CECF: The Concurrent Entry-extensible Cuckoo Filter

Shuiying Yu[1], Changqi Feng[1], Sijie Wu[2], Cheng Hu[1], Yun Huang[1]*

[1] Faculty of Computer and Information Technology, Wuhan Institute of Shipbuilding Technology, China
[2] Huawei Technologies Co., Ltd, Nanjing, China
yushuiying2006@126.com, 64430452@qq.com, wusijie6@huawei.com, 382362818@qq.com, 477063793@qq.com

Abstract

The emergence of huge and dynamic datasets in practical applications poses significant challenges for approximate set representation structures. The cardinality of these sets fluctuates, there's a growing demand for adaptable and flexible representation structures. The existing dynamic representation structures offer filter-level and entry-level extensions Cuckoo Filter. The entry-level extension Cuckoo Filter, E2CF, achieves this by leveraging adjacent buckets with contiguous physical addresses within the Cuckoo Filter to extend bucket entries, avoids many discrete memory accesses in a query. However, for big data processing, a set representation structure may be read and inserted frequently, and query operations may be performed at the same time as the insertion. Since the E2CF insert may have a relocation process, reading this item will fail. Therefore, this paper proposes a multi-core concurrent E2CF (CECF). CECF utilizes path lock table to store the path of the inserting item. Experimental results demonstrate that, when compared to the exising designs, CECF significantly reduces query and insertion times by 59% and 55%, respectively.

Keywords: Cuckoo filter, Entry-extensible, Dynamic set representation, Concurrent set membership query

1 Introduction

Since the advent of large-scale datasets in big data applications [1], the utilization of set representation and membership query structures has become widespread [2-3]. Set representation entails arranging set information according to a predefined format, while membership queries involve determining whether a specific item is part of a set. The efficiency of set membership queries holds paramount importance in numerous practical applications. For example , in the realm of network security monitoring [4-5], prolonged query times can lead to delayed detections and compromised protection. Similarly, in web cache proxies [6], the long query time increase response time, directly affecting user experience. Furthermore, the performance of set membership queries also has implications for non-repeatable insertion and deletion operations, which need to search the item first.

In real world big data applications, achieving accurate set storage and query often falls short of meeting the rigorous demands for both time and space efficiency. Fortunately, approximate set representation and membership query structures present a practical solution. These structures can substantially decrease storage cost and expedite query speeds, albeit with a slight risk of false positives in the query results. Consequently, they have garnered considerable attention in academic research [7-8].

The most commonly utilized structures for approximate set representation are the Bloom Filter (BF) [7], Cuckoo Filter (CF) [8], and their variants [9-11]. The BF [7] is an array of fixed-length bits, initially set to "0". To insert an item into the BF, one utilizes k independent hash functions to map the item to specific indices within the array and subsequently sets the corresponding bits to "1". Queries to determine set membership check if all k bits are set to "1". However, BF has a notable limitation: it does not support item deletion. Flipping bits to "0" to remove an item can cause false negatives for other items sharing those bits.

To delete item, the Counting Bloom Filter (CBF) [10] extends traditional BF by substituting each bit with a d bits counter. Nevertheless, this process demands $d\times$ more space compared to a BF. CF [8] utilizes an array comprising many buckets, wherein each bucket has b entries to store fingerprint. For each new item, CF utilizes two hash functions to identify two potential buckets and stores the item's fingerprint in one entry of the candidate buckets. If the candidate buckets are full, CF first kicks out an existing item and store the new item in it. Then CF puts the kicked out item in its another candidate bucket. Since CF stores the item's fingerprint, it can support deletion by querying and deleting the fingerprint in the candidate buckets.

