The Logarithmic Dynamic Cuckoo Filter

Fan Zhang, Hanhua Chen*, Hai Jin
National Engineering Research Center for Big Data Technology and System
Cluster and Grid Computing Lab
Services Computing Technology and System Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan 430074, China
{zhangf, chen, hjin}@hust.edu.cn

Pedro Reviriego
Departamento de Ingeniería Telemática
Universidad Carlos III de Madrid
Avenida de la Universidad
Leganés, 28911, Madrid, Spain
revirieg@it.uc3m.es

2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00087

***Abstract*—The emergence of big data applications makes efficient representation for large-scale dynamic data sets a challenge. The state-of-the-art design, *i.e.*, the *dynamic cuckoo filter* (DCF), provides extensible approximate set representation by employing a novel chain based data structure which allows appending new building cuckoo filter blocks. However, such a design needs linearly increasing computation costs and memory space when a set scales. This makes it inefficient for big data sets. In this paper, we propose a novel data structure for dynamic big data sets, called *logarithmic dynamic cuckoo filter* (LDCF). LDCF uses a novel multi-level tree structure and reduces the worst insertion and membership testing times from $O(N)$ to $O(1)$, where N is the size of the set. At the same time, LDCF reduces the memory cost of DCF as the cardinality of the set increases. Comprehensive experiment results show that LDCF significantly reduces the membership checking time and the memory space cost for large-scale datasets compared to state-of-the-art designs.**

***Index Terms*—Bit data, dynamic set representation, set membership testing, cuckoo filter.**

I. Introduction

With the emergence of big data applications, set representation becomes an important and challenging problem. By organizing the set elements using some form of data structure, set representation enables the access to the information of the elements, supporting operations such as element insertion, membership testing, and element deletion. Membership testing should support checking whether an element is a member of a given data set. A time and space efficient data structure for set representation is important in various kinds of applications, *e.g.*, cloud storage [1], privacy protection [2], k-mers counting in DNA sequencing [3], and network searching [4].

One of the most basic methods for set representation is the conventional hash coding [5], which organizes a hash area into an array of cells to represent a set. It uses a hash function $h(\cdot)$ to compute an address for an element of a given set $S = \{x_1, x_2, \ldots, x_n\}$. The raw data of an element x_i is stored in the cell with the corresponding $h(x_i)^{th}$ address. If one tests whether an element x_j belongs to S, it needs to compute $h(x_j)$ and check whether x_j matches the data stored in the $h(x_j)^{th}$ cell. The raw data based membership testing incurs no false positives. However, such a structure is inefficient in memory space for set representation.

*The Corresponding Author is Hanhua Chen (chen@hust.edu.cn).

In order to improve the memory efficiency, approximate set representation data structures, *e.g.*, *bloom filter* (BF) [6] and *cuckoo filter* (CF) [7] have been proposed. These approximate set representation structures store Boolean values or fingerprints instead of the raw data. They achieve high space efficiency at the cost of an error in set membership testing with low probability. The approximate set membership testing techniques have been widely adopted in various applications [8].

A problem of the existing BF and CF designs is that they can only represent a set with a known upper bound set size. However, emerging big data applications often involve large-scale dynamic data sets. For example, a real-time streaming application [9] involves large-scale data sets with elements continuously joining and leaving dynamically. A cloud storage application uses approximate set representation data structures for avoiding storing duplicate file chunks and saving storage space [10]. Users can upload new files or delete files at any time, and their behaviour is hard to predict. Thus, the upper bound of the total number of file chunks cannot be estimated in advance. Even if there is a theoretically upper bound of the file chunks, using such an upper bound value as the capacity of a BF or a CF will incur unnecessarily large memory space cost and low utilization in most cases. Therefore, designing a flexible set representation structure to cope with large-scale dynamic data sets is a challenging issue.

To solve the problem, we proposed the *dynamic cuckoo filter* (DCF) [11]. The DCF is made up of a linked list of s CF building blocks. DCF extends its capacity by appending a new CF building block to the linked list when it becomes full (*i.e.*, an insertion failure happens in the current strucuture). The insertion of an element in DCF is performed by inserting the element in the current active CF at the end of the link. The DCF structure can flexibly extend or reduce to cope with the dynamic change of set sizes as well as supporting reliable deletions. However, checking an element against a DCF needs to check every CF in the data structure in the worst case. When the size of dataset scales, DCF's computation and space costs for set membership testing grow linearly.

To solve the above problem, in this paper, we propose a novel approximate set representation data structure for dynamic big data sets, called *logarithmic dynamic cuckoo filter* (LDCF). The basic LDCF designs a novel scalable multi-level

978-1-7281-9184-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00087

tree structure to organize CF building blocks, instead of using classical chain-like structures. Such a design significantly reduces both the computation cost of membership testing and the memory cost of the structure to a logarithmic scale. We further design a compacted LDCF, which reduces the inserting and membership testing time to $O(1)$ with the same scale of memory cost. The compacted LDCF reduces the total memory cost of the DCF with the ratio of $\frac{f-l}{f}$, where f is the fingerprint length and l is the number of levels in LDCF.

The main contributions of this work are threefold:

- We propose LDCF, a novel approximate set representation data structure for dynamic big data sets, which reduces the worst inserting and membership testing time from linear to logarithmic level.
- We propose a compacted LDCF design, which further reduces the inserting and membership testing time to $O(1)$ with the same scale of memory cost.
- We conduct comprehensive experiments using large-scale real world data sets to demonstrate the efficiency of LDCF.

The rest of this paper is structured as follows. Section II reviews the related work. Section III introduces the detail of the LDCF design and its operations. Section IV theoretically analyzes the efficiency of LDCF and the parameter optimization. Section V evaluates the performance of LDCF. Section VI concludes this work.

II. Related Work

A. Bloom Filter and its Variations

The *bloom filter* (BF) [6] is one of the most basic and popular data structures for approximate set representation. A BF is basically a vector of m bits which are all initially set to "0". The BF maps every element in a set $S = \{x_1, x_2, \ldots, x_n\}$ into the m bit addresses using k independent hash functions $h_1(\cdot), \ldots, h_k(\cdot)$. For each element x in S, all the $h_i(x)$ bits $(1 \leq i \leq k)$ are set to "1". If all these k bits are "1", we determine y belongs to S with high probability; otherwise, y is definitely not a member of S. To support deletion, the *counting bloom filter* (CBF) [7] uses an r-bit counter instead of each bit of a BF. The insertion or deletion of an element x only needs to increase or decrease the value of the $h_i(x)$ counters $(1 \leq i \leq k)$ by one. The memory space of a CBF is r times of that of a BF. Given a pre-constructed data structure and an allowed false positive rate, both BF and CBF have a maximal capacity. When the data set exceeds the upper bound of the capacity, BFs suffer a large number of false positives.

To cope with the problem of the capacity limit of BFs, Guo *et al.* [12] propose the *dynamic bloom filter* (DBF). A DBF is made up of a linked list of multiple CBF building blocks. Once the number of elements in a CBF reaches its maximal capacity, it appends a new CBF building block for extending the capacity. Checking an element y against DBF needs to check every CBF in the linked list until finding one with all the $h_i(y)$ bits $(1 \leq i \leq k)$ being non-zero. The DBF design cannot support reliable deletion [11], [12], since it cannot distinguish which CBF stores an element x if the matching element is found in multiple CBF building blocks. This multiple address problem becomes more serious when the number of CBF building blocks increases [11].

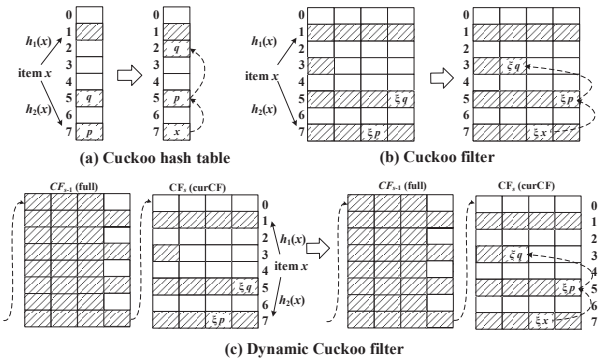

Fig. 1: Cuckoo hash table, cuckoo filter, and dynamic cuckoo filter

Almeida *et al.* propose the *scalable bloom filter* (SBF) [13], which consists of a set of BFs. The design is similar with DBF. However, the size and the number of hash functions of a newly added BF are configured larger than those of the previous BFs. Their design can make the false positive rate converge to a certain value. However, in the SBF, the computation cost for the hash functions increases rapidly with the increase of the number of BF building blocks. As a result, SBF has longer time for membership testing.

