The Entry-Extensible Cuckoo Filter

Shuiying Yu, Sijie Wu, Hanhua Chen(✉), and Hai Jin

National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
{shuiying,wsj,chen,hjin}@hust.edu.cn

Abstract. The emergence of large-scale dynamic sets in real applications brings severe challenges in approximate set representation structures. A dynamic set with changing cardinality requires an elastic capacity of the approximate set representation structure, while traditional static structures, e.g., bloom filter, cuckoo filter, and their variants cannot satisfy the requirements. Existing dynamic approximate set representation structures only provide filter-level extensions, which require a single membership query to probe all discrete filters one by one. The large number of small discrete memory accesses takes up the vast majority of query time and results in unsatisfied query performance. To address the problem, in this work we propose the *entry-extensible cuckoo filter* (E2CF) to reduce memory access overhead for dynamic set representation and accelerate the membership query. E2CF utilizes adjacent buckets with continuous physical addresses in a cuckoo filter to extend bucket entries, which avoids many discrete memory accesses in a query. To further make E2CF space and time efficient, we adopt asynchronous extension and fine-grained splitting methods. Experiment results show that compared to state-of-the-art designs, E2CF reduces the query and insertion time by 82% and 28%, respectively.

Keywords: Dynamic set representation · Set membership query · Entry-extensible · Cuckoo filter

1 Introduction

Since the emergence of large-scale sets in big data applications [4], set representation and membership query structures have been widely used [6]. Set representation means organizing set information based on a given format, while the membership query means determining whether a given item belongs to a set. In practice, the performance of set membership query is crucial to applications. For example, in network security monitoring applications [9], the long membership query time results in late detection and failed protection. Furthermore, the performance of set membership query directly affects deletion and non-repeatable insertion, which need to search the item first.

Published by Springer Nature Switzerland AG 2021
X. He et al. (Eds.): NPC 2020, LNCS 12639, pp. 373–385, 2021.
https://doi.org/10.1007/978-3-030-79478-1_32

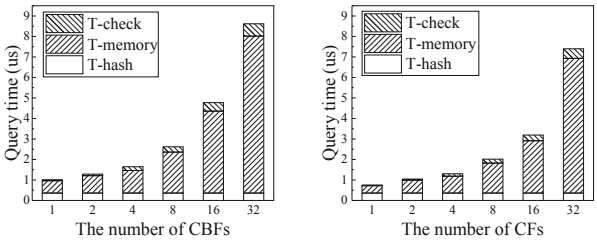


Fig. 1. The time breakdown of a query in DBF and DCF

In real-world applications, precise set storage and query cannot meet the requirements of space and time efficiency. Fortunately, approximate set representation and membership query structures can reduce storage overhead and accelerate query at the cost of a small probability of false positive on the query result, and thus have attracted much efforts in academia [2,7].

The most widely-used approximate set representation structures are *bloom filter* (BF) [2], *cuckoo filter* (CF) [7], and their variants [3,8,14]. BF [2] is a fixed-length array of bits, which are initially set to "0". When inserting an item, BF maps the item into the array by k independent hash functions and transforms the corresponding bits to "1". BF queries whether an item belongs to the set by checking if all its k bits are "1". However, BF does not support deletion. If one deletes an item by flipping its k bits to "0", other existent items that share the k bits will also be regarded as not in the set.

To support deletion, *counting bloom filter* (CBF) [8] replaces each bit of a BF with a counter of d bits. Inserting or deleting an item will increase or decrease the value of the corresponding counter. However, it requires $d\times$ more space than a BF. CF [7] is an array of m buckets and each bucket contains b entries. For an incoming item, CF stores its fingerprint in one of the entries. CF can support deletion by searching and removing the corresponding fingerprint.

The cardinality of the sets in real applications is constantly changing and unpredictable [5,10,11]. For example, the arrival of items in stream processing applications shows high dynamics [12]. Since aforementioned static structures [2,7,8] cannot adjust capacity, they fail to represent the dynamics of set cardinality, and the cost of rebuilding a larger static structure is unacceptable. Very limited work has been done to cope with frequently changing sets. The notable exceptions include *dynamic bloom filter* (DBF) [10] and *dynamic cuckoo filter* (DCF) [5]. To dynamically extend and downsize capacity, DBF [10] appends and merges multiple homogeneous CBFs. However, using multiple CBFs leads to the problem of unreliable deletion. DCF [10] further utilizes multiple CFs to store the fingerprint of an item, and thus supports reliable delete operation.