The sizes of sets in practical applications constantly change unpredictably [12-14]. For instance, the number of the new items from stream processing applications exhibits significant variability [15]. The traditional static structures [7-8, 10], since their inability to adjust their capacity, are unable to capture the dynamic changes in the set cardinality. Furthermore, the memory cost involved in reconstructing a bigger static structure is often prohibitive for many applications. Limited research has addressed the challenges posed by frequently changing sets, with notable exceptions include Entry-Extensible Cuckoo Filter (E2CF) [9], Dynamic Cuckoo Filter (DCF) [12], and Dynamic Bloom Filter (DBF) [13]. To facilitate dynamic extensions and reductions in capacity, DBF [13] employs a method of appending and merging multiple homogeneous

*Corresponding Author: Changqi Feng; Email: 64430452@qq.com
DOI: https://doi.org/10.70003/160792642026072704005

CBFs. Nevertheless, CBFs do not support dependable deletion operations. DCF [13] enhances reliability by utilizing multiple CFs to store item fingerprints, thereby supporting dependable delete operations. E2CF [9] reduces membership query time, and utilizes entry-level extension to guarantee that entries with the same indexes are stored at contiguous physical addresses. Through the asynchronous extension and fine-grained splitting, E2CF achieves optimal space and time efficiency, significantly outperforming query and insert performance compared to DBF and DCF.

However, for big data processing, a set representation structure may be read and inserted frequently, and query operations may be performed at the same time as the insertion [18-20]. Since the E2CF insert may have a relocation process, reading this item will fail. Therefore, this paper proposes a multi-core concurrent E2CF, CECF. CECF utilizes a lock table to store the path of the item insertion, reducing the query and insertion time. We implement CECF and conduct comprehensive experiments utilizing real-world data traces to evaluate our design. The results demonstrate that compared to existing designs, CECF reduces the query time and insertion time by 59% and 55%.

We organize the remainder of the paper as follows. Section 2 illustrates an overview of the related work. Section 3 delves CECF design. Section 4 examines and analyzes the performance of CECF. Section 5 evaluates our design. Section 6 concludes the paper.

2 Related Work

We delve into the related work about set representation structures. Based on their capacity to dynamically adjust their size, we categorize the approximate set representation structures into two main groups: static and dynamic approximate set representation structures

2.1 Static Structures

Bloom filter and counting bloom filter. Bloom Filter (BF) [7] consists of an array of n bits, all initialized to "0". However, BF lacks the capability to support deletion. To address this limitation, Fan et al. [10] introduced Counting Bloom Filter (CBF), replacing each bit in the traditional BF with a counter comprising d bits. This innovation allows for precise adjustments in counter values when inserting or deleting items. However, CBF necessitates $d\times$ more space compared to a standard BF.

Cuckoo filter. Cuckoo Filter (CF) [8] comprises an array comprising m buckets, with each bucket containing c entries. When item x arrives, CF employs a hash function to generate its fingerprint (denoted as ξ_x), and inserts ξ_x into one entry. In order to minimize the collisions during the process of insertion, CF calculate the indexes of two candidate buckets for x and stores its fingerprint ξ_x within an entry of the two candidate buckets. The indexes of these two candidate buckets, $h_1(x)$ and $h_2(x)$, are calculated by Equation (1).

$$\begin{cases} h_1(x) = hash(x) \\ h_2(x) = h_1(x) \oplus hash(\xi_x) \end{cases} \quad (1)$$

The two hash indexes for item x are subtlety and relevant designed. Utilizing the indexes of two candidate bucket, along with the fingerprint ξ_x stored within, CF is capable of determining the other index of x through an XOR operation involving the known index and $hash(\xi_x)$. As a result, when a collision occurs, with both candidate buckets for the new item already occupied, CF employs a strategic eviction process. It selects and removes one of the existing fingerprints from the buckets, making room for the new fingerprint. Subsequently, the victim fingerprint is relocated to its alternative candidate bucket. Figure 1 illustrates the specific process of inserting x and query y by CF.