Yang *et al.* propose the *shifting bloom filter* (ShBF) [14] to support more operations including membership testing, association, and multiplicity. The ShBF is an m-bit array and uses k independent hash functions to project elements. When performing insertion for an element x, ShBF sets both the $h_i(x)$ bits and the $(h_i(x)+o(x))$ bits $(1 \leq i \leq k)$ to '1', where $o(x)$ is an offset which is determined by different operations. Such a design cannot support large-scale dynamic datasets as the regular BF design.

Zhang *et al.* propose the *SuRF* [15], an approximate membership testing structure designed for both point and range queries. SuRF is based on a new trie data structure, called *fast succinct trie*, that matches the performance of the state-of-the-art point and range query schemes with a lower memory usage. However, SuRF does not support dynamic datasets, and it is orthogonal to our work.

B. Cuckoo Hash Table and Cuckoo Filter

A cuckoo hash table [16] is an extension of the conventional hash table [5] with an array of cells, each for storing an element. Using a cuckoo hash table, each element has two candidate cells which addresses are computed by two hash functions $h_1(\cdot)$ and $h_2(\cdot)$. When an element x is inserted, the table checks both $h_1(x)$ and $h_2(x)$ cells and stores x on any of those cells if they are empty. If not, one of the stored elements will be evicted to its alternative cell. Such evicting process continues until all the elements are stored or the total number of relocations exceeds a certain bound. Figure 1(a) shows an example of a cuckoo hash table with eight cells. In this table, the 1^{st}, 5^{th}, and 7^{th} cells are occupied, while the

rest are empty. When inserting a new element x whose two candidate cells are the 1^{st} and the 7^{th} cells, we find they are both full. Therefore, the cuckoo hash table randomly picks a cell, *e.g.*, the 7^{th} cell, evicts the stored element p, and inserts x into this cell. The victim p is then relocated to its alternative cell (the 5^{th} cell), by evicting the stored element q. Finally, q is relocated to an empty alternative cell (the 2^{nd} cell).

By extending the cuckoo hash table, Fan *et al.* [17] propose the *cuckoo filter* (CF) to support approximate set membership testing. CF replaces the raw data of an element x with its fingerprint (ξ_x) to improve space efficiency. Formally, a CF leverages an array of t buckets, each consisting of b entries. To enable relocations, the CF leverages the partial-key cuckoo hashing. Specifically, it computes the alternative bucket address by performing an XOR operation on the current bucket address and the evicted fingerprint. The two candidate bucket addresses can be computed by Eq. (1),

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

Figure 1(b) illustrates an example of a CF which consists of eight buckets, and each bucket has four entries. To insert x, the CF first computes the two candidate buckets addresses, *i.e.*, the 1^{st} and the 7^{th} buckets, and finds they are both full. Therefore, the CF randomly selects and evicts a fingerprint (*e.g.*, ξ_p in the 7^{th} bucket). Then the victim ξ_p relocates itself to the alternative 5^{th} bucket by evicting the stored fingerprint ξ_q. Generally, we can compute the upper bound of the false positive rate of the CF by Eq. (2) [17],

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

The CF performs deletion by finding a matched fingerprint of an element x_i stored in the corresponding bucket, and then removes it. The CF supports reliable deletion since removing a fingerprint of x_i does not affect the membership checking results of any other elements. The problem of CF is that it is unable to support dynamic data sets.

Lang *et al.* [18] examine how to choose BF or CF to maximize the lookup performance under different workloads. Breslow *et al.* propose the morton filter [19], a variant of CF which makes more efficient use of memory cache to achieve higher throughput. Wang *et al.* propose vacuum filter, [20], which designs a multi-range alternate function instead of the partial-key cuckoo hashing function (Eq. (1)) in the CF, to achieve higher data locality and better memory efficiency. The above researches are orthogonal to our work.

C. Dynamic Cuckoo Filter

To support an extensible dynamic data set, we proposed the DCF structure [11] which leverages a standard CF as the building block and consists of a number of s linked homogeneous building blocks, *i.e.*, $\{CF_0, ..., CF_{s-1}\}$. Each CF building block uses a counter for recording the number of stored elements. If the value of the counter in a building block is less than a predefined capacity, the building block is regarded as active. When inserting an element into a DCF, one inserts the element into the current active CF. If an insertion failure happens in the current CF, DCF appends a new empty building block and inserts the element into the new CF. Figure 1(c) shows the structure of a DCF.

TABLE I: Comparison of Existing Work

Data structure	Insert Speed	Space cost	Deletion	Scalability
BF [6]	$1\times$	$1\times$	no	no
CBF [7]	$1\times$	$s\times$	yes	no
DBF [12]	$1\times$	$s\times$	yes	yes
SBF [13]	$< 1\times$	$s\times$	yes	yes
CShBF [14]	$0.5\times$	$s\times$	yes	no
PBF [21]	$\frac{1}{log_2\lvert S\rvert}\times$	$log_2\lvert S\rvert\times$	no	yes
CF [17]	$1.5\times$	$\leq 1\times$	yes	no
DCF [11]	$1.5\times$	$\leq s\times$	yes	yes
LDCF	$1.5\times$	$< s\times$	yes	yes

Performing set membership checking in a DCF needs to check every CF building block. If DCF finds a matched fingerprint, it returns a positive result. If none of the CF building blocks stores a matched fingerprint, it determines that the element does not belong to the set. The time complexity of the membership check operation is $O(bs)$, where s is the number of building blocks in a DCF and b is the number of entries in a bucket. When one performs membership checking for an element which does not belong to the set, the DCF needs to check this element in all the s CFs. Thus, the DCF reports the true positive result only when no false positive occurs in any CF. We denote the false positive rate of a single CF as ϵ_{CF}, then, the probability that no false positives occurs in any of the s CFs is $(1-\epsilon_{\text{CF}})^s$. Therefore, we can compute the upper bound of the false positive rate of a DCF by,

$$\epsilon_{\text{DCF}} = 1 - (1 - \epsilon_{\text{CF}})^s \tag{3}$$

By replacing ϵ_{CF} with Eq. (2), the false positive rate is,

$$\epsilon_{\text{DCF}} = 1 - (1 - \epsilon_{\text{CF}})^s = 1 - (1 - \frac{1}{2^f})^{2bs} \approx \frac{2bs}{2^f} \tag{4}$$

It is not difficult to see that the DCF design needs linearly increasing computation costs and memory space when the set scales. In the worst case, checking an element needs to look up against all the s CF building blocks in the DCF. Facing large-scale sets in real world applications, such a design may be prohibitively costly in computation and memory.

Table I compares previous designs and LDCF in the insert speed, space cost, support of deletion, and scalability.

D. Other Set Representation Structures

The *quotient filter* (QF) [22] is an approximate set representation structure which contains 2^q slots. The QF hashes every element to a fixed length bit string. It uses the first q bits as an address and the remaining r bits as a fingerprint for storage. It deals with the collisions using a variance linear probing scheme. It supports membership checking and deletion by setting metadata such as offsets and flags of occupied cells in each slot. To support counting, Pandey *et al.* propose the *counting quotient filter* (CQF) [23], an extremely compact quotient filter which uses more metadata in a slot to count and compress duplicated fingerprints. The CQF cannot support

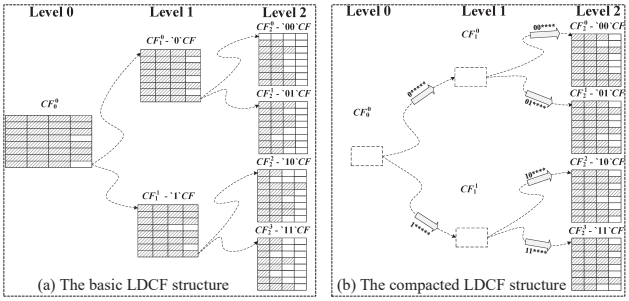

Fig. 2: The LDCF structure

dynamically increasing datasets. When the number of elements in CQF reaches its capacity, it can be resized by reconstructing a new filter with $2^{q'}$ slots ($q' > q$).

