number while we query data items on larger scales on different filters. Among the query results corresponding to the query scale of \(10^6\), 886 false positives come up when we perform query operations on mergeCBF, those for CBF, diCBF and CQF are 5037, 1379 and 691. The proposed mergeCBF can accurately represent a huge number of inserted data in a compact way.

To achieve a comprehensive evaluation about false positives, we present the trend of the false positive increment while we insert more data items in a mergeCBF with a configured capacity \(10^4\), as well as in two others with the configured capacity \(10^5\) and \(10^6\) (see Figure 7). \(R_{fill}\) is a ratio to indicate how much data has been inserted into the filters. \(R_{fill} = N_i/n\), where \(N_i\) is the number of the items that have been inserted in this mergeCBF, and \(n\) is the configured capacity of this mergeCBF. When \(R_{fill}\) is less than 0.85, the number of the false positives remains 0. While \(R_{fill}\) continues to increase from 0.85, an exponential increase comes up, until \(R_{fill} = 1\). Similar to other bloom filters, mergeCBF can provide the perfect data compact representation in the cases before the inserted data items have not filled the filter up. We need to enlarge the capacity of the filter, and make the configuration of the filter capacity more robust for all cases.

C. EFFICIENCY EVALUATION

To evaluate the efficiency of mergeCBF, We study the latency for inserting, deleting and query items on mergeCBF and
the three other baselines. Since among all of the existing approaches, only QF and its variant, CQF, can support merging operations, we compare the latency for merging multiple mergeCBFs with that for CQF.

1) QUERY LATENCY
As shown in Figure 8, the proposed mergeCBF always takes the least time to complete all query tasks on whatever data sets we use. To query $10^4$ data items, all filters achieve similar performance. For a larger data set with a scale of $10^5$, mergeCBF takes 0.09 Second less than that of CQF, 0.16 Second less than that of dlCBF, and 0.44 Second less than that of CBF. mergeCBF outperforms the three baselines much more when we use the largest data set. The involved orBarr improves the query efficiency of mergeCBF, and thus mergeCBF can achieve the minimum in-memory query latency like basic Bloom filter. CQF leverages both memory storage resource and SSD storage resource to store more data information into a hash table, and achieve similar query performance.

2) INSERTING AND DELETING LATENCY
As depicted in Figure 9 and Figure 10, mergeCBF takes at most 5 times of time latency to complete inserting operations and deleting operations compared with those of CBF, dlCBF and CQF. For instance, mergeCBF takes 1.42 second, CBF takes 0.76 second, dlCBF takes 0.45 second, CQF takes 0.37 second to complete $10^5$ inserting operations. For each inserting operation, mergeCBF will use more 10.5 $\mu s$, CBF takes 0.3s, QF takes 0.44s and dlCBF takes 0.43s to complete $10^5$ inserting operations. For each deleting operation, mergeCBF will use more 10.6 $\mu s$.

We include $g$ bit arrays into mergeCBF to replace the counter array used in CBF. Consequently, although we extend CBF to support merging operations, the updating process on these bit arrays causes longer latency. The inserting and deleting operations are called only when there is a new data or an existing data expires. For each data item, these two operations only be called once. Controversially, the query and merging operation are call frequently by edge nodes to achieve efficient in-network data information collecting and analysis. Considering the above facts, we include an additional bit array in mergeCBF to aggregate all the $g$ bit arrays and speed up query operations at a cost of involving more computational overhead on the inserting and deleting operations.
3) MERGING LATENCY
The merging operation is the significantly featured function of mergeCBF, which is one of the indispensable functions for an in-network compact representation. Figure 11 depicts the time cost for CQF and mergeCBF to merge multiple filters, where the x-axis represents the number of filters. One group of mergeCBFs and CQFs inserted $10^5$ data items. Another group of filters have inserted 50000 data items sampled with replacement. We can find for whatever data items are inserted the filters and how many of the filters we perform the merging operations on, mergeCBF achieve better merging performance compared with CQF. The minimum difference between the merging latency of mergeCBF and that of QCF is 2.91 second.

4) IMPACT OF COUNTER SIZE
As depicted in Section III-A, mergeCBF uses g bit arrays to replace the counter array used in the conventional CBF, where g is set in accordance to the estimated maximum counting range of the counters used in CBF. Although this maximum counting range is usually set to be 16, a larger counter can enhance the scalability of CBF and its variants, as well as a smaller one will reduce the storage overhead. Correspondingly, the filters may have a change of the latency regarding different operations. To evaluate the impact of counting range on the performance of mergeCBF, we set the counting range g to be 8, 16, and 32, and the capacity to be $10^5$. We query, insert and delete $10^5$ data items on different filters. As shown in Figure 12, different counting ranges have no impact on the latency of query operations, and only have little impact on the latency of inserting and deleting operations. Although the latency for the virtual-Cuckoo scheduling on g bit arrays is quite small, the scheduling on more bit arrays will cost a little bit more time.