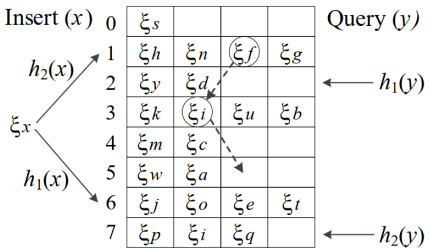

Figure 1. The specific process of CF insertion and query

Counting quotient filter. Pandey et al. [17] introduce the Counting Quotient Filter (CQF), supporting approximate set membership query, counting the item arrivals, resizing, and merging. Nevertheless, CQF resizing mechanism involves reconstructing a new filter with doubled memory, and subsequently transferring all hashes from the old filter to the new table individually. This approach of CQF is unsuitable for dynamic set representation due to the requirement of triple the space occupied by the original filter and the significant time investment during the resizing process.

Guo et al. [13] expose the highly dynamic nature of real-world datasets. However, existing static structures [8, 10] fall short in their ability to extend capacity dynamically. Moreover, prematurely allocating excessive capacity leads to unnecessary space overhead. Consequently, it is imperative to propose a set representation structure suitable for dynamic capacity expansion.

2.2 Dynamic Structures

Dynamic bloom filter. Guo et al. [13] introduce an innovative structure, Dynamic Bloom Filter (DBF), is designed to approximate set representations while effectively handling dynamically changing set cardinalities. Composed of a linked list of s CBFs, the DBF expands its capacity by adding new empty CBFs as the active CBF fills up. For queries, the DBF independently checks each CBF to deicde the existence of an item. However, as the number of items and CBFs increases, the false positive rate (FPR) spikes significantly, rendering the DBF unsuitable for handling large-scale dynamic sets.

Dynamic cuckoo filter. Chen et al. [12] introduce the Dynamic Cuckoo Filter (DCF), an innovative structure capable of dynamically adjusting its capacity to represent dynamic approximate set. A DCF utilizes a linked list of s homogeneous CFs to dynamically, enabling it to extend and reduce capacity. Given that CF inherently supports deletion operations, DCF ensures reliable deletion functionality.

A DCF consistently upholds an active CF. Upon the arrival of a item, the structure endeavors to insert the fingerprint of item to the current active CF. Nonetheless, once the active CF reaches its maximum capacity, the DCF adds an empty CF structure, and sets this CF to the active CF. During the execution of a query, the DCF must scan through all CFs. If the fingerprint being searched matches any CF, the DCF promptly returns a positive result and halts the search. If not, the DCF proceeds to examine all potential buckets and ultimately returns a negative result. Evidently, the query in DCF multiple discrete memory accesses, leading to query time long.

Consistent cuckoo filter. Luo et al. [14] introduce a set representation structure with the bucket-level extension, Index-Independent Cuckoo Filter (I2CF). This filter adaptively add and remove buckets in response to dynamic datasets. However, the scalability of the filter is limited to small-scale capacity expansion, which unfortunately compromises its query performance. To address the challenges posed by the highly dynamic datasets, Luo et al. [14] develop the Consistent Cuckoo Filter, a structure consisting of a linked list of I2CFs. Nevertheless, CCF inherits the long memory access time associated with query operations due to its adoption of the same filter-level extension methods as DBF and DCF.

Entry-Extensible Cuckoo Filter. Yu et al. [9] propose a dynamic structure featuring the entry-level extension, Entry-Extensible Cuckoo Filter (E2CF), E2CF minimizes memory access overhead and accelerate membership queries for dynamic set representations. It accomplishes this by leveraging adjacent buckets with contiguous physical location within the Cuckoo Filter to expand bucket entries, thereby minimizing the number of discrete memory accesses required during a query. Figure 2 illustrates the example of E2CF.

