The Dynamic Cuckoo Filter

Hanhua Chen, *Member, IEEE,* Liangyi Liao, Hai Jin, *Senior Member, IEEE,* and Jie Wu, *Fellow, IEEE*

Abstract—The emergence of large-scale dynamic sets in real applications creates stringent requirements for approximate set representation structures: 1) the capacity of the set representation structures should support flexibly extending or reducing to cope with dynamic change of set size; 2) the set representation structures should support reliable delete operation. Existing techniques for approximate set representation, e.g., the *cuckoo filter*, the *Bloom filter* and its variants cannot meet both the requirements of a dynamic set. To solve this problem, in this paper we propose the *dynamic cuckoo filter* (DCF) to support reliable delete operation and elastic capacity for dynamic set representation and membership testing. Two factors contribute to the efficiency of the DCF design. First, the data structure of a DCF is extendable, which allows the representation of a dynamic set space efficient. Second, a DCF utilizes a monopolistic fingerprint for representing an item and guarantees reliable delete operation. Experiment results show that when compared to the existing state-of-the-art designs, DCF achieves a 75% reduction in memory cost, 50% improvement in construction speed, and 80% improvement in speed of membership query. We implement a prototype file backup system and use DCF for data deduplication. Comprehensive experiment results demonstrate the efficiency of our DCF design compared to existing schemes.

Index Terms—Dynamic set representation; set membership testing; cuckoo filter

1 Introduction

Set representation and membership testing are two core problems of many computer applications. Set representation means organizing the information of the elements of a set using a data structure, which makes the information of the set elements operable by corresponding methods. Set membership testing means checking and determining whether an element with a given attribute value belongs to a given set with a given set representation structure.

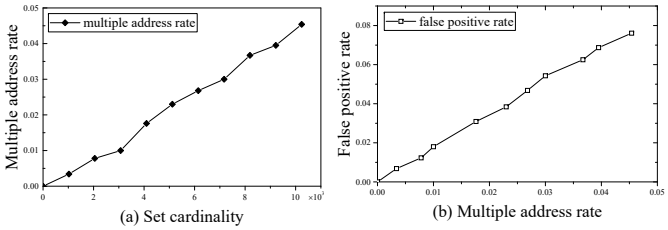
A naive set membership testing data structure is hash coding [2]. In conventional hash coding, a hash area is organized into an array of cells. An iterative pseudorandom computational process $h(\cdot)$, also called a hash function, is used to generate hash addresses of empty cells from the given set of elements $S = \{x_1, x_2, \ldots, x_n\}$. The raw data of the elements are then stored into the empty cells. If we need to test whether or not an item y is an element of S, we first obtain $h(y)$, the hash address of y, and then we check y against the raw data stored in the $h(y)^{th}$ cell. If matched, we determine that y is an element of S. Otherwise, y does not belong to S. The traditional hash coding scheme does not have false positives based on raw data matching. However, such a scheme is costly in both space for storing the raw data and computation for membership testing with raw data matching.

It is not difficult to see that if we allow an error with a low probability in set membership testing, it is not necessary to store the complete raw data in the hash space. Instead, Boolean labels or fingerprints of raw data can be utilized to replace the raw data to save the space. The schemes may bring false positives because different items may happen to collide in the same hash addresses or have the same fingerprints. Such an approximate set membership testing technique is used in many real-world application systems, e.g., Web caches [10], P2P applications [19], routers [20] and file backup systems [11], etc. Approximate set representation structures have attracted much attention in the research community. Most of the existing work focuses on the tradeoff between cost and error rate. Commonly, a smaller number of bits used by the labels or fingerprints leads to a higher false positive rate. In order to balance efficiency and accuracy, several hash coding techniques have been introduced into set representation data structures [3, 9].

A *standard Bloom filter* (SBF) [3] is the most popular approximate set representation structure [25]. An SBF is essentially an array of m bits all initially set to "0". It maps every item of the set $S = \{x_1, x_2, \ldots, x_n\}$ into the bit address space $[0, m-1]$ using a number of k uniform and independent hash functions $h_1(\cdot), \ldots, h_k(\cdot)$. For the item x belonging to S, the bits with the hash addresses $h_i(x)$ are all set to "1" for $1 \leq i \leq k$. When we decide whether the item y belongs to S or not, we first compute $h_i(y)$ for $1 \leq i \leq k$. If all the corresponding $h_i(y)^{th}$ bits are "1", y belongs to S with high probability; otherwise, y is definitely not a member of S. An SBF does not support delete operation because multiple items in S may share any of the hash addresses $h_i(x)$ ($1 \leq i \leq k$). Deleting an item x by flipping all the bits with hash addresses $h_i(x)$ ($1 \leq i \leq k$) from "1" to "0" may lead to the problem of false negative.

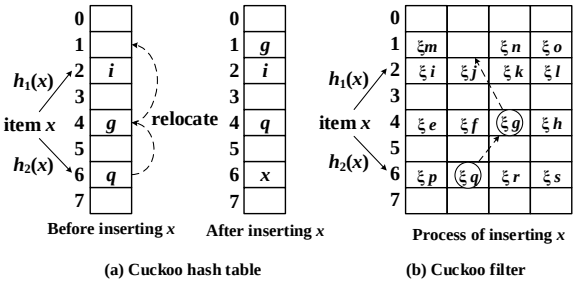
In practice, real applications commonly involve a highly dynamic set with members joining and leaving dynamically and with an unpredictable size of the set [23]. For example, in popular stream applications [22], an unbounded sequence of data tuples come with the data flow. This requires a set representation data structure to have the ability to cope with sets with a changing cardinality. furthermore, in a Web cache proxy [4], the cached set of Web pages is frequently updated according to the cache replacement strategies. This requires

- *H. Chen, L. Liao and H. Jin are with Big Data Technology and System Lab, 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. E-mail:{chen, liaoliangyi, hjin}@hust.edu.cn.*
- *J. Wu is with the Center for Networked Computing, Temple University, Philadelphia, PA 19122, U USA. E-mail:jiewu@temple.edu.*



(a) Set cardinality (b) Multiple address rate

Fig. 1: Multiple address problem in DBF. (The number of hash functions is set to seven. The number of CBFs varies from one to 10. Each CBF has 40,960 bits and can hold 1024 items. Totally 10,240 items are inserted in the experiment.)



(a) Cuckoo hash table (b) Cuckoo filter

Fig. 2: Cuckoo hash table and cuckoo filter

a set representation structure to support delete operation. The above features of dynamic sets in real applications bring more stringent requirements for approximate set representation structures: 1) The capacities of the data structure should support flexibly extending or reducing. 2) The data structure should support reliable delete operation, i.e., the deletion of any element of the set will not affect the accuracy of the membership testing of other elements in the set.

Counting Bloom filter (CBF) [10] replaces each bit of an SBF with a counter of s bits to support item deletion. However, the space cost of a CBF is s times larger than that of the SBF. Inserting or deleting an item x corresponds to increasing or decreasing the value of the $h_i(x)^{th}$ counters by one for $1 \leq i \leq k$.

Another data structure which can support delete operation is the *cuckoo filter* recently proposed by Fan et al. [9]. A CF consists of an array of l buckets. An item x_i of the set $S = \{x_1, x_2, \ldots, x_n\}$ is represented by its fingerprint ξ_{x_i} and stored in one of the bucket. Essentially, a fingerprint is an f-bit fixed-size bit vector derived from the item using a hash function. Each item x_i monopolizes a fingerprint ξ_x and thus removing the fingerprint ξ_{x_i} will not influence the membership testing of any other elements x_j $(j \neq i)$ in S. For the inserting operation, each item can choose either one from the two buckets with the addresses computed by the two independent hash functions $h_1(\cdot)$ and $h_2(\cdot)$. Each bucket in the array has a number of b storage units, also called entries, and each entry can host one fingerprint. If we need to check whether or not an item y is contained in S, we compare all the fingerprints stored in the $h_1(y)^{th}$ and the $h_2(y)^{th}$ buckets with the fingerprint ξ_y. If matched, we believe y belongs to S with high probability. Otherwise, y is definitely not a member of S. Removing an item x from S corresponds to the deletion of the matched fingerprints ξ_x in the $h_1(x)^{th}$ or the $h_2(x)^{th}$ bucket.