However, the query performance of existing dynamic set representation structures [5,10] is unsatisfied. Both DBF [10] and DCF [5] are filter-level extensions, which require a single membership query to probe all conjoint filters one by one until the result is found. Today's CPU reads data from memory at a

coarse-grained granularity of cache lines, which are typically 64 to 128 bytes [3]. However, a query in DBF and DCF must access multiple discrete counters and buckets, which are usually much smaller than a cache line [7,8]. Therefore, a query in existing dynamic set representation structures performs a large number of small discrete memory accesses and reads a large amount of unnecessary data from memory, resulting in long membership query time.

In Fig. 1, we examine the query performance of DBF and DCF by experiments with real-world network traffic traces [1]. We deploy DBF and DCF on a server with an Intel 2.60 GHz CPU and 64 GB memory. In each experiment, we run 1×10^6 queries and record the average query time. We breakdown the query time into three parts, including hash computation time, memory access time, and item check time. The results show that with the increase of the number of static filters, the memory access time grows rapidly and contributes a significant fraction of 90% to the query time. If we can reduce the memory access time, the query performance can be greatly improved.

Based on the observation, we propose the *entry-extensible cuckoo filter* (E2CF), a dynamic set representation structure that satisfies the requirement of fast membership query. E2CF exploits the entry-level extension to store entries with the same indexes at continuous physical addresses. By adopting the asynchronous extension and fine-grained splitting, E2CF achieves both space and time efficiency. We implement E2CF and conduct comprehensive experiments with real-world traces to evaluate the design. The results show that compared to existing designs, E2CF reduces the query and insertion time by 82% and 28%, respectively.

To summarize, our contributions are threefold:

- We identify the problem of long membership query time in existing dynamic set representation structures.
- We propose a novel structure to improve membership query performance by exploiting entry-level extension of CF.
- We implement E2CF and conduct comprehensive experiments with real-world traces to evaluate our design.

The rest of the paper is organized as follows. Section 2 introduces the related work. Section 3 presents E2CF design. Section 4 analyzes the performance of E2CF. Section 5 evaluates our design. Section 6 concludes the paper.

2 Related Work

2.1 Static Structures

Bloom Filter and Counting Bloom Filter. *Bloom filter* (BF) [2] contains an array of n bits. Initially, all the n bits are set to "0". When inserting an item, BF maps the item into the array by k independent hash functions and flips the k corresponding bits to "1". When determining whether item x belongs to a set, BF checks whether the k corresponding bits of x are all "1".

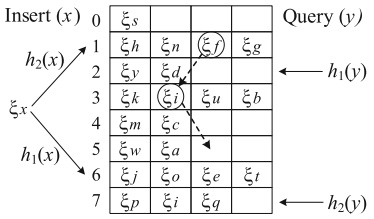


Fig. 2. The process of inserting and querying items in CF

To support deletion in BF, Fan et al. [8] propose *counting bloom filter* (CBF), which replaces each bit of BF with a counter of d bits. When inserting and deleting an item, CBF increases or decreases the value of the corresponding counter. However, it requires $d\times$ more space than a BF.

Cuckoo Filter. *Cuckoo filter* (CF) [7] consists of an array of m buckets and each bucket contains b entries. For an incoming item x, CF generates its fingerprint (denoted as ξ_x) by a hash function and inserts it in one of the entries. To alleviate collisions during insertion, CF computes two candidate buckets for x and stores its fingerprint ξ_x in one entry of the two buckets. The two candidate bucket indexes $h_1(x)$ and $h_2(x)$ are calculated by Eq. (1). With one of the two candidate bucket indexes and ξ_x stored in it, CF can compute the other index of x by using the known index to perform a XOR operation with $hash(\xi_x)$.

$$\begin{aligned} h_1(x) &= hash(x) \\ h_2(x) &= h_1(x) \oplus hash(\xi_x) \end{aligned} \tag{1}$$