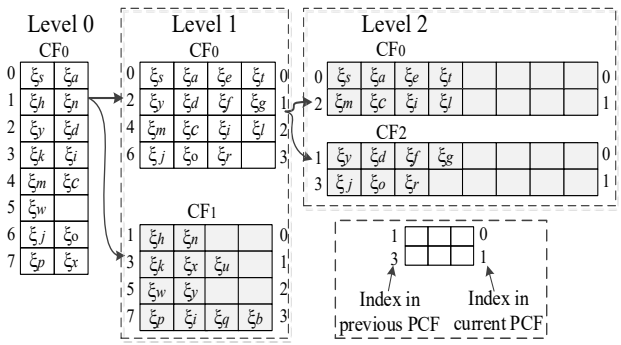

Figure 2. An example of E2CF

None of them takes into account how to query item during the item insertion process, and how to read the results in parallel with multiple cores. Our design, which uses lock table to record paths, solves these problems.

3 Concurrent Entry-extensible Cuckoo Filter

3.1 Overview

In order to ensure data consistency, E2CF has only one write operation performed at a time, which is equivalent to locking the entire structure for any operation. For large-scale datasets, insertion operations lock the entire structure will affect the performance of operations. If E2CF implements multi-threaded parallel insertion and query operations, that is, concurrent E2CF (CECF), it will greatly improve the performance of the operation.

Since CECF replaces the fingerprint of the item during insertion, CECF first records the fingerprint to be replaced by the item to be inserted and its corresponding position. For this purpose, we design a path lock table, adopting cuckoo hash, which can record paths quickly and achieve fine-grained lock at key level, as shown in Figure 3. The structure of this lock tuple is shown in the figure, which is essentially a hash table with five parts in each entry: the serial number of Partial Cuckoo Filter (PCF), corresponding bucket in the first PCF, the serial number of next PCF, corresponding bucket in the second PCF, the victim fingerprint (or inserting fingerprint). For example, tuple (0, 1, 1, 2, ξ_h), the serial number of PCF is 0, the serial number of bucket in the PCF_0 is 1, the serial number of PCF is 1, the serial number of bucket in the PCF_1 is 2, the victim fringerprint is ξ_h. CECF relocates the fingerprint ξ_h from the first bucket in PCF_0 to the second bucket in PCF_1. CECF obtains the location by hashing the first pair of PCF numbers and bucket numbers, and the first location by hashing the second pair of PCF numbers and bucket numbers. When the second pair of PCF numbers and bucket numbers are '-1', the relocation ends.

To reduce the path length, CECF can adjust the Maximum Number of Kickouts (MNK) to simplify the query process. In the analysis and experiment of DCF [12], it is found that the insertion performance decreases rapidly when MNK is too large. When the MNK setting value is small, the space utilization can still be guaranteed.

3.2 Operations of CECF

Membership query. When querying for item x, CECF first computes k_1 and $h_1(x)'$ using Equation (2), then check the path lock table. If the fingerprint of the item is in the table, CECF does not need to query the PCF structure, and return a success response. Otherwise, if $h_1(x)'$ is in the lock table, wait a specified duration before querying the lock table. Otherwise, CECF queries the PCF. If a matching fingerprint ξ_x is found in the bucket of PCF_{k1} CECF returns a success response. If not, it computes k_2 and $h_2(x)'$ to check for the existence of ξ_x in PCF_{k2}. If neither of these two candidate buckets contains ξ_x, CECF returns a failure response. If not, it calculates k_2 and $h_2(x)'$ to check for the existence of ξ_x in PCF_{k2}. If neither of these two candidate buckets contains ξ_x, CECF returns a failure response. Since

CECF only reads two buckets, and the entries within each bucket occupy contiguous physical memory addresses, the overhead introduced by memory access during a query is minimal. Algorithm 1 outlines the detailed process for conducting a membership query in our CECF.