To cope with the issue of set extension, Guo et al. [12] propose the *dynamic Bloom filter* (DBF). A DBF consists of a linked list of s homogeneous CBFs. Whenever the current DBF structure is full, it extends the capacity by appending a new building block of CBF. Inserting an element x is performed by increasing the $h_i(x)^{th}$ counter by one for $1 \leq i \leq k$ in the current active CBF (which is not full) at the end of the link. Querying an item y needs to probe every CBF until one is found with all the $h_i(y)^{th}$ bits $(1 \leq i \leq k)$ being nonzero digits.

The DBF, however, does not support reliable deletion [12], because the DBF is not able to distinguish which CBF stores the bit information of the item x when the $h_i(x)^{th}$ $(1 \leq i \leq k)$ counters are found nonzero in multiple CBFs. Such multiple address is a common case when the number of CBFs increases in the DBF. Thus, the DBF gives up the delete operation and leaves the redundant bits information remained in the data structure. This will result in much more serious false positives of the DBF. Figure 1 analyzes the multiple address problem of the DBF in greater detail using experiments. The result in Fig.1(a) shows that the increment of the number of items inserted in the DBF leads to the growth of multiple address rate (the probability of the occurrence of multiple address when deleting). The growth of the multiple address rate deteriorates the false positives of the DBF (Fig. 1(b)). Therefore, the DBF does not support reliable delete operation for dynamic sets.

In this work, we propose the *dynamic cuckoo filter* (DCF), which successfully meets both of the two requirements for approximate dynamic set representation and membership testing for large-scale data collections. First, DCF utilizes a monopolistic fingerprint for representing an item and thus enables reliable delete operation. Second, DCF exploits a novel extendable and compressible structure to make the data structure space efficient for a dynamic set. We conduct comprehensive experiments using real world datasets as well as implement a prototype file backup system to evaluate the performance of our DCF design. The results show that the proposed DCF reduces the required memory space of the DBF by 75% as well as improve the speeds of inserting and membership testing by 50% and 80%, respectively. The implementation of a prototype system demonstrates the superiority of our DCF design in applying to data deduplication in file backup systems.

The rest of this paper is organized as follows. Section II introduces the background and the related work. Section III presents the DCF design and its operations. Section IV analyzes the false positive rate and examine how the delete operation affects the false positive rate of a DCF. Section V discusses the optimization of the configuration of this design. Section VI evaluates the performance of the DCF using experiment and prototype system implementation. Section VII concludes this work.

2 Background and Related Work

In this section, we briefly introduce the background and the related work of this research. We mainly introduce the recently proposed CF and DBF designs which are mostly similar to our design.

2.1 Cuckoo Hash Table

A cuckoo hash table [17] consists of an array of l buckets. Each bucket is a basic storage unit for storing an item. Each item has two candidate buckets whose addresses are computed by two independent hash functions $h_1(\cdot)$ and $h_2(\cdot)$. Inserting an item x is performed following the steps below: 1) The cuckoo hash table computes the candidate buckets addresses $h_1(x)$ and $h_2(x)$; 2) If either the $h_1(x)^{th}$ or the $h_2(x)^{th}$ bucket is empty, store x into any of the empty buckets; 3) If both the buckets are occupied, the cuckoo hash table randomly selects an occupied bucket and kicks out the item stored in the selected bucket. The item that is kicked out, also called the victim, relocates itself to its alternative bucket; 4) If the alternative bucket is empty, then store x in the empty alternative bucket and the insert process terminates. Otherwise, repeat step 3) until all the items find their own place or the number of relocations reaches a specified upper bound.

Figure 2(a) illustrates an example of inserting an item x into a cuckoo hash table with eight buckets. After hashing mapping, x can be placed in the 2^{nd} or the 6^{th} bucket and neither of the two buckets is empty. Therefore, the algorithm randomly picks a bucket, e.g., the 6^{th} bucket, kicks out the existing item q and inserts x into the 6^{th} bucket. The victim item q relocates itself to the alternative 4^{th} bucket by kicking out the existing item g. After the item g is inserted into the empty alternative 1^{st} bucket, all the items find their own buckets and the insert operation terminates. To control the cost of relocations, the cuckoo hash table specifies an upper bound of the number of relocations, denoted as MNK. When reaching the upper bound, the cuckoo hash table is regarded as a full cuckoo hash table. To reduce the cost for possible frequent relocations, Dietzfelbinger et al. [5] extend the cuckoo hash table by allowing multiple items stored in a single bucket.

2.2 Cuckoo Filter

By replacing the original element x with a fingerprint of the element (denoted as ξ_x) in the cuckoo hash table, Fan et al. recently proposed a new approximate set membership testing data structure, called *cuckoo filter* (CF) [9]. Because the fingerprint ξ_x takes much fewer bits than x itself, CF is space efficient. Formally, a CF leverages an array of l buckets, and each bucket has a number of b entries. Each entry has a fixed size of f bits, where f equals to the size of a fingerprint. Hence, the fingerprint of an item can be stored in a single entry. Storing a fingerprint of an item other than the raw data raises challenges during relocation: it is difficult for a victim element to find the alternative hosting bucket. To solve the problem, CF leverages the partial-key cuckoo hashing, which computes the alternative bucket address by performing the XOR operation on the current bucket address and the evicted fingerprint. The two candidate bucket addresses for x are computed as below.

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

where ξ_x represents x's fingerprint.

Based on such a design, the insert operation of a CF differs from that of the cuckoo hash table in the calculation of the hashing address of the alternative buckets for possible victim elements. When querying an item y in set S, one first needs to compute the fingerprint of y, denoted as ξ_y, and the addresses of the candidate buckets for hosting ξ_y, denoted as $h_1(y)$ and $h_2(y)$. The CF then checks ξ_y the fingerprints stored in the $h_1(y)^{th}$ and $h_2(y)^{th}$ buckets. If matched, y is regarded a member of S; otherwise, y does not belong to S. The delete operation simply removes the matched fingerprint.

Figure 2(b) shows an example of a CF with eight buckets ($l = 8$), each with four entries ($b = 4$). When inserting an item x, the CF first calculates the addresses of the candidate buckets and tries to find a spare entry. At the obtained addresses, if there is a bucket with a spare entry, the fingerprint ξ_x of the item x will be stored in the entry. If both of the buckets are full (e.g., the 2^{nd} and the 6^{th} buckets), the CF randomly kicks out a fingerprint in a randomly chosen bucket (e.g., fingerprint ξ_q in the 6^{th} bucket). Then, the victim ξ_q continues to relocate itself to the alternative 4^{th} bucket by kicking out the fingerprint ξ_g. After the victim ξ_g successfully relocates itself to the 1^{st} bucket, all the fingerprints find their own places and the insert operation terminates.

The upper bound of false positive rate can be computed by the equation [9],

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

Compared with an SBF, the greatest advantage of a CF is the support of delete operation. A CF achieves deletion by removing the monopolistic fingerprint for an item x_i. It is clear that removing the fingerprint of an item x_i will not affect the membership testing of any other elements x_j ($j \neq i$) in the CF. Although a CF, in some degree, satisfies the deletion requirement of the representing dynamic sets, it lacks the ability to flexibly extend its capacity on demand.