Sketches are classical data structures for element frequency estimation. One of the most widely used sketch structure is the *count-min sketch* [24], which projects elements in $d \times w$ counters. Count-min sketch employs d hash functions, where each hash function projects an element to w values. However, in a count-min sketch, the estimated count of each element may be larger than the accurate count due to hash collisions, especially for elements who have low count values. Thus, count-min sketch cannot efficient support membership checking due to the high false positive rate.

Zhou *et al.* propose the *cold filter* [25], a two stage sketch structure for more accurate counting for cold elements. It estimates and stores cold elements in the first stage, and stores hot elements in the second. Cold filter improves the accuracy for applications such as Top-k elements and heavy hitters identification. However, it does not support membership checking and dynamically changing sets.

III. LDCF Design

A. The Basic LDCF Design

In this section, we present the overview of the basic LDCF design. LDCF exploits a multi-level tree structure. Compared with a classical chain-like structure [11], a tree-like structure achieves much lower time complexity of element insertion and look up.

Figure 2(a) shows the structure of the basic LDCF design. On the l^{th} level, the basic LDCF contains 2^l CFs. For simplicity, we use the notation CF_p^q to denote the q^{th} CF building block on the p^{th} level. CF_p^q contains an array of t buckets $\{B_{p,q}(0), ..., B_{p,q}(t\text{-}1)\}$, and each bucket has a number of b entries. Table II lists the notations in this design.

Initially, LDCF starts with a single CF on level 0, *i.e.*, CF_0^0. An element x can be inserted by appending its fingerprint ξ_x into an empty bucket of a corresponding bucket in CF_0^0. The size of the entry in each bucket in CF_0^0 is the same as the length of ξ_x. When an insertion failure happens in CF_0^0, we regard CF_0^0 as it is full. Then LDCF extends the capacity by appending the 1^{st} level CFs, *i.e.*, CF_1^0, and CF_1^1, and performs insertion in the new CFs for the element that failed insertion. When inserting an element x into the 1^{st} level CFs, we first generate its fingerprint ξ_x and check the first bit of ξ_x. If ξ_x starts with '0', we insert ξ_x into CF_1^0; otherwise, we insert ξ_x into CF_1^1. Here, our insight is that there is no need to store the first bit of ξ_x in CFs of the 1^{st} level of LDCF. That is to say, we can save one bit in every entry with the 1^{st} level CFs compared to CF_0^0. Similarly, when the 1^{st} level CFs are full, we append the 2^{nd} level, which contains four CFs, including CF_2^0, CF_2^1, CF_2^2, and CF_2^3. It is clear, we can assign CF_2^0 for the coming elements with fingerprints that start with '00'. In the same way, CF_2^1, CF_2^2, and CF_2^3 are for elements with the fingerprint with the prefixes '01', '10', and '11', respectively. We can save two bits for every entry in these four CFs on the 2^{nd} level. More generally, on the i^{th} level, LDCF has a set of 2^i CFs, *i.e.*, $\{CF_i^0, CF_i^1, ..., CF_i^{2^i-1}\}$. Each CF on the i^{th} level can save a number of i bits for each entry. Therefore, such a design can significantly reduce the memory cost when facing large-scale sets. Moreover, this allows us to build CFs with longer fingerprints when constructing an LDCF. Longer fingerprints can reduce the false positive of a CF building block (Eq. (4)).

LDCF supports membership checking by looking up the queried element in each level in turn until its fingerprint is found. On each level, only one corresponding CF will be checked. Thus, the time complexity for membership checking in LDCF with l levels is $O(l)$. Compared with a DCF with the same capacity, *i.e.*, a DCF consists of $(2^l - 1)$ CFs, LDCF significantly reduces the time complexity to a logarithmic scale. LDCF can also support deletion by removing the found fingerprint. Although LDCF introduces slightly higher computation time for finding the corresponding leaf CF when insertion compared with the DCF design, the additional computation cost is $O(l)$, where l is the number of levels. Such a cost is low compared to the potential high relocation time during one insertion in a CF building block [26].

B. The Compacted LDCF Design

The basic LDCF design can achieve higher space efficiency and much shorter processing latency for insertion and membership testing than DCF when LDCF is full. However, each

TABLE II: Notations in LDCF

Notations	Description
CF_p^q	the q^{th} CF in the p^{th} level
S	a set of elements to be represented
N	the expected size of the set S
x_i	the i^{th} element in set S
ξ_{x_i}	the fingerprint of the element x_i
f	the number of bits in a fingerprint
s	the number of CFs in LDCF
t	the number of buckets in each CF
b	the number of entries in each bucket
l	the number of levels in LDCF
μ_{x_i}, ν_{x_i}	two bucket addresses of the element x_i
$B_{p,q}(\mu_{x_i}), B_{p,q}(\nu_{x_i})$	two candidate buckets of x_i in CF_p^q
ϵ_{CF}	false positive rate of each CF
ϵ_{LDCF}	false positive rate of LDCF
c	the capacity of a CF
α	the load factor of LDCF

time when we extend a new level, most of the CFs on the new level have low space utilization. Thus, the space efficiency may be lower than that of DCF unless the current level l is higher than $\frac{f}{2}$. To achieve higher space efficiency, we consider to compact the basic LDCF structure, which moves the elements from CFs on the lower levels to the higher level so that the CFs on the higher level have larger occupancy. Moreover, we can remove all the empty CFs on the lower levels. Each time when we extend a new level, the capacity of two new children CFs is twice that of the parent CF. That is to say, the space is enough to separate the elements stored in the parent CF into two new children CFs. Then, the parent CF is empty and can be removed from LDCF to release memory space.

Accordingly, we propose the *compacted LDCF*, an improved LDCF design which only keeps the highest level of CFs. Figure 2(b) shows the structure of the compacted LDCF design. The logical structure is still a multi-level binary tree structure. Differently from the basic design, only the leaves of LDCF are a set of CF building blocks. Initially, the compacted LDCF structure only has a root node, which is originally a single CF building block, *i.e.*, CF_0^0. Once a node in the tree structure becomes a parent node (*e.g.*, CF_1^0 and CF_1^1), it deletes the corresponding CF building block and only keeps two pointers to its two children nodes. Thus, in the compacted LDCF structure, the number of CF building blocks is the same as that of the leaf nodes.

The operations of the compacted LDCF are performed as follows.

Insert. Initially, the compacted LDCF is made up of a single CF in level 0 (*i.e.*, CF_0^0), and the insert operation is the same with that of a standard CF. If the CF on level 0 is full, we add two new CFs on level 1, one is the '0' CF (CF_1^0) and the other is the '1' CF (CF_1^1). Different from CF_0^0, the length of each entry in CF_1^0 and CF_1^1 is reduced by one. We insert new elements to the corresponding level 1 CFs according to the first bit of their fingerprints. For an element with the fingerprint starting with a '0' bit, LDCF inserts it into CF_1^0. When inserting, we omit the first bit of the fingerprint, and insert the remaining $f-1$ bits. For an element with the fingerprint starting with a '1' bit, the procedure is similar. When one of the CFs on level 1 is full (*e.g.*, CF_1^1), we append two corresponding new CFs on level 2 (the '10' CF (CF_2^2) and the '11' CF (CF_2^3)), and decrease the length of each entry in these CFs by one, and so forth.

Algorithm 1 describes the insert operation in detail. Each time when inserting a new element x, it searches the corresponding leaf CF from the root according to its fingerprint. Assume the found CF is on level i. In the found CF, we firstly compute the two candidate bucket addresses for x by using Eq. (1), and check whether an entry of these two buckets is empty. If we find an empty entry, we insert the last $f-i$ bits of x's fingerprint into the empty entry. If all the entries are full, we randomly evict a fingerprint ξ_y' (whose length is only $f-i$ bits), and insert the last $f-i$ bits of x's fingerprint. For the evicted fingerprint ξ_y', we recover it to the original length of f by adding the first i bits of x's fingerprint before ξ_y'. We use the new combined fingerprint for computing the alternative bucket address of element y. If the number of relocations achieves the specified threshold, the algorithm appends new CFs, and inserts the victim to the corresponding new CF.