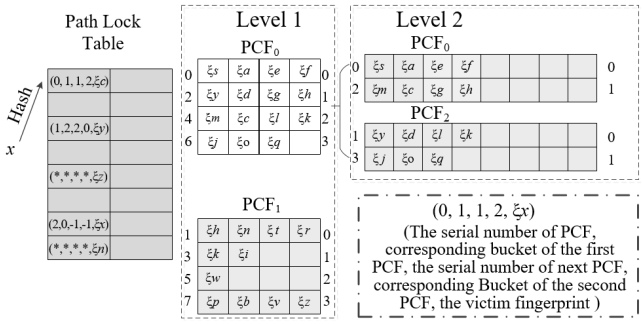

Figure 3. An example of CECF

Insertion. When inserting a item, CECF first obtains the insertion path. If there is a bucket in the lock table in the insert path, wait for a specified duration before inserting the bucket. If the relocation number exceeds the value of MNK, CECF needs to extend one PCF. Otherwise, the record path is in the path lock table. After obtaining the insertion path, record the path in the path lock table. After the insertion is complete, delete the path tuples.

Deletion. To delete an item x, CECF first query the path lock table. If the lock table contains the fingerprint ξ_x, wait for a specified duration before performing the operation. Then performs a membership query to locate the position of the fingerprint ξ_x. Upon successful location, CECF removes ξ_x. If the fingerprint ξ_x is not present in CECF, the deletion operation returns a failure response.

Algorithm 1. CECF: query (item x)

```
1  fp = fingerprint(item x);
2  h1(x) = hash(x);
3  h2(x) = h1(x) ⊕ hash(fp);
4  Compute k1 and h1(x)'.
5  Calculate the position of this path in the lock table
   based on k1 and h1(x)'.
6  if (Look for fp in the lock table == true)
7      return true;
8  else{
9      if (Look for the fp in PCF_k1.Bucket[h1(x)'] ==
   true)
10         return true;
11     else return false;
12 }
```

4 Optimization of CECF

4.1 Split Condition

CECF splits a PCF when the number of relocations reaches MNK in order to insert an item, however, the load of this PCF is not always the heaviest in the relocation process, this leads to lower space utilization. CECF can choose the PCF with the heaviest load among the last several PCFs to split. CECF records the last five relocated the serial numbers of candidate PCFs and the corresponding bucket indexes to a link list. If no empty entry is found, CECF will not kick out the victim's fingerprints. When the number of relocations reaches MNK, CECF finds the heaviest load PCF from the link list for splitting operations. The structure of one node in the link list is (the serial numbers of candidate PCF: the bucket index: victim's fingerprint). For example, the link list is (14 : 5 : ξ_l) → (7 : 21 : ξ_v) → (15 : 3 : ξ_m) → (10 : 60 : ξ_t) → (1 : 31 : ξ_a). CECF finds the heaviest load PCF is PCF_{15}, then kick out the victim's fingerprints in the order (14 : 5 : ξ_l) → (7 : 21 : ξ_v), and split PCF_{15} into two new PCFs and insert the victim to one of the two new PCFs.

4.2 Adjustable Overall Load

Dynamically changing the load factor threshold of CECF can adapt to insertion performance. Fan et al. [8] has already analyzed the insertion performance, CF has higher insertion performance when it is a lower load factor. When the number of inserted items reaches the capacity threshold ($\alpha \times m \times b$), CECF can split PCF to extend capacity, there is a time-memory trade-off between the insertion performance and load of CECF. According to the real application requirements, CECF can dynamicly change the overall load to adjust the insertion performance, which adjusts the load factor α. For example, if the insertion request unable to respond in time when a higher load factor, CECF can reduce the load by adjust α. In contrast, if the insertion request performance of the real application is less than the insertion performance of the CECF. For space, E2CF can make the load factor lower.

5 Performance

5.1 Experiment Setups

We implement CECF , we have chosen SHA1 to generate hash values. All the experiments were carried out on a dedicated server equipped with 16-core Intel 2.60GHz Xeon E5-2670 CPU and 64GB RAM. This CPU owns 32KB L1 data cache, 256KB L2 cache, and 20MB L3 cache. The servers' cache line size is 64 bytes. To thoroughly evaluate the performance of CECF, we utilize the synthetic dataset [21] and real-word network traffic traces [16] to evaluate its performance.