2.3 Dynamic Bloom Filter

The most similar work with our design is the *dynamic bloom filter* [12]. A DBF consists of a linked list of s homogeneous CBFs and extends its capacity by appending new building blocks of CBFs. The capacity of a CBF c denotes the number of inserted items when the false positive rate of the CBF reaches the limit of the allowed false positive rate ϵ_{CBF}. Formally, given the number of inserted items n_r, the CBF is called an active CBF when $n_r < c$. When there are no active CBFs in the current DBF, the DBF creates a new CBF and appends the new one to the linked list. Inserting a new item x into a DBF first needs to find an active CBF, and then insert x into the active CBF by increasing all the $h_i(x)^{th}$ counters of the CBF by one for $1 \leq i \leq k$. Checking an item y needs to probe every CBFs until finding a CBF with all the $h_i(y)^{th}$ bits ($1 \leq i \leq k$) are nonzero digits.

When deleting an item x, the DBF first needs to determine whether or not the item x exists in the DBF. If only one CBF is found matched, the delete operation will be performed by decreasing all the $h_i(x)^{th}$ counters by one for $1 \leq i \leq k$ in the matched CBF. If more than one

TABLE 1: Notations in DCF

Notations	Description
CF_k	the k^{th} CF in DCF
S	a set of items to be represented
x_i	the i^{th} item in set S
ξ_{x_i}	the fingerprint of the item x_i
s	the number of CFs in DCF
l	the number of buckets in each CF
b	the number of entries in each bucket
μ_{x_i}, ν_{x_i}	two bucket addresses of the item x_i
$B_k(\mu_{x_i}), B_k(\nu_{x_i})$	two candidate buckets of x_i in CF_k
ϵ_{CF}	false positive rate of each CF
ϵ_{DCF}	false positive rate of DCF
c	the capacity of each CF

CBFs are found with matched result, the DBF is not able to decide which CBF contains the item x. The problem is called multiple address [12]. In the presence of the multiple address, the DBF gives up the delete operation to avoid possible false deletion of an item. However, this keeps the redundant information remained in the DBF and leads to the rising of false positive rate. Such a problem becomes even acute when the set cardinality changes frequently and makes the DBF not available for large-scale real world applications.

It is clear that existing work cannot satisfy both the two requirements for dynamic sets. In this work, we propose a novel *dynamic cuckoo filter* design which supports an elastic capacity as well as a reliable delete operation.

3 Dynamic Cuckoo Filter

3.1 Overview

A DCF consists of a number of s linked homogenous CFs $\{CF_1, \ldots, CF_s\}$. Each CF is an array of l buckets $\{B(1), \ldots, B(l)\}$. A fingerprint ξ_i $(1 \leq i \leq n)$ is a hash code with a fixed size of f bits for representing an item x_i $(1 \leq i \leq n)$ in a given set $S = \{x_1, \ldots, x_n\}$. Each bucket $B_k(\mu)$ has a number of b entries. Every entry is a vector with f bits, initially all set to "0". Thus, an entry can store one fingerprint to represent an element of S. Every fingerprint ξ_i has two candidate buckets with the addresses μ and ν. The address is generated by the hash function $\mu = h_1(x_i)$ while the address ν is encoded using a partial-key cuckoo hash $\nu = \mu \oplus h_1(\xi_i)$. The fingerprint ξ_i represents x_i in the k^{th} CF in the DCF and is stored in the bucket $B_k(\mu)$ or $B_k(\nu)$ $(1 \leq k \leq s)$. Querying an item y against S needs to probe every CF in the DCF until a matched result is found. The algorithm first calculates the corresponding fingerprint ξ_y, and the bucket addresses μ, ν for y. If there exists a CF_k $(1 \leq k \leq s)$ which stores ξ_y in either the bucket $B_k(\mu)$ or $B_k(\nu)$, we assume the item y belongs to S. Otherwise, y is definitely not a member of S. Deleting an item x first calculates the candidate addresses μ, ν for x, and then matches the corresponding fingerprint of x against the fingerprints in $B_k(\mu)$ and $B_k(\nu)$ for $1 \leq k \leq s$. If matched, set the entry which holds the matched fingerprint to "0". Inserting an item into DCF simply corresponds to fingerprint insertion in one of the CFs. Table 1 summarizes the notations for the definition of a DCF.

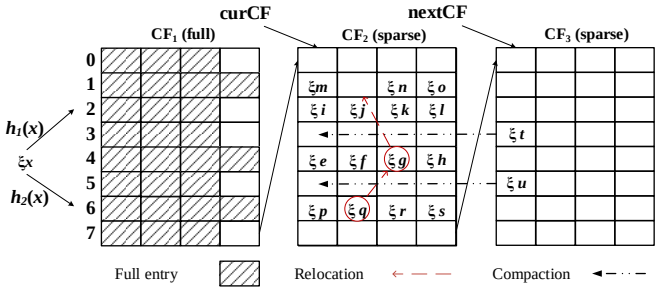


Fig. 3: An example of DCF

It is not difficult to see that compared to a CF, no extra operations are needed for inserting an item into DCF. Therefore, the time complexity of insert operations of a DCF and a CF are both $O(1)$. For a membership query operation, a DCF needs to probe every CF in the worst case. Accordingly, the time complexity of membership query operation of a DCF and a CF are $O(bs)$ and $O(b)$, respectively. Deleting an item needs to find the bucket $B_k(\mu)$ holding a matched fingerprint. Thus, a query operation should be performed ahead of the removal of the matched fingerprints. The time complexity of the delete operations are the same as that of the membership query operations, i.e., $O(bs)$ for a DCF and $O(b)$ for a CF.

Essentially, a DCF extends its capacity by appending new CFs and decreases its capacity by dropping the vacant CFs. Each CF has a counter to record the number of items inserted inside the CF. A CF is full when the value of the counter increases to a predefined capacity c and is empty when the counter decreases to "0". We call a CF active when it is not full. When the current CFs are full, a new empty CF will be appended to the DCF. An item x_i with the fingerprint ξ_i can be inserted in any active CF in the DCF. Based on the observation, we can design a novel compact algorithm which repeats moving the fingerprints from a sparse CF to the corresponding buckets of other CFs until the sparse CF becomes empty. Thus, we can release the space occupied by the empty CF to improve better memory efficiency of a DCF.

We use ϵ_{DCF} and ϵ_{CF} to denote the false positive rates of a DCF and a CF, respectively. It is not difficult to see that ϵ_{DCF} is apportioned by each ϵ_{CF}. Querying an item needs to go through every CF in the worst case. Therefore, the total false positive rate ϵ_{DCF} will grow due to the partial increment of the false positive rate ϵ_{CF} of each CF and vice versa. From Eq. (2), ϵ_{CF} is affected by f, the size of fingerprint. A smaller value of f makes a CF occupy less space while leads to a higher probability of fingerprint collision. On the contrary, if we increase the value of f, we can trade off space efficiency for a lower false positive rate ϵ_{CF}, further contributing to the reduction of ϵ_{DCF}.

In the following Section 3.2, we present the operations of a DCF in detail. In Section 4, we will conduct deeper theoretical analysis of DCF after presenting the basic operations.

3.2 Operations of DCF

Insert. Initially, a DCF consists of a single *cuckoo filter*, and the insert operation has no difference from that of a CF. We design two extension strategies with the consideration of different application efficiency requirements of space and time, i.e., *active extension* and *passive extension*.