Algorithm 1: **Insert** (x)

```
curLevel ← 0;
curLoc ← root;
ξ_x ← fingerprint(x);
μ_x ← hash(x);
ν_x ← μ_x ⊕ hash(ξ_x);
while curLoc.CF = NULL do
    curLevel ← curLevel +1;
    pre ← getPrefix(ξ_x, curLevel);
    curLoc ← curLoc.getChild(i, pre);
curCF ← curLoc.CF;
bucket = randomly choose μ_x or ν_x;
for r = 0 to MAX_RELOCATION do
    randomly choose ξ_y from one entry of curCF.B(bucket);
    victim← ξ_y;
    ξ_x = cutPrefix(ξ_x, pre);
    insert ξ_x into curCF.B(bucket);
    victim ← AddPrefix(victim, pre);
    bucket = bucket ⊕ hash(victim);
if curCF is full then
    curLoc ← Append(curCF, curLevel);
return true.
```

Compared with the insertion operation of the DCF design, LDCF introduces slightly higher computation time for finding the corresponding leaf CF. The additional computation cost is $O(l)$, where l is the number of levels. Such a cost is low compared with the potential high relocation time during one insertion in a CF building block [26]. Moreover, in DCF, all the insertion operations need to be performed on the same CF. This makes it hard to achieve high parallelism due to the complex relocation operations. Instead, in LDCF multiple insertion operations in different CF building blocks can be performed in parallel without any lock mechanism.

Append. Whenever a CF in the current LDCF is full, LDCF appends two corresponding new CFs as its two children in a higher level. Each time when we append two new CFs on a higher level, we need to remove all the elements from the current level CFs and insert them back into the new level CFs. Specifically, we move the elements from the lower level CFs to the corresponding bucket addresses of the new level CFs according to the prefix of their fingerprints. For example, for an initial compacted LDCF which has only one level (level 0), when we append two CFs on the 1^{st} level, we need to move all the elements stored in the CF on level 0 to level 1. For example, if there are four elements $y_i(i=0,1,2,3)$ with the six-bit fingerprints ξ_{y_0}='001000', ξ_{y_1}='101101', ξ_{y_2}='110010', ξ_{y_3}='011110' in address d in the CF on level 0, we need to move the elements y_0 and y_3 to the address d of the '0' CF on level 1, and the rest two fingerprints to the address d of the '1' CF on level 1. Finally, we delete the lower level CFs in order to save space. Algorithm 2 shows the procedure of the append operation.

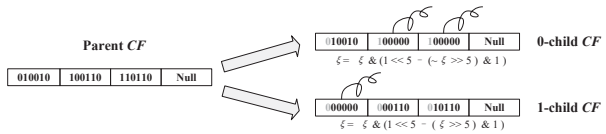

Fig. 3: Parallel evicting operation when appending child CFs

To speedup the append operation, we use the efficient bitwise operation for quickly removing fingerprints whose first bit does not correspond to the current child CF. Figure 3 shows an example. When appending, the parent CF is copied to two new children CFs. For each fingerprint ξ in '0'-child CF, we compute $\xi = \xi$ & $(1 \ll (l'\text{-}1)$ - $(\sim \xi \gg (l'\text{-}1))$ & 1), where l' is the current length of ξ (in Fig. 3, l'=6). Thus, if ξ starts with '0', *e.g.*, ξ = '010010', it will perform $\xi = \xi$ & '011111'. After the computation of ξ, the last five bits of ξ are still '10010'. But if ξ starts with '1', *e.g.*, ξ = '100110', it will perform $\xi = \xi$ & '100000'. The last five bits of ξ will be changed to '00000'. Thus, we only need to remove the first bit of each fingerprint and evict all the fingerprints which are equal to '00000'. Similarly, for each fingerprint ξ in '1'-child CF, we perform $\xi = \xi$ & $(1 \ll (l'\text{-}1)$ - $(\xi \gg (l'\text{-}1))$ & 1) to evict all the fingerprints that start with '0'.

When the two children CFs finish the evicting operation, the parent CF generates two pointers to them and then releases its space. Membership check operations can then be performed on these two new CFs correspondingly. During the whole procedure, the parent CF can still be used for membership test operations. If there are new elements which need to be inserted before the two children CFs are ready for insertions, the elements will be cached in a queue temporarily.

Membership Test. The membership test operation with a compacted LDCF checks the queried element in each level in turn until the element is found. On each level, only one corresponding CF will be checked. Algorithm 3 describes the membership testing procedure in detail. For checking an element y with a level l LDCF, the algorithm firstly computes the fingerprint ξ_y and the two candidate bucket addresses of y. Then the algorithm stars from the root point. If CF_0^0 is null, the algorithm keeps searching in the '0' or '1' child CF of the current CF on the $i+1_{th}$ level according to the i_{th} bit of ξ_y, until we find a CF which is not null. When we check y against a CF on level i, we check whether there is a fingerprint which can match the rest $f-i$ bits of ξ_y. The algorithm returns true when it finds a matched fingerprint.

The membership test operation in the compacted LDCF is performed in only one CF. Therefore, compared to the basic design, the compacted LDCF further reduces the complexity of membership testing from $O(\log N)$ to $O(1)$. Moreover, in the basic LDCF design, CFs on lower levels can be a bottleneck for membership checking. For example, all the membership test queries should access CF_0^0. But in the compacted LDCF, such a bottleneck does not exist since each CF would have approximately the same membership test workload. Thus, we can easily run multiple accesses in parallel.

Delete. To delete an element y from a level l LDCF, we first perform the membership test operation for element y. If the last $f-i$ bits of the corresponding fingerprint ξ_y can be found in a CF on level i, LDCF removes the matched fingerprint directly from this CF by setting all the $f-i$ bits to "0". Algorithm 4 presents the procedure of deletion. The time complexity of deletion is the same as that of the membership checking of LDCF. The deletion operations can also be performed in parallel by using multi-threading.

IV. Analysis and Optimization

A. False Positive Rate

The false positive rate is one of the most important parameters for approximate membership checking. To perform membership checking, unlike DCF which needs to check all the s CFs, the compacted LDCF only needs to check one corresponding leaf CF. In LDCF with l_c levels, the length of the fingerprint is only $f-l_c$. Thus, for each entry, the probability that the stored fingerprint matches a non-existent element is $\frac{1}{2^{f-l_c}}$. Accordingly, we can compute the false positive rate of the compacted LDCF by the Eq. (5).

$$\epsilon_{LDCF_{compact}} = 1 - (1 - \frac{1}{2^{f-l_c}})^{2b} \approx \frac{2b}{2^{f-l_c}} \qquad (5)$$

B. Computation Cost of Membership Checking

To perform a membership testing, we only need to check one CF per level according to the prefix of the element's fingerprint. For example, if the element y has the six-bit fingerprint ξ_y ='001000' with a level 3 LDCF (such an LDCF has a total number of 15 CFs), we need to check the fingerprint of y against only four CFs out of the 15 CFs, including the single CF on level 0, the '0' CF on level 1 (CF_1^0), the '00' CF on level 2 (CF_2^0), and the '001' CF on level 3 (CF_3^1). This reveals a 73.3% reduction of computation cost compared to the existing DCF design, which needs to probe every CF in the data structure before finding the matched fingerprint. More generally, with a level l LDCF, we only need to check

Algorithm 2: Append (*Loc*, l)

newlevel $\leftarrow l+1$;
Loc.0child $\leftarrow$ *CuckooFilter*(m, b, $f-$*newlevel*);
Loc.1child $\leftarrow$ *CuckooFilter*(m, b, $f-$*newlevel*);
curCF $\leftarrow CF$;
Free *Loc.CF*;
return *Loc*.

Algorithm 3: Membership Check (x)

ξ_x = fingerprint(x);
μ_x = hash(x);
$\nu_x = \mu_x \bigoplus$ hash(ξ_x);
for $i = 0$ *to curLDCF.level* **do**
  pre = *getPrefix*(ξ_x, i);
  curCF = *curLDCF.getCF*(*i, pre*);
  ξ_x = *cutPrefix*(ξ_x, *pre*);
  if *curCF.B*(μ_x) *or curCF.B*(ν_x) *contains* ξ_x **then**
    return true.
return false.

l CFs in the worst case. Table III shows the reduction of computation cost by membership testing using LDCF versus that of previous DCF, where l ranges from one to five. It is not difficult to see that LDCF significantly reduces the time for membership testing to a logarithmic scale. Accordingly, LDCF also greatly reduces the time for element insertion and deletion. This is because when inserting an element into a DCF, DCF needs to perform membership testing first in the current structure to avoid duplication.