VI. RELATED WORK
Internet has massively increased the amount of data available for novel intelligent applications. To achieve efficient represent for massive data in Internet, Bloom filter (BF) [9] has been applied as the most widely used as a compact data representation mechanism, and many different types of Bloom filters have been derived to meet diverse requirements of novel data-intensive applications. Among them, Counting Bloom filter (CBF) [12] is designed to support both inserting operation and deleting operation. CBF expands the single-bit cell of BF into a fixed-length multi-bit cell for counting, and increment each cell hits by hashing operations on inputted items. The hit times of the cell is recorded in the counter to support deleting the corresponding items, by reducing the counter value.

To optimize the performance of CBF by reducing the storage overhead and the false positive ratio, a variety of new counting Bloom filters have been proposed. Among them, the most popular approaches are present as the following. Cohen et al. proposed Spectral Bloom Filter (SBF) [13], it can ensure that the counter of any cell will never exceed its maximum counting range by extending the counter array of the CBF, but its storage overhead is larger compared with CBF. dlCBF [14] is proposed based on d-left hash, and the storage cost is optimized under the same false positive rate. MLCCBF data structure [15] improves standard CBF in terms of fast accessing and limited memory consumption, but can only be implemented in the small and fast local memory or “on-chip SRAM”. Rottenstreich et al. proposed a general method [16] based on variable increments, called Variable Increment Counting Bloom Filter (VI-CBF), can always achieve a lower false positive rate and a lower overflow probability bound than CBF in practical systems. VI-CBF is a popular data structure for the representation of dynamic sets, achieving a good trade off between memory efficiency and query accuracy. Tandem Counting Bloom Filter (T-CBF) [17] is a new data structure that relies on the interaction among counters to describe sets with higher accuracy compared with VI-CBF. Quotient Filter [18] is an another type of data representation structure that achieves efficient insert, delete and membership query on a hash table on SSD. As a variant of Quotient Filter,
the counting quotient filter (CQF) [19], also base on memory and SSD, is proposed to support approximate membership testing and data item counting, meanwhile supporting to the merge operation on multiple filters.

VII. CONCLUSION

In this paper, we propose a novel Mergeable Counting Bloom filter (mergeCBF). mergeCBF can efficiently insert, delete and query data items, and its computational complexity is at the same scale of the conventional counting Bloom filter. The performance of the query operation, the most frequent operation during edge computing, is enhanced compared with that of other counting filters. In addition, mergeCBF supports efficient multi-filter merging, which lays the foundation for a variety of scenarios that require an in-network compact representation mechanism. It would be extensively studied in further work to adapt more specific in-network representation scenarios, e.g., distributed deep learning for augmented reality and intelligent transportation.