We compare query time, insertion time, and memory cost of CECF with E2CF [9, 13] and DCF [12]. To ensure fairness, given that both DBF and DCF operations are single-threaded, we conduct all experiments using single-threaded operations. We run non-repeatable insertion operation, prohibiting the insertion of duplicate items. To compare the insertion time comprehensively, under the different number of items increases, we assess the time of insertion operation comprehensively by measuring both instantaneous and cumulative time. For CECF, E2CF and DCF, we fix the parameters as $m = 2^{22}$, $b = 4$, $f = 16$, and vary α between 0.8 and 0.9. For DBF, we adjust the parameters to maintain the same false positive rate as CECF. We run each experiment five times, and record the

average values. DBF, DCF, and E2CF adopt single core, while CECF adopts 4-core by default [22-24].

5.2 Results

Figure 4 shows the query time with different s. We make negative queries, involving queries for items that do not exist. The results reveal that the splitting operation has minimal impact on query performance. Furthermore, CECF significantly reduces the query time of DBF, DCF, and E2CF by 92%, 90% and 46%, respectively.

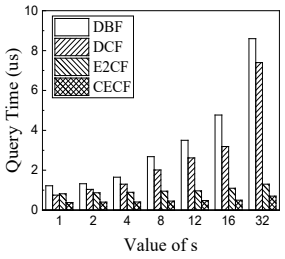

Figure 4. Query time with different values of s

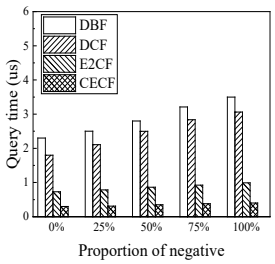

Figure 5. Query time with the queries of different negative

Figure 5 represents the query time with the queries of different negative, and sets s = 32. The experimental result indicates that CECF consistently outperforms DBF, DCF and E2CF by a significant margin. CECF reduces the query time of DBF, DCF and E2CF by 87%, 87% and 59%, respectively.

Figure 6 evaluates the query time under different set cardinalities. The results demonstrate that CECF reduces the query time of DBF, DCF and E2CF by 93%, 91% and 57%, respectively.

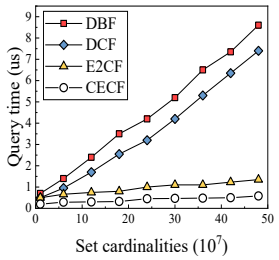

Figure 6. Query time under different set cardinalities

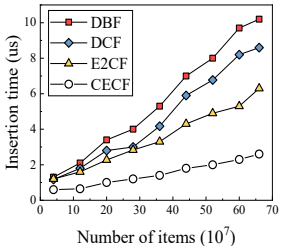

Figure 7. Instantaneous insertion time, MNK=10

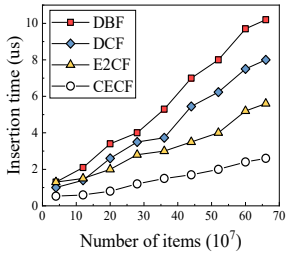

Figure 8. Instantaneous insertion time, MNK=40

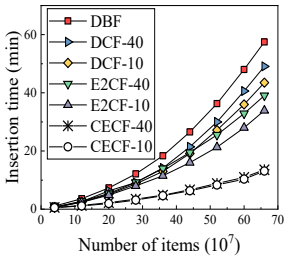

Figure 9. Cumulative insertion time, α = 0.9

Figure 7 and Figure 8 show the instantaneous insertion time when MNK = 10, MNK = 40, and α = 0.9. CECF reduces the instantaneous insertion time of DBF, DCF and E2CF by 74%, 68%, and 55%, respectively.