C. Space Efficiency

As aforementioned, LDCF can save one bit per entry on level 1, two bits on level 2, ..., i bits on level i, and so on. Thus, in LDCF of which all the leaf CFs are on level l, the total memory cost is $2^l * (f - l) * c$ bits, where f is the length of a fingerprint and c is the number of entries in a CF. We consider the previous DCF design with the same capacity. A DCF with a total number of 2^l CFs has memory cost of $f * (2^l) * c$ bits. The ratio of the space cost of LDCF to that of DCF is less than $\frac{f-l}{f}$. This reveals that LDCF has better space efficiency than DCF with the same capacity.

Let us now consider the space utilization, *i.e.*, the ratio of the number of the stored fingerprints and the capacity. We insert new items in a CF building block until it reaches the maximum load factor (*e.g.*, 95% [17]). When the number of the fingerprints stored in a CF exceeds the maximum load factor, we regard the CF as full and move all the stored fingerprints to two new children CFs. In this case, the capacity of LDCF becomes higher but no more than twice. Accordingly, the space utilization decreases, but it is no less than 47.5%. For dynamic datasets which have an expected size of roughly N, we can optimize the capacity of a single CF building block so that the expected space utilization is close to 95%. We will discuss the details in Section IV-D.

The deletion operation will also decrease the space utilization of LDCF. Unlike the DCF design, if we delete a fingerprint from an entry, the entry can still be utilized for the next possible insertion in LDCF. Thus, for many applications in which insertions happen much more frequently than deletions, we have no need to shrink the space of LDCF. Otherwise, we can perform a *Compact* operation which is the opposite to the append operation. Specifically, we merge two children CFs to a parent CF, if for every bucket address the sum of the stored fingerprint number in the two children CFs is not larger than b. The compact operation can be performed by moving each fingerprint from the children CF to the parent CF according to the bucket address, and adding the corresponding first bit, '0' or '1'.

Algorithm 4: Delete (x)

```
ξ_x = fingerprint(x);
μ_x = hash(x);
ν_x = μ_x ⊕ hash(ξ_x);
for i = 0 to curLDCF.level do
    pre = getPrefix(ξ_x, i);
    curCF = curLDCF.getCF(i, pre);
    ξ_x = cutPrefix(ξ_x, pre);
    if curCF.B(μ_x) or curCF.B(ν_x) contains ξ_x then
        remove ξ_x;
        return true.
return false.
```

TABLE III: The reduction rate of computation cost of membership testing by using LDCF

levels	l = 1	l = 2	l = 3	l = 4	l = 5
reduction rate	33.3%	83.3%	93.3%	96.8%	98.4%

D. Parameter Setting and Space Optimization

We discuss how to set the parameters to optimize the memory usage of the compacted LDCF design. Although in many real applications, we cannot estimate the upper bound of the size of the set in advance, we assume that we can obtain a roughly expected value N of the size of the set in most cases according to historical statistics (*e.g.*, the system logs). We want the false positive rate to be lower than a given threshold ϵ if the size of the set is lower than N. Under these assumptions, we can optimize the most important parameters in our LDCF design including the number of buckets t, the number of entries in each bucket b, the expected number of levels l, and the fingerprint length f.

According to Eq. (5), with a given false positive rate ϵ, we can compute the fingerprint length of LDCF by,

$$f_c = \lceil \log_2(\frac{2b}{\epsilon}) + l \rceil \tag{6}$$

From the above equation, to ensure the actual false positive rate to be lower than the fixed value ϵ, the value of the number of entries b and the fingerprint length f should be well designed. If we set a larger number of entries b, we should set a larger fingerprint length f.

We now discuss the setting of the number of buckets t for each CF block. As we have discussed in Section IV-A, LDCF with several levels can achieve better space efficiency compared to DCF. However, having more levels will also lead to a higher false positive rate. On the other hand, we also want the load factor of LDCF to be high. On the l-th level of LDCF, which contains 2^l CFs, if there are N_0 elements inserted, the load factor can be computed by $\alpha = N_0/(2^l \times t \times b)$. Thus, with the expected size of the set N and a given expected maximum load factor α, we can obtain $2^l \times t \times b = N/\alpha$. Thus, the number of buckets t can be computed by,

$$t = \frac{N}{2^l \times b \times \alpha} \tag{7}$$

In real applications, the set is always dynamic, and the peak size of the set will not always be N. To improve the space utilization, we have no need to set N as the initial capacity of LDCF like the non-extendable approximate set representation structures. Next, we consider some common data distributions including the Zipf distribution, the normal distribution, and the uniform distribution for the set. For a dynamic set S whose

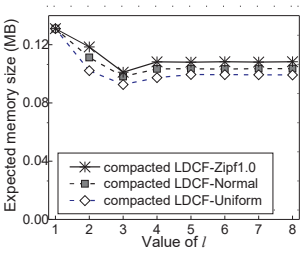

Fig. 4: The expected memory size under different distributions

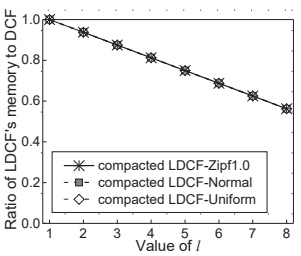

Fig. 5: The ratio of the optimal expected memory size of LDCF to DCF

size follows a certain distribution, we compute the expected space of LDCF. We use p_j to denote the probability that S contains j elements, where $1 \leq j \leq N$ and $\sum_{j=1}^{N} p_j = 1$. The capacity of a single CF is $c = t \times b \times \alpha$. Thus, the actual number of levels i for obtaining all the j elements satisfies $2^{i-1} \times c < j \leq 2^i \times c$. That is to say, the probability that LDCF needs i levels can be computed by $r_i = \sum_{j=c\times 2^{i-1}+1}^{c\times 2^i} p_j$, where $1 \leq i \leq l$. The total number of bits used in LDCF with i levels is $2^i \times t \times b \times (f - i)$. Thus, for such a dynamic set S, the expected number of bits of the compacted LDCF can be computed by,

$$\begin{aligned} E_c(S) &= r_l \times 2^i \times t \times b \times (f_c - i) \\ &= \sum_{j=c\times 2^{i-1}+1}^{c\times 2^i} p_j \times 2^i \times t \times b \times \lceil \log_2(\frac{2b}{\epsilon}) + l - i \rceil \end{aligned} \quad (8)$$

Combining Eqs. (6) and (8), with a given false positive rate ϵ and the load factor α, the optimization problem is equivalent to the linear programming problem for minimizing the value of $E(S)$. With the above equations, we can obtain all the parameters for building a space optimized LDCF.

Figure 4 shows the expected memory size of LDCF with different data distributions (*i.e.*, Zipf, normal, and uniform distribution) of the dynamic set. In this experiment, we set the false positive rate ϵ to 5×10^{-3}. The size of the set is upper bounded by 10,000. The load factor α is set to 90%. Figure 4 shows little variation with different distributions.

We use the same parameter settings in the DCF design, and calculate the optimal memory size of DCF with the same number of building blocks (*i.e.*, the value of the blocks varying from one to 255). Figure 5 shows the ratio of the optimal expected memory size of LDCF to that of DCF. We can see that with a larger number of levels, LDCF reduces more memory space compared to DCF. When the number of levels reaches eight, the compacted LDCF reduces the memory size to 59% of the memory used by DCF.