Algorithm 1 Insert (x)

```
1: ξ_x = fingerprint(x);
2: i_1 = hash(x);
3: i_2 = i_1 ⊕ hash(ξ_x);
4: if curCF is full then
5:     curCF ← allocate new building block;
6: i = randomly pick i_1 or i_2;
7: for n = 0; n < MNK; n++ do
8:     randomly pick an entry e from bucket curCF.B(i);
9:     swap ξ_x and fingerprint in e;
10:    i = i ⊕ hash(ξ_x);
11:    if curCF.B(i) has an empty entry then
12:        insert ξ_x into curCF.B(i);
13:        return true;
14: victim ← the last item kicked out;
15: nextCF ← curCF; //Initialized as current CF
16: for victim exists do
17:    nextCF ← the next building block of nextCF;
18:    insert victim into nextCF;
19: return true.
```

For applications preferring fast insert speed, the DCF provides the *active extension* strategy. In *active extension*, the DCF appends the new, empty CF aggressively whenever an insert failure occurs. An original CF is considered full when the insert failure occurs, i.e., the number of relocations for inserting an item reaches a specified maximum value, denoted as MNK. The last kicked out victim will be evicted and stored in the newly appended CF. It is clear that such an *active extension* strategy provides lower inserting delay at the cost of more space. The setting of the parameter MNK greatly affects the tradeoff between the space utilization and the insert time. We will analyze the influence of the parameter MNK in detail in Section V.

A *passive extension* strategy is provided for applications with stricter requirement of space efficiency. Specifically, the *passive extension* strategy assigns each CF a uniform capacity c, which guarantees an acceptable memory efficiency. This strategy allows the DCF to keep inserting items into the CF until its counter reaches the capacity c. Thus, a failure handle algorithm is essential in the *passive extension* to handle the kicked out victim when the number of relocations exceeds MNK.

Algorithm 1 specifies the insert operation in detail. The algorithm keeps two pointers, *curCF* and *nextCF*. The *curCF* points to the current CF and if the *curCF* is full, then a new CF building block will be allocated and assigned to *curCF*. The fingerprint will be inserted into *curCF* first, which is the same as the insert operation of CF. If the number of relocation reaches the specified maximum value, denoted as MNK, the algorithm will record the last fingerprint kicked out. In order to avoid insert failure, DCF keeps inserting the victim into the *nextCF* iteratively.

Algorithm 2 Membership Query (x)

```
1: ξ_x = fingerprint(x);
2: i_1 = hash(x);
3: i_2 = i_1 ⊕ hash(ξ_x);
4: for k = 1 to s do
5:     if CF_k.B(i_1) or CF_k.B(i_2) has ξ_x then
6:         return true;
7: return false.
```

Algorithm 3 Delete (x)

```
1: ξ_x = fingerprint(x);
2: i_1 = hash(x);
3: i_2 = i_1 ⊕ hash(ξ_x);
4: for k = 1 to s do
5:     if CF_k.MembershipQuery(x) success then
6:         remove a copy of ξ_x;
7:         return true;
8: return false.
```

Figure 3 shows an example of inserting element x into a DCF that currently has three building blocks. In Fig. 3, the first building block is already full. The pointer *curCF* points to the second building block, where the next insert operation will be performed. The fingerprint ξ_x is first generated by hash function and the two candidate buckets (the 2^{nd} bucket and the 6^{th} bucket) are computed by Eq (1). Assume we leverage the *passive extension* here. The fingerprint will be inserted into the building block pointed by curCF, i.e., the second building block. The insert process is the same as CF: after finding that the two candidate buckets are full, the fingerprint ξ_q in the 6^{th} bucket is randomly chosen. The fingerprint ξ_q relocates itself and takes up the entry of ξ_g. After ξ_g relocates itself into the empty entry in the 1^{st} bucket, the insert process ends. If insert failure occurs (i.e., the number of relocations in the second building block reaches the predefined MNK), the fingerprint kicked out will be inserted into the following building blocks (the 3^{rd} bucket in Fig. 3) one by one until the fingerprint is successfully inserted. New building blocks will be generated and appended to DCF if there are no following building blocks.

Membership Query. Membership testing with a DCF needs to probe every CF in the DCF, i.e., $2bs$ entries, in the worst case. Algorithm 2 presents the operation of the membership query in detail. The algorithm looks through all the s CFs and performs query evaluation in every CF. If a matched fingerprint is identified, the algorithm returns the positive result. If none of the CFs have a matched fingerprint, the DCF determines that the item x is not a member of the set. The time complexity of the membership query operation of a DCF and a CF are *O(bs)* and *O(b)*, respectively.

Delete. The deletion of an item x needs to first perform a membership query operation. If a corresponding fingerprint ξ_x is found, then the matched fingerprints will be removed from the DCF. Algorithm 3 shows the details of the delete operation. The time complexity of the delete operation is the same as those of the membership query operation, i.e., *O(bs)* for a DCF and *O(b)* for a CF.

Compact. DCF provides a compact operation to release the vacant space and achieve better space efficiency when the size of the dynamic set decreases. With items of a dynamic set having been deleted, the space utilization of a DCF may decrease with time. To achieve better space efficiency, the compact operation of the DCF iteratively moves the fingerprints from sparse CFs to their counterpart buckets in other denser CFs. In order to release a CF with the least fingerprint movements, we leverage a greedy strategy, which

Algorithm 4 **Compact ( )**

```
for k = 1 to s do
    if CF_k is not full then
        add CF_k to CFQ;
sort CFQ by ascending order;
for i = 2 to m do // m is the number of CFs in CFQ
    curCF ← CFQ.element[i − 1];
    for j = 1 to l do
        if bucket curCF.B(j) is not empty then
            for k = m to i do
                CFQ.element[k].B(j) ← fingerprints of
                curCF.B(j);
        if curCF is empty then
            remove curCF from DCF;
            break;
return true.
```

moves the items out of the currently sparsest CF. Figure 3 illustrates an example where CF_1 is a full building block while CF_2 is the sparest one. By moving ξ_t and ξ_n in CF_3 to the corresponding 3^{rd} and 5^{th} bucket addresses in CF_2, CF_3, the sparsest building block becomes empty and then can be released. Algorithm 4 presents the compact operation in detail. The DCF maintains a CF queue called *CFQ* to store the CFs whose counters are less than the capacity c. The CFs in the queue is sorted in an ascending order of the number of items stored. Each time the sparsest CF at the head of the queue is picked out and the fingerprints inside it are moved to the CF at the tail of the queue. If the CF at the tail of the queue cannot host all the fingerprints moved from another CF, those fingerprints will continually be moved to the second last CF in the queue by such analogy until the first CF becomes empty or all the CFs in the queue are traversed.

4 Analysis of DCF

4.1 False Positive Rate

According to the membership query operation of a DCF, the membership testing of an item x that does not belong to S, needs to check a number of s CFs. The false positive rate is defined as the probability that at least one CF among all the CFs reports a false positive for x. Assuming the false positive rate of each CF is ϵ_{CF}, the probability that no false positives happen in all the s CFs is $(1-\epsilon_{\text{CF}})^s$. Therefore, the upper bound of a DCF's false positive rate is,

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

By replacing ϵ_{CF} with Eq. (2) and further leveraging the Taylor formula, we can obtain the following approximation,

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

We plot Fig. 4 according to Eq. (4). The figure shows that the false positive rate of the DCF is correlated with both the number of CFs in DCF and the fingerprint size. Specifically, increasing the value of f will greatly reduce the false positive rate ϵ_{DCF}. At the same time, given the bucket size b and the fingerprint size f, the growth of the parameter s leads to the increase of the false positive rate ϵ_{DCF}.