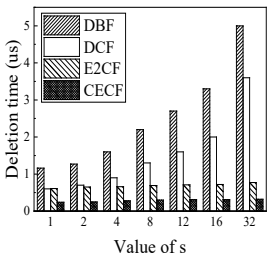

Figure 10. Delete time, α = 0.8, MNK=10

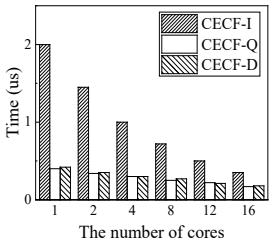

Figure 11. Insertion, query and deletion time under different cores

Figure 9 illustrates the cumulative insertion time under different MNKs when α = 0.9. As we elevate MNK from 10 to 40, the number of relocations increases, resulting in a rise in the insertion time for both DCF, E2CF and CECF. The result demonstrates that CECF reduces the cumulative insertion time of DBF, DCF and E2CF by 75%, 69% and 54%, respectively.

Figure 10 exhibits the deletion time of filters under different values of s. As s increases, the deletion time for DBF and DCF exhibits a linear growth trend, whereas the deletion time of CECF and E2CF is virtually unchanged. Figure 11 depicts the insert, query and deletion time under different number of cores, CECF-I, CECF-Q and CECF-D denote the insertion, query and deletion time, respectively. The result reveals that the insertion, query, and deletion time of CECF decrease as the number of cores increases.

6 Conclusion

In this paper, we found that E2CF could not query during insert operations and could not operate in parallel with multiple cores. We design and implement CECF, a multi-core concurrent E2CF, which supports fast membership query for large-scale dynamic dataset. CECF leverages path lock table to store the path of the inserting item, which can achieve multiple times the reading efficiency and realize that the item can be queried during insertion. The experimental results demonstrate that CECF greatly outperforms existing schemes.

Acknowledgements

This research is supported in part by the 2024 Special Project for High-level Talents No. 2024G05.

References

[1] H. Chen, H. Jin, S. Wu, Minimizing inter-server communications by exploiting self-similarity in online social networks, *IEEE Transactions on Parallel and*

Distributed Systems, Vol. 27, No. 4, pp. 1116–1130, April, 2016.

[2] R. Pagh, F. F. Rodler, Cuckoo hashing, *Journal of Algorithms*, Vol. 51, No. 2, pp. 122–144, May, 2004.

[3] N. Dayan, M. Athanassoulis, S. Idreos, Optimal bloom filters and adaptive merging for lsm-trees, *ACM Transactions on Database Systems*, Vol. 43, No. 4, pp. 16:1–16:48, December, 2018.

[4] B. Groza, P. Murvay, Efficient intrusion detection with bloom filtering in controller area networks, *IEEE Transactions on Information Forensics and Security*, Vol. 14, No. 4, pp. 1037–1051, April, 2019.

[5] J. Grashöfer, F. Jacob, H. Hartenstein, Towards application of cuckoo filters in network security monitoring, *Conference on Network and Service Management (CNSM)*, Rome, Italy, 2018, pp. 373–377.

[6] A. Cidon, A. Eisenman, M. Alizadeh, S. Katti, Cliffhanger: Scaling performance cliffs in web memory caches, *Symposium on Networked Systems Design and Implementation (NSDI)*, Santa Clara, CA, USA, 2016, pp. 379–392.

[7] B. H. Bloom, Space/time trade-offs in hash coding with allowable errors, *Communications of the ACM*, Vol. 13, No. 7, pp. 442–426, July, 1970.

[8] B. Fan, D. G. Andersen, M. Kaminsky, M. Mitzenmacher, Cuckoo filter: Practically better than bloom, *Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, Sydney, Australia, 2014, pp. 75–88.

[9] S. Yu, S. Wu, H. Chen, H. Jin, The entry-extensible cuckoo filter, *IFIP International Conference on Network and Parallel Computingc (NPC)*, Zhengzhou, Henan, China, 2020, pp. 373-385.

[10] L. Fan, P. Cao, J. M. Almeida, 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, June, 2000.

[11] M. Wang, M. Zhou, S. Shi, 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, October, 2019.