E. Reliable Deletion and Duplicated Elements Insertion

Deleting elements from a compacted LDCF will not introduce the multiple value problem which happens in the DBF [12]. The multiple value problem happens when a fingerprint of a certain element x can be found in more than one CF building block. Thus, to perform deletion for x, it is critical to determine which fingerprint should be removed. The problem can be easily solved in LDCF by removing any one of the matching fingerprints found. In a compacted LDCF, if two elements share the same fingerprint (*e.g.*, $\xi_x = \xi_y = 010010$), they will be inserted into the same CF building block as they have the same prefix (*e.g.*, the '010' CF). Thus, deletion in LDCF is the same as that in a single CF structure. Consider the following two cases: 1) we have insert two different elements x and y, which have the same fingerprint (*i.e.*, $\xi_x = \xi_y$) and either of them is stored in an entry of the two candidate bucket addresses of x's (*i.e.*, the addresses $h_1(x)$ and $h_1(x) \oplus hash(\xi_x)$); or 2) we have inserted the same element x twice. In the first case, when we perform deletion for x, we can find two matched fingerprints. No matter which fingerprint we choose for deletion, the membership testing for y will always find the remaining fingerprint, and then return the correct result (we can easily prove that y's two candidate bucket addresses are also the addresses $h_1(x)$ and $h_1(x) \oplus hash(\xi_x)$). The second case is similar. The deletion of one copy of the fingerprint of x will not affect the membership testing results for the other copy of x. Therefore, LDCF can support reliable deletion.

We now discuss the insertion of duplicated elements in LDCF in more detail. In the previous DCF design, duplicates can either be allowed to be inserted multiple times, or be filtered. The two strategies are different in the space overhead and the false negative rate caused by the deletion. That is to say, DCF can store the same fingerprint in multiple entries with the same address. However, in LDCF, the situation is totally different. We consider inserting an element x multiple times. On each level, x can only be inserted into the corresponding one CF, and be hashed to the fixed two buckets. If we allow x to be inserted into this CF more than $2b$ times, the CF will be regarded as full. Any other element hashed to the same buckets can no longer be inserted in, since no fingerprints can be kicked out from these two buckets. In the compacted LDCF design, the same element can be inserted at most $2b$ times. According to the above analysis, we should limit the maximum number of times that the same element is inserted into LDCF. Specifically, each time an element needs to be inserted, LDCF performs a membership testing to find whether the element is already in LDCF and how many duplicates are stored. If the number of duplicates is higher than a threshold (*e.g.*, b), we finish the processing without inserting any fingerprints. In the real applications that we consider, the probability that an element is repeatedly inserted and deleted multiple times is low, thus keeping b replicas in LDCF is enough for membership checking. If it is necessary, we can store the duplicated element and its count to a separated structure for more reliable and accurate insertion, deletion, and membership checking.

V. Evaluation

We have implemented the LDCF toolkit and made the source code publicly available[1]. In this section, we evaluate the

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

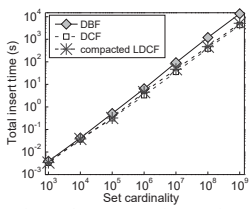

Fig. 6: Total insertion time

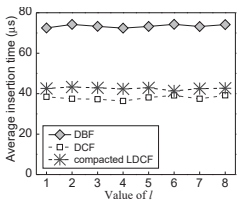

Fig. 7: Average insertion time

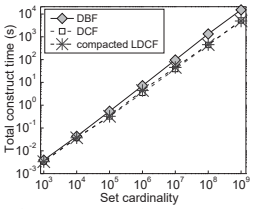

Fig. 8: Total construction time (i:d=10:1)

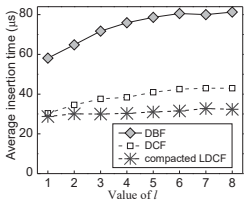

Fig. 9: Average construction time (i:d=10:1)

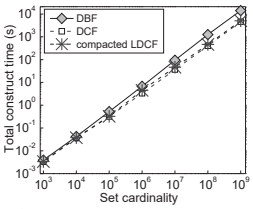

Fig. 10: Total construction time (i:d=50:1)

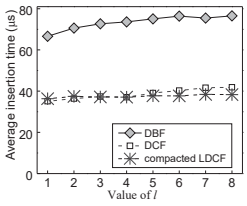

Fig. 11: Average construction time (i:d=50:1)

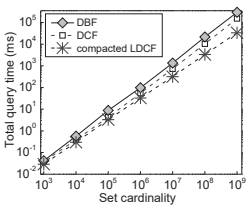

Fig. 12: Total membership query time

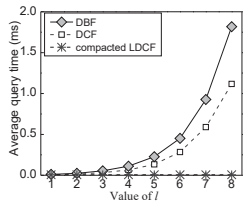

Fig. 13: Average membership query time

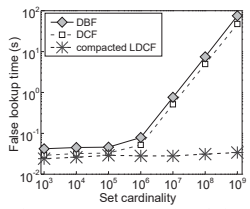

Fig. 14: Total false lookup time

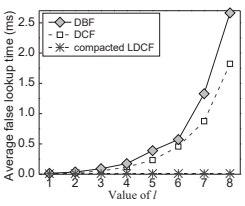

Fig. 15: Average false lookup time

performance of this design by comparing the memory space, insertion, and membership testing speeds with those of the state-of-the-art schemes including DCF [11] and DBF [12]. We use large-scale real world datasets and two typical real applications for evaluation. We run the experiments on a machine equipped with a 16-core 2.4GHz Xeon CPU, 64 GB RAM, and 1TB HDD.

A. Experiment Setup

In our LDCF and DCF implementation, we use SHA1 [27] to generate hash values. We hash each element to a 64-bit hash value. We use the highest f-bit hash value as the fingerprint and the second m-bit value as the bucket address. In the DBF implementation, we use the same method to generate the two hash values of $h_1(x)$ and $h_2(x)$. We further compute the remaining $(k-2)$ hash values of DBF by using Eq. (9). The computation cost for hashing in the implementations of these three schemes are nearly the same. Thus, we omit the differences of hash computation in the evaluation. We set the false positive rates of all the data structures to a fixed value of 1×10^{-3} by default.

$$h_{i+2}(x) = h_i(x) + ih_{i+1}(x) + i^2 (i = 0, 1, 2, ..., k-2) \quad (9)$$

For the basic performance evaluation, we use two datasets. First, we follow the previous work [17], [20] and generate ten billion 64-bit integers as items from random number generators. Second, we collect a dataset containing 30 million certificates [20]. We set the expected value of the set cardinality to ten billion and 30 million separately, set the expected false positive rate to 0.1%, and then compute the optimal parameters as we have discussed in Section IV-D. We set the same capacity of a single CF building block and a single CBF block in DCF and DBF accordingly. Then we construct LDCF, DBF, and DCF for these two datasets separately. We evaluate the false positive rate, the space efficiency, and the construction time of these data structures carefully.

We further conduct experiments to show the performance variation when the data set dynamically changes. In the experiment, we keep inserting new items, and also randomly deleting a fraction of inserted items to an initial empty LDCF (or DBF, DCF). We record the total construction time when the number of inserted items or the value of l in LDCF (or s in DBF, DCF) reaches a certain value. With fixed different set cardinality or levels, we compare the processing performance of insertion, membership checking, and deletion operations of all the three structures. Since the elements are randomly generated, we filter the duplicated elements before performing the insert operation in all the data structures.

Then, we use two real applications for evaluating the performance of these schemes: k-mers counting [3] and cloud storage [10]. The k-mers counting application is an important part of DNA sequencing analysis [3]. A k-mer is a k-length subsequence of a given DNA sequence (which consists of k characters from A, T, C, G). For example, a sequence ATGCTAG contains five different 3-mers: ATG, TGC, GCT, CTA, and TAG. Counting the appearances of k-mers and identifying the unique k-mers (which means the k-mer appears only once) in a DNA sequence can help filtering out errors, analysing variations, finding anchors and so on. To achieve this, existing schemes [3] use approximate set representation data structures such as bloom filters to find unique k-mers, and a hash table for counting other k-mers. The total number of k-mers can be estimated by the length of a DNA sequence. However, k-mers have high repetition rate so that it is hard to estimate the number of unique k-mers in advance. Using the total number of sequences as the capacity of an approximate set representation structure is space inefficient. Thus, we need a flexible structure which can extend the capacity on demand. We collect large-scale DNA sequencing data from DDBJ Center [28], which contains 35 to 74 million DNA sequence records in FASTQ formats. FASTQ format is a text-based format which stores the DNA sequence as well as its corresponding quality scores. Table IV summarizes the datasets we use for the experiments.