Figure 2 presents the examples of inserting x and querying y. For the insertion, since both the two buckets of ξ_x are full, CF randomly kicks out a fingerprint as a victim, e.g., ξ_f, and stores ξ_x in it. Then CF computes the other candidate bucket index for victim ξ_f. Since the other bucket 3 of ξ_f is also full, CF evicts a new victim ξ_i and stores ξ_f in bucket 3. The eviction repeats until all items find empty entries or the number of relocations reaches the pre-defined *maximum number of kickouts* (MNK). For the query of y, CF reads the fingerprints from its two candidate buckets and checks whether ξ_y exists in them. For an item deletion, CF finds the corresponding fingerprint and removes it.

Guo et al. [10] reveal that data sets in real applications are highly dynamic. However, existing static structures [7,8] lack the ability to extend capacity. In addition, allocating large enough capacity in advance will cause a waste of space. Therefore, it is rather important to design a set representation structure that supports dynamically extending and downsizing capacity.

2.2 Dynamic Structures

Dynamic Bloom Filter. Guo et al. [10] propose the *dynamic bloom filter* (DBF), which is an approximate set representation structure that copes with

dynamically changing of set cardinality. A DBF consists of a linked list of s homogeneous CBFs. When the current CBF is full, DBF extends its capacity by appending new building blocks of CBFs. For a query, DBF checks every CBF independently to determine whether the item exists in it.

Dynamic Cuckoo Filter. Chen et al. [5] propose the *dynamic cuckoo filter* (DCF), which uses a linked list of s homogeneous CFs to dynamically adjust capacity. DCF always maintains an active CF. When a new item comes, DCF tries to insert it into the active CF. If the active CF is full, DCF appends a new empty CF and stores the item in it. Then the new CF becomes the active CF. For a query, the DCF, however, needs to probe the candidate buckets in all the CFs to check whether the fingerprint exists in it. Clearly, a query process in DCF will access discrete memory multiple times, resulting in long query time.

Consistent Cuckoo Filter. Luo et al. [11] propose the *index-independent cuckoo filter* (I2CF), which adds and removes buckets adaptively to cope with dynamic set. However, I2CF can only handle small-scale capacity extension. To deal with large-scale highly dynamic data set, Luo et al. [11] further propose the Consistent cuckoo filter, which consists of a linked list of I2CFs.

Existing dynamic structures typically adopt filter-level extension, which requires a membership query to probe all discrete homogeneous filters, resulting in unsatisfied query performance. Differently, E2CF exploits entry-level extension, reduces memory accesses, and further improves query performance.

Table 1. Notations

Notation	Explanation
ξ_x	The fingerprint of the item x
PCF_k	The k_{th} PCF in E2CF
k_1, k_2	The serial number of PCFs which an item belongs to
$h_1(x)$, $h_2(x)$	The two candidate bucket indexes of x in primary CF_0
$h_1(x)'$, $h_2(x)'$	The two candidate bucket indexes of x in PCF_{k_1} and PCF_{k_2}
m	The number of buckets in primary CF_0
b	The number of entries in a bucket of primary CF_0
l_h, l_l	The highest and lowest levels that PCFs exist
f	The length of fingerprint
α	The maximum permissible load factor of PCF

3 Entry-Extensible Cuckoo Filter

3.1 Overview

As aforementioned, the query performance of existing dynamic set representation structures suffers from the large number of small discrete memory accesses. The

E2CF exploits the entry-level extension to avoid many time-consuming memory accesses. An E2CF is initially a standard CF, i.e., CF_0. When E2CF needs to extend capacity, it creates a new *partial cuckoo filter* (PCF) and alternately moves half of the old buckets to the new PCF. We call the newly allocated filter PCF because it only contains partial buckets of the primary standard CF_0. Then, the empty entries of the removed buckets are merged into the adjacent remaining buckets. In this manner, the entries with the same index are stored in continuous physical addresses. For a query, E2CF determines which two candidate buckets the fingerprint resides in and reads them from memory. To improve space efficiency, E2CF allows every PCF to extend capacity independently. E2CF also leverages a fine-grained splitting method to support operations during splitting. We summarize the notations used in this paper in Table 1.