REFERENCES

[1] Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update 2017–2022, GMDD, Cisco, San Jose, CA, USA, 2017.
[2] P. G. Lopez, A. Montresor, D. Epeña, A. Datta, T. Higashino, A. Ianniti, M. Barcellos, P. Felber, and E. Riviere, “Edge-centric computing: Vision and challenges,” ACM SIGCOMM Comput. Commun. Rev., vol. 45, no. 5, pp. 37–42, Oct. 2015.
[3] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge computing: Vision and challenges,” IEEE Internet Things J., vol. 3, no. 5, pp. 637–646, Oct. 2016.
[4] K. Zhang, Y. Mao, S. Leng, Y. He, and Y. Zhang, “Mobile-edge computing for vehicular networks: A promising network paradigm with predictive offloading,” IEEE Veh. Technol. Mag., vol. 12, no. 2, pp. 36–44, Jun. 2017.
[5] Z. Xiong, Y. Zhang, D. Niyato, P. Wang, and Z. Han, “When mobile blockchain meets edge computing,” IEEE Commun. Mag., vol. 56, no. 8, pp. 33–39, Aug. 2018.
[6] S. T. Boshroooyeh, A. Küpçü, and Ö. Özkasap, “Privado: Privacy-preserving group-based advertising using multiple independent social network providers,” ACM Trans. Privacy Secur., vol. 23, no. 3, pp. 1–36, Jul. 2020.
[7] J. Xie, D. Guo, X. Shi, H. Cai, C. Qian, and H. Chen, “A fast hybrid data sharing framework for hierarchical mobile edge computing,” in Proc. IEEE INFOCOM Conf. Comput. Commun., Jul. 2020, pp. 2609–2619.
[8] A. Broder and M. Mitzenmacher, “Network applications of Bloom filters: A survey,” Internet Math., vol. 1, no. 4, pp. 485–509, Jan. 2004.
[9] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” Commun. ACM, vol. 13, no. 7, pp. 422–426, Jul. 1970.
[10] Y. Matsumoto, H. Hazeyama, and Y. Kadobayashi, “Adaptive Bloom filter: A space-efficient counting algorithm for unpredictable network traffic,” IEICE Trans. Inf. Syst., vol. E91-D, no. 5, pp. 1292–1299, May 2008.
[11] F. Deng and D. Rafiei, “Approximately detecting duplicates for streaming data using stable Bloom filters,” in Proc. ACM SIGMOD Int. Conf. Man. Data (SIGMOD), 2006, pp. 25–36.
[12] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, “Summary cache: A scalable wide-area Web cache sharing protocol,” IEEE/ACM Trans. Netw., vol. 8, no. 3, pp. 281–293, Jun. 2000.
[13] S. Cohen and Y Matias, “Spectral Bloom filters,” in Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD), 2003, pp. 241–252.
[14] F. Bonomi, M. Mitzenmacher, R. Panigrahi, S. Singh, and G. Varghese, “An improved construction for counting Bloom filters,” in Proc. Conf. Eur. Symp., 2006, pp. 684–695.
[15] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci, “MultiLayer compressed counting Bloom filters,” in Proc. IEEE INFOCOM 27th Conf. Comput. Commun., Apr. 2008, pp. 311–315.
[16] O. Rottenstreich, Y. Kanizo, and I. Keslassy, “The variable-increment counting Bloom filter,” IEEE/ACM Trans. Netw., vol. 22, no. 4, pp. 1092–1105, Aug. 2013.
[17] P. Reviriego and O. Rottenstreich, “The tandem counting Bloom filter—It takes two counters to tango,” IEEE/ACM Trans. Netw., vol. 27, no. 6, pp. 2252–2265, Dec. 2019.
[18] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. Kusznmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok, “Don’t thrash: How to cache your hash on flash,” Proc. VLDB Endowment, vol. 5, no. 11, pp. 1627–1637, 2012.
[19] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, “A general-purpose counting filter: Making every bit count,” in Proc. ACM Int. Conf. Manage. Data, May 2017, pp. 775–787.
[20] R. Pagh and F. F. Rodler, “Cuckoo hashing,” J. Algorithms, vol. 51, no. 2, pp. 122–144, May 2004.
[21] B. Jenkins, Algorithm Alley: Hash Functions. [Online]. Available: https://www.drdobbs.com/database/algorithm-alley/184410284
[22] M. Mitzenmacher and E. Upfal, Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis. Cambridge, U.K.: Cambridge Univ. Press, Jul. 2017, pp. 99–101.

WENJING LIU received the B.S. degree from the Inner Mongolia University of Technology, Hohhot, China, in 2011, the M.S. degree from the Beijing University of Posts and Telecommunications, Beijing, China, in 2018. She is currently an Assistant Professor with the Inner Mongolia University of Technology. Her research interests include automated software testing and performance analysis.

ZHIWEI XU (Member, IEEE) received the B.S. degree from the University of Electronic Science and Technology of China, Chengdu, China, in 2002, and the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2018. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, State University of New York at Stony Brook, Stony Brook, NY, USA. He is an Associate Professor and a M.S. Supervisor with the Inner Mongolia University of Technology, while working as an Adjunct Professor with the Institute of Computing, Chinese Academy of Sciences. His research interests include network performance analysis and the related mathematics problems.

JIE TIAN received B.S. degree in computer science from the Department of Computer Science, Tianjin University, China, in 2005, the M.S. degree in computer science from the Department of Computer Science, Nankai University, China, in 2008, and the Ph.D. degree in computer science from the Department of Computer Science, New Jersey Institute of Technology, in 2015. He is currently working as an Adjunct Professor with the New Jersey Institute of Technology. His main research interests include wireless sensor networks and cloud computing.

YUJUN ZHANG (Member, IEEE) received the B.S. and Ph.D. degrees from Nankai University, Tianjin, China, in 1999 and 1999, respectively, and the second Ph.D. degree from ICT, CAS, in 2004. He is currently a Professor and a Ph.D. Supervisor. His main research interests include future Internet and Internet security assessment and verification.