In cloud storage applications, approximate set representation data structures are widely used to avoid storing duplicate file

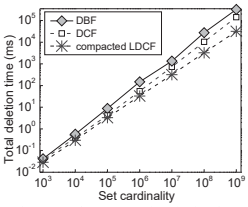

Fig. 16: Total deletion time

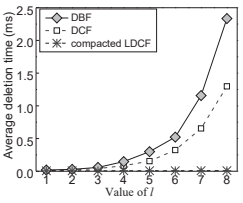

Fig. 17: Average deletion time

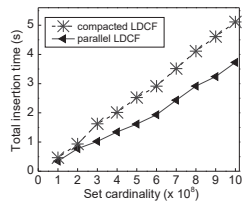

Fig. 18: Insertion time with parallel setting

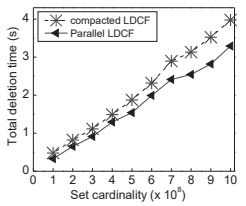

Fig. 19: Deletion time with parallel setting

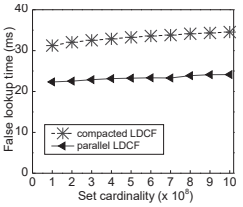

Fig. 20: Lookup time with parallel setting

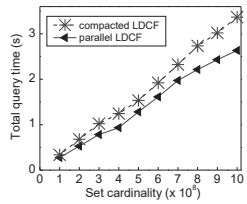

Fig. 21: Total query time with parallel setting

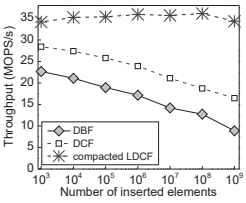

Fig. 22: Throughput in cloud storage application (I9D1L3)

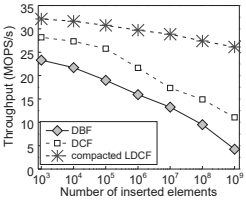

Fig. 23: Throughput in cloud storage application (I3D1L3)

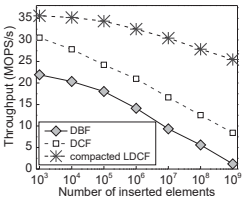

Fig. 24: Throughput in cloud storage application (I3D1L9)

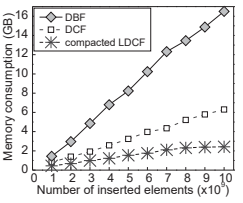

Fig. 25: Memory consumption in cloud storage application

chunks [10], [29]. However, the upper bound of the total number of file chunks cannot be estimated in advance. Thus, the application needs an extendable approximate set representation structure. To examine the performance, we generate three different large-scale traces: 1) I9D1L3, which means the ratio of the new chunk insertion rate, deletion rate, and the look up rate is 9:1:3; 2) I3D1L3 with rates of 3:1:3; and 3) I3D1L9 with rates of 3:1:9. We use these different traces to evaluate the overall throughput of all the schemes.

B. Results

Overall Performance. We construct approximate set representation data structures for the integer and the certificates datasets, and keep randomly picking items (belonging to or not belonging to the set) to examine the false positive rate and the lookup performance of the three data structures. The results in Table V show that LDCF achieves the highest space efficiency, the lowest false positive rate [30], and the highest lookup throughput compared to DBF and DCF.

Element Insertion. We evaluate the performance of element insertion in two aspects. First, we only insert (without deletion) items until the set cardinality reaches a certain value, and test the total insertion time. Additionally, each time when we append a new level l in LDCF (for DBF and DCF, that means when we append the s-th building block, s=2^l), we evaluate the average insertion time.

Figure 6 shows the insertion time of the three data structures with different set cardinality. We set the expected value of levels to four and compute the optimal parameters for each data structure. We can see that the construction time of DCF and the compacted LDCF are nearly the same. We can also see that the reduction rate of the construction time of LDCF to that of DBF increases with the set cardinality. The results show that LDCF significantly reduces the total construction time of DBF by 64% when the cardinality is one billion.

Figure 7 presents the average insertion time of the three data structures with different levels. Since all the three structures perform the insert operation in only one of the building blocks, the insertion time does not depend on the number of levels. The results show that LDCF reduces the average insertion time of DBF by 44%.

Dynamic Set Construction. We evaluate the construction speed of three data structures when facing dynamic data sets. In these experiments, every time when we have inserted 100 items, we randomly choose ten (or two) items from all the inserted items, and perform deletion for them. We record the total construction time when the set cardinality or the value of l (s) reaches a certain value. Figure 8 shows the total construction time of the three data structures with different set cardinality. During construction, the ratio of insertion and deletion is 10:1. We can see that the construction time of DCF and the compacted LDCF are nearly the same. The results show that LDCF significantly reduces the total construction time of DBF by 69% when the cardinality is one billion. Figure 9 presents the average construction time of the three

TABLE IV: DNA sequencing datasets description

Dataset name	Size (GB)	Total sequences (million)
SRX424605	2.8	74
SRX425716	1.9	54
SRX426213	1.8	41
SRX426499	1.7	59
SRX426506	1.3	35

TABLE V: Overall performance comparison

Dataset	metrics	LDCF	DBF	DCF
Integer	bits per item	**8.423**	18.665	13.147
	false positive rate (%)	**0.113**	0.119	0.114
	lookup throughput (MOPS)	**42.3**	5.1	17.4
Certificates	bits per item	**9.413**	19.227	13.674
	false positive rate (%)	**0.072**	0.083	0.075
	lookup throughput (MOPS)	**62.5**	11.3	24.5

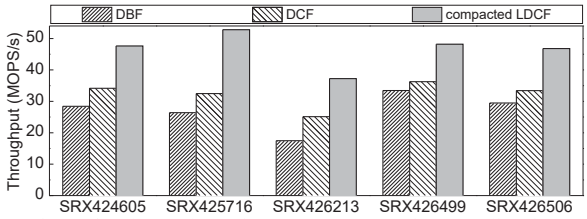

Fig. 26: Throughput in DNA sequencing application

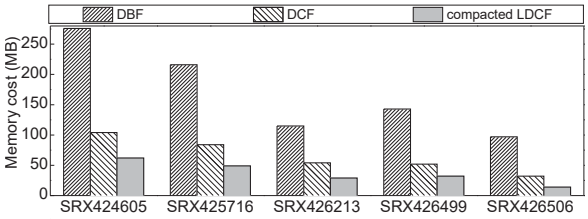

Fig. 27: Memory consumption in DNA sequencing

data structures with different levels. During construction, the ratio of insertion and deletion is 10:1. The results show that LDCF reduces the average construction time of DBF by 61%, and reduces that of DCF by 25% when the level is eight.

Figure 10 shows the total construction time of the three data structures with different set cardinality. During construction, the ratio of insertion and deletion is 50:1. We can see that the construction time of DCF and the compacted LDCF are nearly the same. LDCF significantly reduces the total construction time of DBF by 66% when the cardinality is one billion. Figure 11 presents the average construction time of the three data structures with different levels. During construction, the ratio of insertion and deletion is 50:1. The results show that LDCF reduces the average construction time of DBF by 50%, and reduces that of DCF by 9% when the level is eight.

Membership Query. We evaluate the performance of membership query in two aspects. First, we test all the elements which have been inserted and show the query time. Then, we query a million elements which have never been inserted and present the query time. The results reflect the performance of the membership querying for elements that are not in the set which is the worst case.

Figure 12 shows the total membership query time for inserted elements of the three data structures with different set cardinality. The results show that the compacted LDCF design significantly reduces the total membership query time of DBF by 89% and reduces that of DCF by 80% when the set cardinality is one billion.

Figure 13 presents the average membership query time of the three data structures with different levels. The compacted LDCF reduces the average membership query time of both DBF and DCF by over two orders of magnitudes with $l = 8$.

Figure 14 shows the look up time for elements that are not in the set for the three data structures and different set cardinalities. LDCF greatly reduces the total query time of both DBF and DCF by more than two orders of magnitudes for large cardinalities.

Figure 15 presents the average look up time of the three data structures with different levels. LDCF greatly reduces the average query time of DBF by 97%, and reduces that of DCF by 95% when the number of levels is eight.

Element Deletion. We evaluate the performance of deletion by deleting all the elements which have been inserted into all the three structures, and report the total processing time.