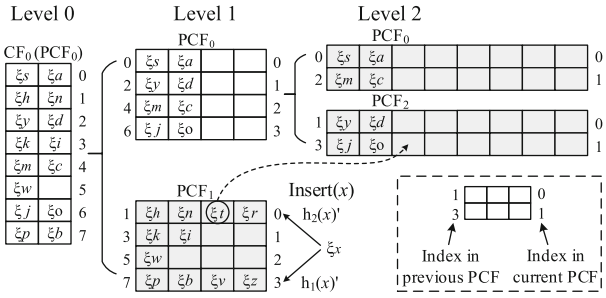


Fig. 3. An example of E2CF

3.2 Entry-Level Extension

E2CF exploits the entry-level extension to reduce the large number of small discrete memory accesses. However, if we directly allocate new memory to extend entries, the physical addresses of entries with the same indexes will be discrete. Fortunately, we find that the physical addresses of the adjacent buckets in a CF are continuous, which could be utilized in entry extension.

Initially, E2CF consists of a primary CF, denoted as CF_0, which has m buckets. Each bucket in CF_0 has b entries. When the number of items increases, E2CF extends its capacity. E2CF first allocates a new PCF_1, which has the same size as CF_0. To leverage the continuous memory addresses of adjacent buckets to extend entries, E2CF moves the buckets with odd indexes from CF_0 to PCF_1. The empty entries of these buckets in CF_0 are then merged into the adjacent buckets with smaller even indexes, which forms new PCF_0. After the extension, both PCF_0 and PCF_1 have $m/2$ buckets and each bucket contains $2 \times b$ entries. The process of moving buckets with odd index to the new PCF is called splitting.

Figure 3 illustrates an example of E2CF. For simplicity, we only plot the fingerprints in the primary CF_0 during splitting. When CF_0 in level 0 starts to

split, E2CF will allocate a new PCF_1 and move the fingerprints in buckets with odd indexes, e.g., ξ_h and ξ_n in bucket 1, to PCF_1. Then the buckets with even indexes in CF_0 absorb the adjacent empty entries. Since PCF_0 and PCF_1 in level 1 are derived from the same PCF in level 0, we call them brother PCFs. However, if we split all PCFs in E2CF simultaneously, the space cost will increase exponentially, which is unacceptable for applications with expensive storage.

To solve the problem, we adopt the asynchronous extension method, which enables each PCF to extend capacity independently. A PCF performs splitting only when one of the following two conditions is met. First, the load factor of a PCF, i.e., the ratio of the used entries to the total entries, exceeds the predefined maximum permissible load factor α. Second, the number of relocations reaches MNK. We will split the PCF that performs the last kicking out.

Once a PCF reaches the split condition, E2CF splits it independently. For example, in Fig. 3, if PCF_0 in level 1 reaches the split condition, E2CF splits PCF_0 into PCF_0 and PCF_2 in level 2. Since PCF_1 in level 1 does not satisfy the split condition, it remains in level 1. At this moment, E2CF contains three grey PCFs, i.e., PCF_0, PCF_1, and PCF_2. The total number of buckets in E2CF remains eight. When a new item comes, E2CF finds its candidate PCFs and buckets according to our insertion method (Sect. 3.3).

To help locate the PCFs and buckets, we assign a serial number for each PCF. After the split of PCF_k in level l, the serial number of the PCF that holds even index buckets remains k. For the newly allocated PCF that receives odd index buckets from the PCF_k, its serial number is the sum of k and 2^l. For example, when the PCF_0 in level 1 splits, the serial number of PCF_0 does not change. The serial number of the newly allocated PCF is the sum of 0 and 2^1, which is two. Algorithm 1 presents the splitting process.