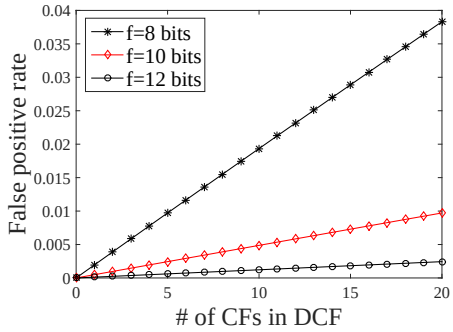


Fig. 4: The false positive rate of DCF

4.2 Reliable Deletion

In the following subsection, we evaluate the delete operation of DBF and DCF. As aforementioned, in order to avoid false negatives, a DBF has to give up the delete operation and leaves the redundant bits information remained in DBF. We will compare the delete operation of a DCF with that of DBF in this subsection.

As mentioned in [12], a DBF has the multiple address problem, i.e., a particular item x of set S is tested to be in multiple CBFs. At most only one CBF has x inserted, while all the other CBFs' reports of matching x are false positives. It is difficult to distinguish which CBF truly represents item x. Therefore, the DBF is not able to decide which CBF should perform the delete operation. Basically, there are two choices for a delete operation when such a multiple address problem occurs. 1) Randomly choose a CBF and performing the delete operation. If the selected CBF is a false positive, the delete operation should be a false deletion which destroys the DBF and leads to at most a number of k potential false negatives. 2) Perform delete operation in all the CBFs appearing to represent x. This guarantees that the item x will be deleted thoroughly. However, it leads to more serious false negatives problem. Both of the above two options will bring false negatives to DBF, which makes the DBF not usable. Thus, the DBF chooses neither of the solutions above. Instead, it gives up the delete operation and keeps the membership information of x remained to avoid introducing false negatives.

It is clear to see that the redundant information remained in the DBF will raise up the false positive rate. Assuming the number of deleted items in DBF is n_0, the upper bound of the probability of multiple address problem is computed by $\epsilon_{\text{multi}} = 1 - (1 - \epsilon_{\text{BF}})^{s-1}$ [12]. After n_0 items are deleted, there will be at most $n_0 \times \epsilon_{\text{multi}}$ items facing the multiple address problem. After deleting n_0 items, the false positive rate should be,

$$\epsilon_{\text{real}}(n) = \epsilon_{\text{DBF}}(n + n_0 \times \epsilon_{\text{multi}}), \tag{5}$$

where $\epsilon_{\text{real}}(n)$ denotes the false positive rate after deleting n_0 items, while ϵ_{DBF} represents the false positive rate of DBF without multiple address problem. Theoretically, it is clear to see from Eq.(5) that the items remained will increase the false positive.

DCF offers a better solution, the delete operation first requires a membership query evaluation. It is possible to

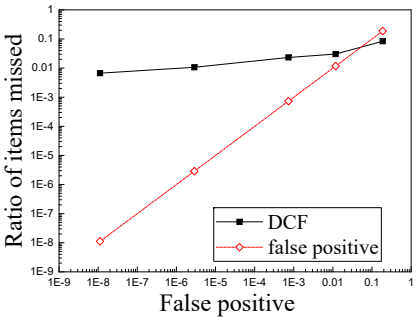


Fig. 5: Ratio of items missed under different false positive

find multiple matched fingerprints when traversing all the fingerprints in the total number of $2bs$ entries. Different from the multiple address problem of the DBF, the multiple address problem of a DCF is the case when an item x appears to be represented by multiple fingerprints hosted in a DCF. The upper bound of the probability of a DCF's multiple address problem is calculated by,

$$\epsilon^*_{\text{multi}} = 1 - (1 - 1/2^f)^{2bs-1}. \tag{6}$$

Basically, there are three possible solutions for dealing with the multiple address problem of a DCF. The first option is to delete all the matched fingerprints in the DCF. This guarantees that the item x will be removed thoroughly. However, it brings false negatives to the DCF because it removes not only item x but also the items which happen to have the same fingerprints ξ_x. The probability of false deletion is bounded by $\epsilon^*_{\text{multi}}$. The second option is to keep the fingerprint information in DCF. This will lead to the same problem as DBF: a number of $n_0 \times \epsilon^*_{\text{multi}}$ redundant fingerprints are kept by the DCF after deleting n_0 items. This further influences the false positive rate of DCF. Therefore, the DCF adopts the third option, which removes one copy of the matched fingerprints. It is clear that compared with a DBF, the delete algorithm of DCF will not lead to redundant information left and will not leave a permanent influence to false positive rate. Therefore, we conclude that our DCF design outperforms DBF in supporting of delete operation, and thus DCF is more practical in representing dynamic sets in real world applications.

In the following discussion, we discuss the reliability of the delete operation of the DCF design. Obviously, in order to guarantee a safe deletion, only previously inserted items can be removed. Thus, we only consider the delete operation with two abnormal but inevitable situations in practice, including multiple value and duplicates.

Multiple Value: Different from the multiple address problem of the DBF, diverse items have very low probability to be inserted with the same fingerprint in the same bucket address in the DCF design. To differentiate from the multiple address problem, we call this multiple value. We consider the case where the items x and y share the same bucket address and happen to collide in the same fingerprint ($\xi_x = \xi_y$). According to Eq. (1), the addresses of the alternative buckets of items x and y are the same as well. If deleting x removes one copy of the fingerprint, the item y can still be found. In this case, it seems that the false positive rate increases since querying x still succeeds. However, we should notice that determining the existence of item x after the deletion of x is essentially equivalent to a false positive, whose probability is computed by Eq. (4). Even if the DCF encounters multiple value, removing one matched fingerprints does not lead to redundant information left due to the monopolistic fingerprint, which guarantees that no false negative is introduced. Therefore, we can conclude that compared with the DBF, our DCF design supports reliable delete operation.

Duplicates: Duplicated items commonly occur in real world systems. Assuming the item x has been inserted twice, there must be two copies of the fingerprint ξ_x inserted in the DCF. Obviously, deleting item x thoroughly requires removing fingerprint ξ_x twice. If inserting duplicated items is not allowed, the DCF can filter the duplicate by performing a query operation before insertion. This guarantees that items can be removed thoroughly by removing the fingerprint once. At the same time, it might introduce slight false negatives. Considering the case that we mentioned in multiple value, items x and y share the same bucket address and happen to collide in the same fingerprint ($\xi_x = \xi_y$). When x and y both need to be inserted into the DCF, only one copy of the fingerprint is inserted to avoid duplicates. After inserting a single copy, a possible false negative may occur. For example, item y will no longer be found if we delete item x. We examine the ratio of items missed during insert operation (the fraction of items that should be inserted but filtered as duplicates by mistake) under different false positive rates. Figure 5 shows the ratio changes slowly when the false positive rate varies, and storing enormous duplicates will result in the decrease of the space efficiency due to the nonuniform distribution of fingerprints to trade off extremely slight false negative for a higher space efficiency is preferable for certain applications. According to the analysis above, we recommend not to filter the fingerprints when handling data sets with rare duplicates while filtering duplicates is preferable for data sets with large fraction of duplicates to achieve better space efficiency.

5 Optimization of DCF

In this section, we discuss the optimization of the DCF. We mainly analyze two important aspects, including the setting of MNK (i.e., the maximum number of relocations) and how to optimize the space cost.

5.1 Maximum Number of Relocations

Here, we mathematically analyze how the settings of the parameter MNK affects the DCF in detail. Given a CF with l buckets each with b entries, the utilization of the CF is proportional to the number of inserted items while the average insert time has positive correlation to the total number of relocations of all the inserted items. Therefore, we turn to analyzing the influence of MNK on the expected number of inserted items as well as the expected total number of relocations during the insert operation.