Figure 16 shows the deletion time of the three data structures with different set sizes. During deletion, DCF will trigger the compact operation, which leads to a longer processing time. The compacted LDCF greatly reduces the total deletion time of DBF by 88% and reduces that of DCF by 78%.

Figure 17 plots the average deletion time of the three data structures with different levels. The compacted LDCF reduces the average deletion time of both DBF and DCF by more than two orders of magnitudes when the number of levels is eight.

From Figs. 6-17, we can see that DCF is efficient only when facing continuous insertion workloads, while LDCF is more efficient for membership testing and deletions on large-scale dynamic datasets.

Performance of Parallel Operations. We evaluate the performance of the compacted LDCF by running four threads to perform operations in parallel.

Figure 18 shows the parallel LDCF greatly reduces the total insertion time of LDCF by 28% and reduces that of the compacted LDCF without parallelism by 21%.

Figure 19 shows that the parallel LDCF reduces the total deletion time of LDCF by 62% and reduces that of compacted LDCF without parallelism by 17%.

Figure 20 shows that the parallel LDCF significantly reduces the false lookup time of LDCF by 87% and reduces that of the compacted LDCF without parallelism by 9%.

Figure 21 presents the membership query time with different set sizes. The results show that the parallel LDCF significantly reduces the membership query time of LDCF by 62% and reduces that of compacted LDCF without parallelism by 11%.

Applications. We evaluate the performance of our design with the k-mer counting and the cloud storage application. We mainly examine the throughput and the memory consumption of the applications using different structures.

Figures 22-24 show the throughput of the cloud storage application with three different workloads. The results show that with any of the three workloads, the application with LDCF achieves the highest throughput. Figure 22 shows that the application with the compacted LDCF increases the throughput of DCF by 82% when the ratio of insertion, deletion, and membership testing is 9:1:3. Figure 23 shows that the application with compacted LDCF improves the throughput of DCF by 1.13× when the ratio is 3:1:3. Figure 24 shows that the application with compacted LDCF improves the throughput of DCF by 2.43× when the ratio is 3:1:9.

Figure 25 shows the memory consumption of the cloud storage application. We compute the memory cost every time when we insert 100 million new items. The result shows that, the compacted LDCF significantly reduces the memory cost of DBF by 85% and reduces that of DCF by 61%. The significant reduction of the memory cost in LDCF is mainly because 1)

the average value of levels is high, and the stored fingerprints are much shorter than that stored in DCF; 2) LDCF achieves higher average load factor of each CF building blocks (94%) than that of DCF (87%).

Figure 26 shows the throughput of the k-mer counting application by using different DNA sequencing datasets. In this application, we set k = 12. Each time before we insert a new k-mer, we look up whether it has been once inserted to identify unique k-mers. The results show that the k-mer counting application with the compacted LDCF outperforms that with the other data structures in all the five DNA sequencing datasets. On average, for the k-mer counting application the compacted LDCF structure improves the throughput of the application with DBF by 75% and improves the throughput with DCF by 37%.

Figure 27 shows the memory consumption of the k-mer counting application by using different DNA sequencing datasets. The result shows that the memory cost of the compacted LDCF structure is much less than those of the other data structures. On average, the compacted LDCF structure reduces the the memory cost of DBF by 79% and reduces that of DCF by 43%.

VI. Conclusion

In this paper, we propose LDCF, a novel efficient approximate set representation structure for large-scale dynamic data sets. LDCF reduces the membership checking time to $O(log(N))$ and the memory space significantly. We further design a compacted LDCF, which reduces the inserting and membership testing time to $O(1)$ with the same scale of memory cost. We use large-scale datasets and real world applications for evaluating the performance of the LDCF design. Comprehensive experiment results show that compared to existing state-of-the-art designs, LDCF significantly reduces the membership checking time and the memory cost for large-scale datasets.

VII. Acknowledgements

This research is supported in part by the National Key Research and Development Program of China under grant No.2018YFB1004602, NSFC under grants Nos. 61972446, 61422202. Pedro's work is supported by the ACHILLES project PID2019-104207RB-I00 and the Go2Edge network RED2018-102585-T funded by the Spanish Ministry of Economy and Competitivity and by the Madrid Community research project TAPIR-CM grant no. P2018/TCS-4496.

References

[1] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu, "Elasticbf: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores," in *Proceedings of USENIX ATC*, 2019, pp. 739–752.

[2] R. Li, Z. Xu, W. Kang, K. C. Yow, and C.-Z. Xu, "Efficient multi-keyword ranked query over encrypted data in cloud computing," *Future Generation Computer Systems*, vol. 30, no. 1, pp. 179–190, 2014.

[3] R. S. Roy, D. Bhattacharya, and A. Schliep, "Turtle: Identifying frequent *k*-mers with cache-efficient algorithms," *Bioinformatics*, vol. 30, no. 14, pp. 1950–1957, 2014.

[4] H. Chen, H. Jin, J. Wang, L. Chen, Y. Liu, and L. M. Ni, "Efficient multi-keyword search over p2p web," in *Proceedings of WWW*, 2008.

[5] A. P. Batson, "The organization of symbol tables," *Communications of the ACM*, vol. 8, no. 2, pp. 111–112, 1965.

[6] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[7] L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000.

[8] A. Z. Broder and M. Mitzenmacher, "Survey: Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2003.

[9] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy, "Storm@twitter," in *Proceedings of SIGMOD*, 2014.

[10] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhang, and Y. Tan, "Design tradeoffs for data deduplication performance in backup workloads," in *Proceedings of FAST*, 2015.

[11] H. Chen, L. Liao, H. Jin, and J. Wu, "The dynamic cuckoo filter," in *Proceedings of ICNP*, 2017.

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

[13] P. S. Almeida, C. Baquero, N. M. Preguiça, and D. Hutchison, "Scalable bloom filters," *Information Processing Letters*, vol. 101, no. 6, pp. 255–261, 2007.

[14] T. Yang, A. X. Liu, M. Shahzad, D. Yang, Q. Fu, G. Xie, and X. Li, "A shifting framework for set queries," *IEEE/ACM Transactions on Networking*, vol. 25, no. 5, pp. 3116–3131, 2017.

[15] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo, "Surf: Practical range query filtering with fast succinct tries," in *Proceedings of SIGMOD*, 2018.

[16] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[17] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of CoNEXT*, 2014.

[18] H. Lang, T. Neumann, A. Kemper, and P. A. Boncz, "Performance-optimal filtering: Bloom overtakes cuckoo at high-throughput," *Proceedings of the VLDB Endowment*, vol. 12, no. 5, pp. 502–515, 2019.

[19] A. Breslow and N. Jayasena, "Morton filters: Faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity," *Proceedings of the VLDB Endowment*, vol. 11, no. 9, pp. 1041–1055, 2018.

[20] M. Wang, M. Zhou, S. Shi, and C. Qian, "Vacuum filters: More space-efficient and faster replacement for bloom and cuckoo filters," *Proceedings of the VLDB Endowment*, vol. 13, no. 2, pp. 197–210, 2019.

[21] Y. Peng, J. Guo, F. Li, W. Qian, and A. Zhou, "Persistent bloom filter: Membership testing for the entire history," in *Proceedings of SIGMOD*, 2018.

[22] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok, "Don't thrash: How to cache your hash on flash," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1627–1637, 2012.

[23] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, "A general-purpose counting filter: Making every bit count," in *Proceedings of SIGMOD*, 2017.

[24] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[25] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig, "Cold filter: A meta-framework for faster and more accurate stream processing," in *Proceedings of SIGMOD*, 2018.

[26] F. Wang, H. Chen, L. Liao, F. Zhang, and H. Jin, "The power of better choice: Reducing relocations in cuckoo filter," in *Proceedings of ICDCS*, 2019.

[27] H. Chen, H. Jin, and 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, 2016.

[28] Fragaria vesca genome sequence dataset, https://www.ddbj.nig.ac.jp/dra/index-e.html, 2020.

[29] T. Ying, H. Chen, and H. Jin, "Pensieve: Skewness-aware version switching for efficient graph processing," in *Proceedings of SIGMOD*, 2020.

[30] B. Peng, Z. Lü, and T. C. E. Cheng, "A tabu search/path relinking algorithm to solve the job shop scheduling problem," *Computers & Operations Research*, vol. 53, pp. 154–164, 2015.