3.3 Operations of E2CF

Insertion. When inserting an item x, E2CF first calculates its fingerprint ξ_x. Then E2CF needs to determine which PCFs (PCF_{k_1} and PCF_{k_2}) and buckets (bucket $h_1(x)'$ and $h_2(x)'$) the fingerprint ξ_x belongs to. For an item x, E2CF computes the primary candidate bucket indexes $h_1(x)$ and $h_2(x)$ by Eq. (1). Since we do not know in which levels the candidate PCFs exist, we search from the lowest level l_l. We calculate the serial number k_1 of the first PCF by Eq. (2). If PCF_{k_1} exists in the level, then we calculate $h_1(x)'$ by Eq. (2). Otherwise, we

Algorithm 1: E2CF: Splitting ()

```
if PCF_k in level l reaches the split condition then
    new PCF_{k+pow(2,l)};
    for n = 1; n < m/pow(2,l); n+ = 2 do
        move PCF_k.B[n] to PCF_{k+pow(2,l)}.B[(n − 1)/2];
    increase the levels of PCF_k and PCF_{k+pow(2,l)} by 1;
```

repeat the same computation from level $l_l + 1$ to l_h until PCF_{k_1} and $h_1(x)'$ are found. Similarly, we can obtain another candidate PCF_{k_2} and bucket indexes $h_2(x)'$ of item x. During the insertion, the PCF that reaches the split condition performs splitting operation.

$$\begin{aligned} k_i &= h_i(x)\%2^l \\ h_i(x)' &= \lfloor h_i(x)/2^l \rfloor \quad i \in \{1, 2\} \end{aligned} \tag{2}$$

Figure 3 presents an example of inserting item x. When there are no empty entries in the two candidate buckets of x, E2CF randomly kicks out a fingerprint ξ_t, which is called a victim, and stores ξ_x. For ξ_t, E2CF calculates its another candidate bucket and continues the relocation. For an insertion, if both two candidate buckets have empty entries, E2CF inserts the item into the bucket in the lower level to ensure load balance. In this manner, the value of $l_h - l_l$ is usually very small in practice, resulting in little index computation overhead. Algorithm 2 presents the detailed process of the insertion in E2CF.

Membership Query. For a query of item x, E2CF computes k_1 and $h_1(x)'$ by Eq. (2). If a matched ξ_x is identified in a bucket of PCF_{k_1}, E2CF returns success. Otherwise, E2CF further calculates k_2 and $h_2(x)'$ and checks whether ξ_x exists in PCF_{k_2}. If both the two candidate buckets do not contain ξ_x, E2CF returns failure. Since E2CF only needs to read two buckets and entries in each bucket have continuous physical addresses, very little memory access overhead is introduced in a query.

Algorithm 2: E2CF: Insert (x)

```
1  fp = ξ_x, h_1(x) = hash(x), h_2(x) = h_1(x) ⊕ hash(fp);
2  for j = 0; j < MNK; j++ do
3      // Calculate k_1 and h_1(x)'
4      for l = l_l; l ≤ l_h; l++ do
5          k_1 = h_1(x)%2^l;
6          if PCF_{k_1} belong to level l then
7              h_1(x)' = ⌊h_1(x)/2^l⌋;
8              break;
9      if j = 0 then
10         calculate k_2 and h_2(x)';
11     if PCF_{k_1}.B[h_1(x)'] or PCF_{k_2}.B[h_2(x)'] has an empty entry then
12         put fp to the empty entry;
13         return success;
14     randomly kick out a victim ξ_t from the two buckets and store fp;
15     fp = ξ_t;
16     let h_1(x) be another primary candidate bucket index of the victim ξ_t;
17 PCF_{k_1}.Splitting () and insert fp;
```

Deletion. For the deletion of item x, E2CF searches the position of ξ_x by a membership query and removes it from E2CF. If the fingerprint ξ_x is not in E2CF, the delete operation returns failure.

However, deletions may result in the reduction of space utilization in E2CF. To improve space efficiency, we compress sparse PCFs to downsize the capacity of E2CF. For a PCF_p in the highest level l_h, if the load factors of both PCF_p and its brother PCF are less than $\alpha/2 - 0.1$, we merge them into a PCF in level $l_h - 1$. If the sparse PCF_p is not in level l_h, we first try to move the fingerprints in level l_h to it. Then, we merge the sparse PCFs in the highest level l_h. With the compression method, we can restrict the maximum level difference in E2CF, i.e., $l_h - l_l \leq 2$, which improves the insertion and query performance.