The expected number of inserted items. When the number of inserted items is n, the probability that a certain bucket is full can be computed by,

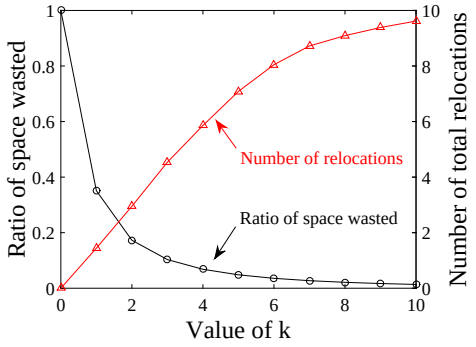


Fig. 6: Ratio of space wasted and number of relocations

$$\rho(n) = \frac{P_b^n \times P_{n-b}^{(l-1)b}}{P_n^{lb}}, n \in [0, lb]. \tag{7}$$

where P_b^n denotes b-permutations of n.

By leveraging Eq. (5), the probability that a number of τ relocations happen when successfully inserting the n^{th} item can be computed by,

$$P(T = \tau | N = n) = \rho^{\tau}(n-1)[1 - \rho(n-1)], \tau \in [0, +\infty). \tag{8}$$

Accordingly, the probability that less than MNK relocations happen when successfully inserting the n^{th} item can be computed by Eq. (9),

$$\begin{aligned} P(T < \text{MNK} | N = n) &= \sum_{\tau=0}^{\text{MNK}-1} P(T = \tau | N = n) \\ &= 1 - \rho^{\text{MNK}}(n-1). \end{aligned} \tag{9}$$

The probability of successfully inserting the n^{th} item with a failure of inserting the $(n+1)^{th}$ item is quantified by Eq. (10), which quantifies the probability that the DCF only successfully inserts n items.

$$\begin{aligned} \Theta(N = n) &= \{\prod_{j=1}^{n} P(T < \text{MNK} | N = j)\} \\ &\times P(T \geqslant \text{MNK} | N = n+1) \\ &= \{\prod_{j=1}^{n} [1 - \rho^{\text{MNK}}(j-1)]\} \times \rho^{\text{MNK}}(n). \end{aligned} \tag{10}$$

Thus, the expected number of inserted items in a CF is as bellow,

$$\begin{aligned} E[N] &= \sum_{i=0}^{lb} i \times \Theta(N = i) \\ &= \sum_{i=0}^{lb} \{i \times \prod_{j=1}^{i} [1 - \rho^{\text{MNK}}(j-1)] \times \rho^{\text{MNK}}(i)\}. \end{aligned} \tag{11}$$

According to Eq. (11), the expected number of items which could be stored in a CF is related to the variable MNK. By setting the number of bucket l to eight and the number of entries b to four, we plot the ratio of space wasted in Fig. 6. Figure 6 shows that a fraction of 95% of the entries can be filled with fingerprints when the value of MNK is set to five. The utilization changes slowly when MNK reaches three. Figure 6 also shows that the space efficiency increases with the growth of MNK. If we pay more attention to the space utilization of a CF, we can set MNK relatively large in practice.

The expected total number of relocations. According to Eq. (8), the expected number of relocations of inserting the n^{th} item is computed by,

$$\begin{aligned} E[R](\tau, c) &= \sum_{\tau=0}^{\text{MNK}-1} \tau \times P(T = \tau | N = n) \\ &= \sum_{\tau=0}^{\text{MNK}-1} \tau \times \rho^{\tau}(n-1)[1 - \rho(n-1)]. \end{aligned} \tag{12}$$

By leveraging the number of inserted items $E[N]$ obtained from Eq. (11), the total number of expected relocations of inserting a number of n items can be derived from Eq. (13),

$$\text{SUM}_E = \sum_{c=1}^{E[N]} E[R] = \sum_{c=1}^{E[N]} \sum_{\tau=0}^{\text{MNK}-1} \tau \times \rho^{\tau}(c-1)[1 - \rho(c-1)]. \tag{13}$$

Figure 6 shows that the expected total number of relocations increases with the growth of the value of MNK.

In the example shown in Fig. 6, we can find that there exists a knee point in the ratio of the space wasted curve, i.e., when MNK is around three in the example in Fig. 6. After reaching the knee point, the ratio of wasted space decreases slowly while the growth of the number of relocations still increases normally. Therefore, we suggest setting MNK equal to the knee point to achieve an optimum tradeoff between space and time costs during the construction of the DCF.

5.2 Space Optimization

We analyze the space cost of the DCF by adjusting the table length and the number of building blocks. In real application systems, the largest size of set N can be several orders of magnitude larger than the average cardinality [8]. Therefore, it is important for the DCF to optimize the space efficiency according to the history records. In real systems, the attributes, such as the maximum number of items N that can be processed and the distribution of the size of a dynamic data set can be easily obtained through system logs.

Assuming that the DCF has s building block CFs. According to Eq. (3), given the false positive rate of DCF ϵ_{DCF}, the false positive rate for each CF is computed by $\epsilon_{\text{CF}} = 1 - (1 - \epsilon_{\text{DCF}})^{\frac{1}{s}}$. Assuming that a DCF can accommodate at most N items, the capacity of each CF can be computed by $c = \lceil \frac{N}{s} \rceil$. With a given distribution of the size of the dynamic set, e.g., uniform, normal, or Zipf distribution, let p_j represent the probability that the set S has a number of j items ($1 \leq j \leq N$ and $\sum_{j=1}^{N} p_j = 1$). In order to compute the expected number of bits used, we need to know the probability of a number of i CFs are used ($1 \leq i \leq s$). We simply use the notation r_i to represent the

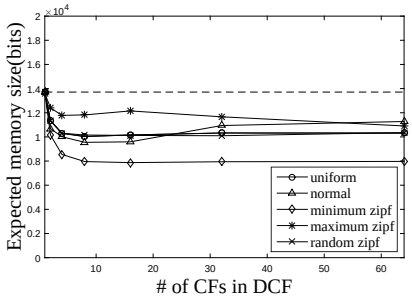


Fig. 7: The expected memory size under different distribution.

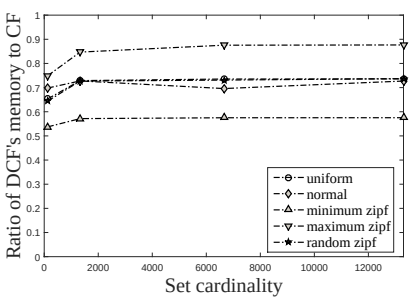


Fig. 8: The ratio of the optimal expected memory size of DCF to CF.

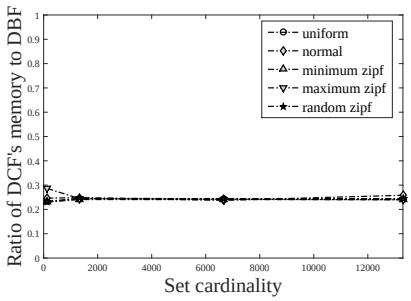


Fig. 9: The ratio of the optimal expected memory size of DCF to DBF.

probability that a number of i $(1 \leq i \leq s)$ CFs are used. It is clear that r_i can be computed by $r_i = \sum_{j=c\times(i-1)+1}^{c\times i} p_j$. Assuming each CF uses a number of m bits, the expected number of bits used by the DCF can be computed by $\sum_{i=1}^{s} i \times m \times r_i$. According to the structure of the CF, the total number of bits used is $m = \lceil \frac{N}{s} \rceil \times \frac{f}{\alpha}$ where f represents the size of a fingerprint and α represents the load factor (or utilization) of the CF. Thus, the expected number of bits used is $\sum_{i=1}^{s}(r_i \times i \times (\frac{N}{s}) \times \frac{f}{\alpha})$. After substituting f with $f = log_2(\frac{2b}{\epsilon_{\text{CF}}})$, which is derived from Eq. (2). The optimization problem can be modeled as a linear programming problem,

$$\begin{aligned} \underset{s}{\text{MIN}} \quad & \sum_{i=1}^{s} r_i \times i \times (\frac{N}{s}) \times \frac{log_2(\frac{2b}{\epsilon_{\text{CF}}})}{\alpha} \\ \text{s.t.} \quad & \epsilon_{\text{CF}} = 1 - (1 - \epsilon_{\text{DCF}})^{\frac{1}{s}}, s > 0. \end{aligned} \tag{14}$$

With the above linear programming, we aim at obtaining a certain value of s to achieve the minimal expected number of bits used by the DCF. Once the value of s is determined, the capacity c, false positive rate ϵ_{CF} and the fingerprint size f can also be obtained from the equation. Therefore, we can obtain all the parameters required from the linear programming equation to build a space optimized DCF. We can solve the linear programming problem using the simplex method [7].