[12] H. Chen, L. Liao, H. Jin, J. Wu, The dynamic cuckoo filter, *IEEE International Conference on Network Protocols (ICNP)*, Toronto, ON, Canada, 2017, pp. 1–10.

[13] D. Guo, J. Wu, H. Chen, Y. Yuan, X. Luo, The dynamic bloom filters, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 22, No. 1, pp. 120–133, January, 2010.

[14] L. Luo, D. Guo, O. Rottenstreich, R. T. B. Ma, X. Luo, B. Ren, The consistent cuckoo filter, *IEEE Conference on Computer Communications (INFOCOM)*, Paris, France, 2019, pp. 712–720.

[15] B. D. Monte, S. Zeuch, T. Rabl, V. Markl, Rhino: Efficient management of very large distributed state for stream processing engines, *Proceedings of the International Conference on Management of Data (SIGMOD)*, online conference, 2020, pp. 2471–2486.

[16] WIDE project, 2020. http://mawi.wide.ad.jp/mawi/.

[17] P. Pandey, M. A. Bender, R. Johnson, R. Patro, A general-purpose counting filter: Making every bit count, *Proceedings of the International Conference on Management of Data (SIGMOD)*, Chicago, IL, USA, 2017, pp. 775–787.

[18] B. Fan, D. G. Andersen, M. Kaminsky, Memc3: Compact and concurrent memcache with dumber caching and smarter hashing, *Symposium on Networked Systems Design and Implementation (NSDI)*, Lombard, IL, USA, 2013, pp. 371–384.

[19] X. Li, D. G. Andersen, M. Kaminsky, M. J. Freedman, Algorithmic improvements for fast concurrent cuckoo hashing, *European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, 2014, pp. 27:1–27:14.

[20] Z. Chen, X. He, J. Sun, H. Chen, L. He, Concurrent hash tables on multicore machines: Comparison, evaluation and implications, *Future Generation Computer Systems*, Vol. 82, pp. 127–141, May, 2018.

[21] A. Clauset, C. R. Shalizi, M. E. J. Newman, Power-law distributions in empirical data, *SIAM Review*, Vol. 51, No. 4, pp. 661–703, October, 2009.

[22] C. Dai, S. Lu, C. Ma, S. Garg, M. Alrashoud, An Adaptive Rank-based Tensor Ring Completion Model for Intelligent Transportation Systems, *IEEE Transactions on Consumer Electronics*, Vol. 70, No. 1, pp. 2235–2243, February, 2024.

[23] S. Lu, P. Wang, W. Zhu, C. Dai, Y. Zhang, C. Liu, S. Dai, A Nonlocal Feature Self-Similarity Based Tensor Completion Method for Video Recovery, *Neurocomputing*, Vol. 580, Article No. 127513, May, 2024.

[24] C. Dai, S. Lu, C. Liu, B. Guo, A Light-weight Skeleton Human Action Recognition Model with Knowledge Distillation for Edge Intelligent Surveillance Applications, *Applied Soft Computing*, Vol. 151, Article No. 111166, January, 2024.

Biographies

Shuiying Yu received the PhD degree in computer systems organization from the Huazhong University of Science and Technology (HUST). She works in Faculty of Computer and Information Technology at Wuhan Institute of Shipbuilding Technology. Her interests include stream data processing and distributed algorithms.

Changqi Feng works in Faculty of Computer and Information Technology at Wuhan Institute of Shipbuilding Technology, associate professor. His main research interests include BigData, intelligent sensor.

Sijie Wu received the PhD degree in computer systems organization from the Huazhong University of Science and Technology (HUST), in 2022. He is currently a senior engineer in Huawei Technologies Co., Ltd. His research interests include BigData processing systems and large language models.

Cheng Hu works in Faculty of Computer and Information Technology at Wuhan Institute of Shipbuilding Technology, and mainly focuses on Big Data.

Yun Huang works in Faculty of Computer and Information Technology at Wuhan Institute of Shipbuilding Technology. Her research interests include digital media technology, virtual reality technology and big data processing.