3.4 Fine-Grained Splitting

The dynamic sets in real applications typically require real-time insert and query operations [9]. However, supporting real-time operations is difficult when E2CF is splitting. To solve the problem, we design a fine-grained splitting method, which allows E2CF to perform operations during splitting.

When a PCF starts to split, we mark it as in the splitting state. In E2CF, we add a 1-bit flag to each bucket. For the primary CF_0, we initialize the flag to "0", which represents that the bucket has not split. When a new item x is inserted into a bucket with flag "0" in the splitting CF_0, we check whether the bucket index $h_1(x)'$ is odd. If so, we flip the flags of old buckets $h_1(x)'$ and $h_1(x)' - 1$ to 1, move the bucket $h_1(x)'$ to the newly allocated PCF, and store ξ_x in the new PCF. Otherwise, we change the flags of old buckets $h_1(x)'$ and $h_1(x)' + 1$, move the bucket $h_1(x)' + 1$ to the new PCF, and store ξ_x in the old PCF. When moving the bucket, we move the flag together. After CF_0 is split into PCF_0 and PCF_1 in level 1, all the flags are converted to "1". So for PCFs in even levels, flag "0" indicates that the bucket has not split; while for PCFs in odd levels, flag "1" does. The PCF splitting process finishes when all flags of the buckets in the old PCF are flipped. Then we reset the PCF to non-splitting state.

In this manner, E2CF moves buckets during insertion. For an operation on the splitting PCF, if its candidate bucket has been moved, we perform it on the newly allocated PCF. Otherwise, we operate on the old splitting PCF.

4 Analysis of E2CF

4.1 Query Performance

According to Fig. 1, the memory access time dominates the query time. Therefore, we use the memory access time T_{mem} to represent the query performance. Today's CPU reads data from memory at cache line granularity. We denote the size of a cache line as M and use T_c to represent the time of reading a cache line size memory. For DBF and DCF, they access small discrete memories that are much smaller than M. Their memory access time is calculated by Eq. (3).

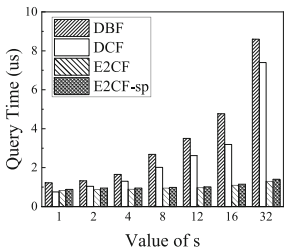


Fig. 4. Query time with different values of s

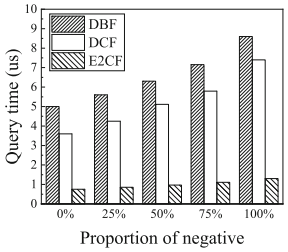


Fig. 5. Query time with different negative query rates

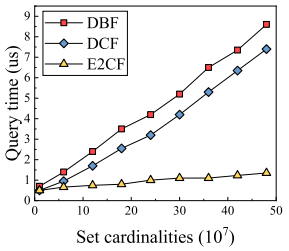


Fig. 6. Query time with different set cardinalities

$$T_{DBF-mem} = k \times T_c \times s,\ T_{DCF-mem} = 2 \times T_c \times s \tag{3}$$

where k represents each CBF in DBF has k independent hash function, and s is the number of static filters in DBF and DCF. The memory access time of a query in E2CF is calculated by Eq. (4), where s is the number of PCFs in E2CF.

$$T_{E2CF-mem} = 2 \times T_c \times \lceil \frac{2^{\lceil log_2 s \rceil} \times b}{\lfloor \frac{M}{f} \rfloor} \rceil \tag{4}$$

In practice, M is typically 64 to 128 bytes. In E2CF, we set $b = 4$ and $f = 16$ bits. Therefore, $\lfloor \frac{M}{f} \rfloor \in [32, 64]$. Also, $s \leq 2^{\lceil log_2 s \rceil} \leq 2 \times s$. Equation (5) presents the range of $T_{E2CF-mem}$, which is much smaller than that of DBF and DCF.

$$T_c \times \lceil \frac{s}{8} \rceil \leq T_{E2CF-mem} \leq T_c \times \lceil \frac{s}{2} \rceil \tag{5}$$

4.2 False Positive Rate

According to [7], the false positive rate of CF can be computed by Eq. (6).