Figure 7 shows the expected memory size of the DCF under five distributions of the dynamic set cardinalities where the false positive rate ϵ_{DCF} is set to 9.8×10^{-3}, while the upper bound of the set cardinality N is 1,330. The result shows how the expected memory size changes with the number of CFs (s varying from one to 64).

The baseline implies that the space is allocated by the CF. The baseline remains unchanged because the space of the CF is pre-allocated for all possible items. With current parameters, the optimal memory size is achieved when s is equal to an inflection point under all the five distributions. We set the value of s to the inflection point under different distributions and plot the ratio of the optimized memory sizes of the DCF to those of the CF in Fig. 8. We can see from Fig. 8 that the DCF reduces the optimal expected memory size of the CF by 25% under uniform, normal and random Zipf distributions. The DCF reduces optimal expected memory size of the CF by 15% and 40% under maximum Zipf and minimum Zipf distributions, respectively. Figure 9 shows how the ratio of the DCF's optimal expected memory size changes with the DBF's optimal expected memory size. The DCF reduces the memory size of the DBF by 75%.

6 Performance

We have implemented the DCF toolkit [1]. In this section, we evaluate the performance of the DCF by comparing the performance of the DCF with that of the previous DBF design. We further implement a prototype file backup system and examine the performance of the DCF for data deduplication using large-scale real world datasets.

6.1 Experiment Results

As aforementioned, a DBF requires k hash functions to generate k bit addresses for membership testing. Thus, the computation for hash addressing would be performed k times. Some optimization techniques speed up the process of generating k bit addresses by using hash functions twice instead of k times. Such a kind of optimization is effective in query intensive applications where computationally non-trivial hash function are used [6]. For example, A. Kirsch *et al*. [13] use two hash functions $h_1(x)$ and $h_2(x)$ to simulate the rest $k-2$ hash functions by using the following equation,

$$h_{i+2}(x) = h_i(x) + ih_{i+1}(x) + i^2. \tag{15}$$

We choose SHA1 to generate the hash values. In the DCF implementation, we use the highest 32 bits and lowest 32 bits to represent the fingerprint and one of the bucket address, respectively. In DBF, the corresponding two parts of the hash value represent the value of $h_1(x)$ and $h_2(x)$, respectively. The DBF further simulates the other $k-2$ hash values using Eq. (15). Thus, the implementations of DCF and DBF consume nearly the same computation in the hash functions.

In the experiment, we set the false positive rate of DCF and DBF to the same fixed value of 1.17×10^{-2}. We conduct two groups of experiments. The first group of experiments compares the operation speeds of DCF and DBF for item insert, membership query and element delete. In the experiment, we configure both DCF and DBF with the space optimized parameter settings when varying the size of a dynamic set from zero to 64,512. In the experiment, we observe that under the chosen set cardinality, the optimal

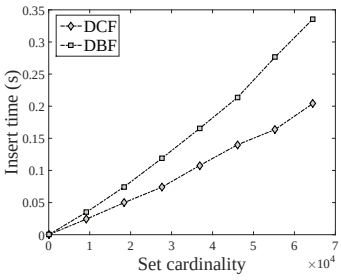


Fig. 10: Insert time with different value of N

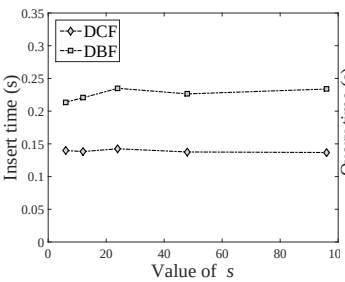


Fig. 11: Insert time with different value of s

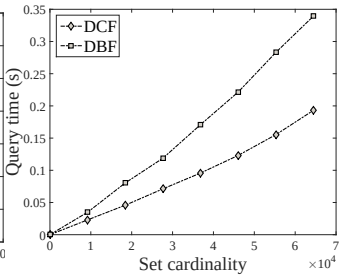


Fig. 12: Query time with different value of N

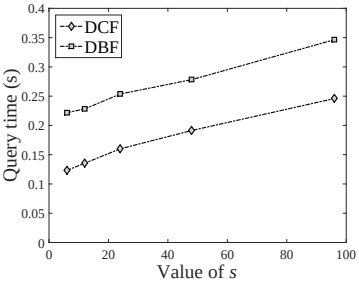


Fig. 13: Query time with different value of s

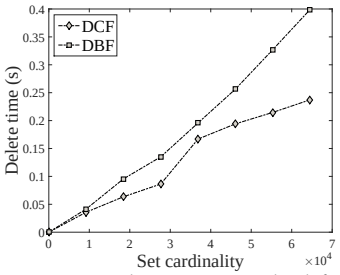


Fig. 14: Delete time with different value of N

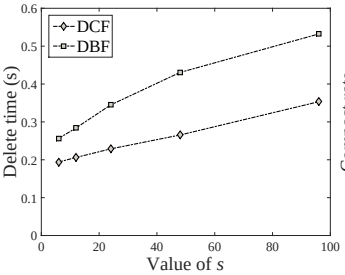


Fig. 15: Delete time with different value of s

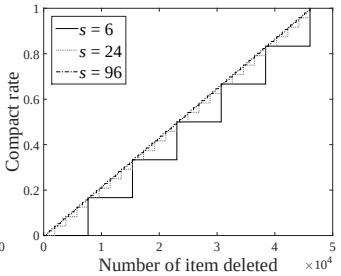


Fig. 16: Compact rate with different number of item deleted

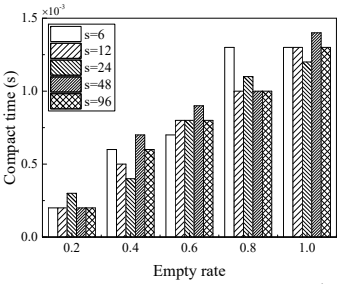


Fig. 17: Compact time with different empty rate

spaces for DCF and DBF are achieved when the value of s is around six. Therefore, in the first group of experiments we fix the parameter s at six for both DCF and DBF. The second group of experiments evaluate the influence of parameter s when the set cardinality is fixed at 46,080. We vary the value of s to see how it influences the operation speed of item insert, membership query and element delete of both DCF and DBF. We evaluate the speed of different operations by computing the time consumed by each operation and the number of operated items.

Item Insert. Figure 10 shows the time consumed by DCF and DBF for the insert operation with different sizes of set when s is set to six. The results show that DCF reduces the construction time of DBF by 35%, while improving the construction speed of DBF by 50% in average. We can also see that the performance gap for item insert operation between DCF and DBF becomes greater with the increase of the set cardinality.

Figure 11 illustrates the time consumed by DCF and DBF for item insert when s varies from six to 96. The results show that the time consumed for insert by both DCF and DBF changes very slightly with the increase of s. The results are in agreement with the analysis in Section 3. The insert operation in DCF and DBF are operated in a certain CF or CBF and it requires no extra operations. Therefore, the value of s has no significant influence on the insert operation. We can also derive from Fig. 11 that the DCF nearly always outperforms DBF by 50% in terms of construction speed.