$$FP_{CF} = 1 - (1 - \frac{1}{2^f})^{2b} \approx \frac{2b}{2^f} \tag{6}$$

In E2CF, PCFs may exist in multiple levels, and the number of entries in the PCFs of level l is $2^l \times b$. We use n_l to denote the number of PCFs in level l. The false positive rate of E2CF is calculated by Eq. (7). When all the PCFs exist in one level, the false positive rate of E2CF and DCF [5] is the same.

$$FP_{E2CF} = 1 - \sum_{l=l_l}^{l_h} \frac{n_l}{s} \times (1 - \frac{1}{2^f})^{2 \times 2^l \times b} \approx \frac{b}{2^f \times s} \sum_{l=l_l}^{l_h} (2^{l+1} \times n_l) \tag{7}$$

5 Performance

5.1 Experiment Setups

We implement E2CF and make the source code publicly available[1]. We conduct the experiments on a server with an Intel 2.60 GHz Xeon E5-2670 CPU and 64 GB RAM. The CPU has 32 KB L1 data cache, 32 KB instruction cache, 256 KB L2 cache, and 20 MB L3 cache. The size of a cache line in the server is 64 bytes. We use both real-word network traffic traces [1] and synthetic data set to evaluate the performance.

We compare E2CF with DBF [10] and DCF [5], respectively. For E2CF and DCF, we set $m = 2^{22}$, $b = 4$, $\alpha = 0.9$, and $f = 16$. For DBF, the parameters are calculated based on the same false positive rate as E2CF. We run each experiment five times and record the average value.

5.2 Results

Figure 4 compares the query time with different values of s. The E2CF-sp means some PCFs are in the splitting state during the query. We perform negative queries, which mean searching for a non-existent item. The result shows that the splitting state barely affects the query performance and E2CF reduces the query time by up to 85% and 82% compared to DBF and DCF, respectively. That is because DBF and DCF need to access small discrete memory multiple times while E2CF only needs to read entries in continuous memory.

Figure 5 presents the query time with different proportions of negative queries when $s = 32$. The result shows that E2CF always greatly outperforms DBF and DCF. Figure 6 compares the query time [13] with different set cardinalities. The result shows that E2CF reduces the query time by up to 82% compared to DCF.

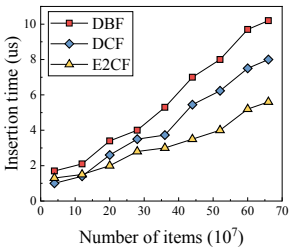
Figure 7 shows the instantaneous insertion time with different MNKs. We test non-repeatable insertion, which does not allow an item to be inserted twice. With the increase of the number of inserted items, E2CF reduces the instantaneous insertion time by up to 45% and 28% compared to DBF and DCF, respectively.

Figure 8 plots the cumulative insertion time with different MNKs. When we increase MNK from 10 to 40, more relocations will occur, and the insertion time of DCF and E2CF increases. The result shows that compared to DBF and DCF, E2CF reduces the cumulative insertion time by up to 41% and 22%.

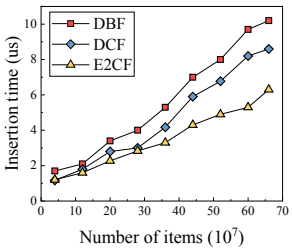
Figure 9 presents the deletion time with different values of s. When s increases, the deletion time of DBF and DCF grows linearly, while the time of E2CF is essentially unchanged. Figure 10 plots the deletion time with different set cardinalities. The result shows that E2CF reduces the deletion time by up to 84% and 79% compared to DBF and DCF, respectively.

Figure 11 compares the memory cost when increasing set cardinalities. E2CF-syn denotes E2CF without asynchronous extension. As shown in the figure, E2CF

[1] https://github.com/CGCL-codes/E2CF.



(a) MNK=10



(b) MNK=40

Fig. 7. Instantaneous insertion time

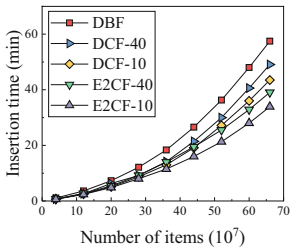


Fig. 8. Cumulative insertion time

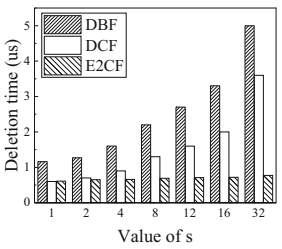


Fig. 9. Deletion time with different values of s

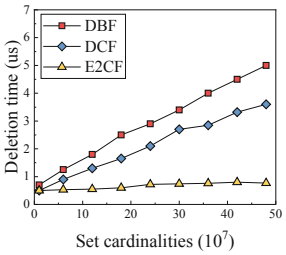


Fig. 10. Deletion time with different set cardinalities

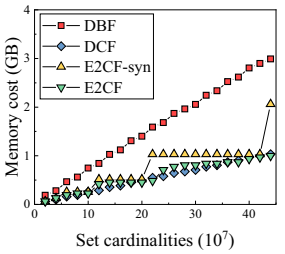


Fig. 11. Memory cost with different set cardinalities

reduces the memory cost by up to 66% and 50% compared to DBF and E2CF-syn and achieves comparable space efficiency with DCF. But note that E2CF greatly outperforms DCF in terms of the query, insert, and delete performance.

6 Conclusion

In this paper, we find that the query performance of existing set representation structures is subject to long memory access time. We design and implement the E2CF, a dynamic structure that supports fast membership query for large-scale dynamic data set. The E2CF exploits entry-level extension to extend and downsize capacity. We further adopt asynchronous extension and fine-grained splitting methods to achieve space and time efficiency. Theoretical analysis and experiment results show that E2CF greatly outperforms existing schemes.

Acknowledgement. This research is supported in part by the National Key Research and Development Program of China under grant No. 2016QY02D0302, and NSFC under grant No. 61972446.

References

1. WIDE project (2020). http://mawi.wide.ad.jp/mawi/
2. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 442–426 (1970)
3. Breslow, A., Jayasena, N.: Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. Proc. VLDB Endow. **11**(9), 1041–1055 (2018)
4. Chen, H., Jin, H., Wu, S.: Minimizing inter-server communications by exploiting self-similarity in online social networks. IEEE Trans. Parallel Distrib. Syst. **27**(4), 1116–1130 (2015)
5. Chen, H., Liao, L., Jin, H., Wu, J.: The dynamic cuckoo filter. In: Proceedings of ICNP, pp. 1–10 (2017)
6. Dayan, N., Athanassoulis, M., Idreos, S.: Optimal bloom filters and adaptive merging for LSM-trees. ACM Trans. Database Syst. **43**(4), 16:1–16:48 (2018)
7. Fan, B., Andersen, D.G., Kaminsky, M., Mitzenmacher, M.: Cuckoo filter: practically better than bloom. In: Proceedings of CoNEXT, pp. 75–87 (2014)
8. Fan, L., Cao, P., Almeida, J.M., Broder, A.Z.: Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans. Networking **8**(3), 281–293 (2000)
9. Groza, B., Murvay, P.: Efficient intrusion detection with bloom filtering in controller area networks. IEEE Trans. Inf. Forensics Secur. **14**(4), 1037–1051 (2019)
10. Guo, D., Wu, J., Chen, H., Yuan, Y., Luo, X.: The dynamic bloom filters. IEEE Trans. Knowl. Data Eng. **22**(1), 120–133 (2010)
11. Luo, L., Guo, D., Rottenstreich, O., Ma, R.T.B., Luo, X., Ren, B.: The consistent cuckoo filter. In: Proceedings of INFOCOM, pp. 712–720 (2019)
12. Monte, B.D., Zeuch, S., Rabl, T., Markl, V.: Rhino: efficient management of very large distributed state for stream processing engines. In: Proceedings of SIGMOD, pp. 2471–2486 (2020)
13. Peng, B., Lü, Z., Cheng, T.C.E.: A tabu search/path relinking algorithm to solve the job shop scheduling problem. Comput. Oper. Res. **53**, 154–164 (2015)
14. Wang, M., Zhou, M., Shi, S., Qian, C.: Vacuum filters: more space-efficient and faster replacement for bloom and cuckoo filters. Proc. VLDB Endow. **13**(2), 197–210 (2019)