The results show that the insert speed of DCF is slightly affected by the elastic structure. The changing trends of the insert time under different values of s are linear with the growth of set cardinality. Such an advantage of the DCF provides a fast and stable construction speed in practical implement, which is crucial for applications with a huge incoming data flow.

Membership Query. To evaluate the performance of membership query, we let all the queried items in the experiment pre-inserted into DCF and DBF. Figure 12 and 13 present the time spent for membership query.

Figure 12 shows the time spend for membership query during different number of queried elements when $s = 6$. The results show that the time spent for both DCF and DBF change nearly linearly when the number of queried elements increases. We can see that DCF reduces the query time of DBF by 45%, while it improvs the query speed of DBF by 80% in average. The performance gap between DCF and DBF increases with the growth of the number of queried elements.

Figure 13 shows how the membership query time changes when s varies from six to 96. Different from the insert operation, the membership query time of both DCF and DBF grow linearly with the increase of s. The results show that DCF achieves a higher speed for membership query than DBF. The increase of the membership query time with s is in the expectation, since the membership query operation needs to probe every CF until a matched result is found. Thus, a larger value of s definitely results in the rise of the cost for the membership query operation. If the system administrator only considers the speed of the membership query operation, the value of s should be set as small as possible. However, when more factors are taken into consideration, more attention should be paid to achieve a good tradeoff between space and time.

Element Delete. To evaluate the performance of element delete, we let the items pre-inserted into both DCF and DBF. We evaluate the delete speed by computing the time

TABLE 2: Dataset description

Dataset	Total size (GB)	Total files (thousands)	Versions	Avg. snapshot size (GB)	Avg. number of files in a snapshot
Kernels(Linux 2.6.0-2.6.39)	13	903	40	0.3	23
CentOS (5.0-5.6)	31	1,300	7	4.5	195

consumed for the delete operation. Figure 14 shows that the time spent for the delete operation of DCF and DBF increases when the number of deleted elements grows. DCF reduces the delete time of DBF by 20%, while improving the delete speed of DBF by 25% in average, when we set s to six in the implementation. Figure 15 compares the delete time of DCF and DBF under different settings of s. It is clear that the value of s has great impact on the delete speed for both DCF and DBF. A higher value of s results in a lower delete speed, because the delete operation needs to perform a membership query operation to locate the target fingerprint, while a membership query operation needs to probe every CFs until a matched result is found. Therefore, the value of s will pose a direct impact on the delete speed. The results in Fig. 15 show that the delete time of DBF doubles when s changes from six to 96 while the delete time of DCF with $s = 96$ is greater than that with $s = 6$ by 60%. For the purpose of speeding up the delete operation, the value of s should be set to a small value.

Space Compact. The compact algorithm is essential for the support of the DCF's elastic capacity, which guarantees the space efficiency of the DCF especially under the condition of frequent delete operations. For evaluating the compact algorithm of DCF, we compared the number of CFs in DCF before the performing the compact operation with that after the operation. We compute the ratio of space saved by performing the algorithm with different settings of the number of items deleted. We further vary the parameter s from six to 96 to examine how the values of s influence the performance of the compact operation.

In the experiment, we surprisingly find that the compact algorithm is highly sensitive to the number of deleted items while the algorithm can achieve a compact rate extremely close to the upper bound shown in Fig. 16. For example, in the experiment we implement 46,080 items in a DCF with $s = 6$. Theoretically, when the faction of deleted items is $1/6$, the compact algorithm can achieve an empty CF and release the space, thus obtaining a compact rate of $1/6$. In the experiment, we set the faction of deleted items to 0.1666, which is a little smaller than $1/6$, and 0.1667, which is a little larger than $1/6$. We find that the achieved compact rate is zero in the former setting, while it is $1/6$ in the latter. We observe the same high sensitivity when varying the faction of deleted items and the value of s. We run each of the above experiments fifty times and report the average results. The results show that the space will shrink when the number of items in DCF is at least ten items less than that of theoretical compact limit. The high sensitivity of compact algorithm is mainly because the distribution of fingerprints is nearly absolutely random in each CF. Essentially, the hash function we choose and the relocation technique of fingerprints mainly contributes to this randomness.

Figure 16 also shows the upper bound of the compact rate with different settings of the number of deleted item and different value of s. The higher value of s contributes to a higher theoretical upper bound of the compact rate. In order to achieve a higher average compact rate, we can increase the value of s.

We further analyze the time spent for the compact operation for 46,080 items under different delete fraction. Figure 17 shows that the compact time has no distinct relation with the value of s while it yields a linear relation to the delete fraction, which directly influences the fingerprints relocation workload. Moreover, the highest compact time takes only 1.12% of the membership query time for 46,080 items under the same configuration. The time consumed by compact operation is next to nothing compared with the query time and it will not bring too much interference to other operations in practice. Thus, we conclude from the analysis that the compact algorithm is a highly sensitive and efficient algorithm.

From the results above, it is clear that the value of s plays an important role in the time and space efficiency of the DCF. A higher value of s leads to a higher compact rate with the lower speeds of membership query and element delete. The tradeoff between the space and time is important for a system administrator of a DCF. Considering both of the space and time, we suggest the value of s should be set to the value that achieves the optimized space for DCF, which can be calculated by solving the linear program problem denoted by Eq. (14).

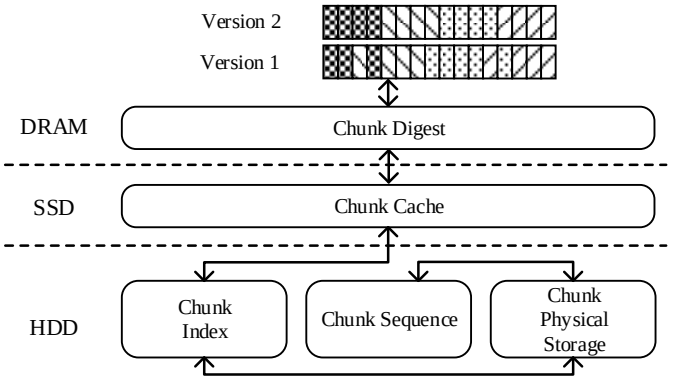


Fig. 18: Prototype system of data deduplication

6.2 Implementation in File Backup System

We apply the DCF in the file backup system for data deduplication [11]. The system eliminates duplicated data chunks during file backup to save unnecessary storage space and provide cost-effective Internet scale service. Previous research shows that indexing chunks of the data requires a large amount of space (e.g., indexing every 1PB data raises 8TB index). It is clear that storing such a huge size of index in DRAM is prohibitively costly. Moreover, identifying new chucks by checking against the chuck index stored in HDD suffers the disk I/O bottleneck [26]. Instead of relying on

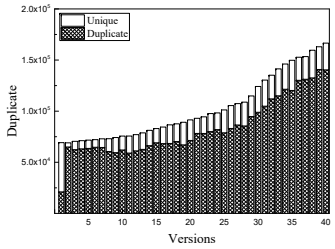


Fig. 19: Chunk Type

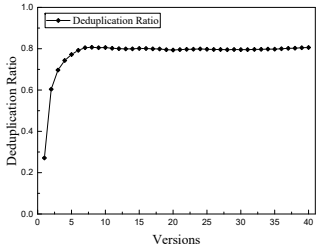


Fig. 20: Deduplication Ratio

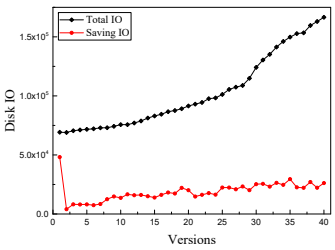


Fig. 21: Disk IO saving

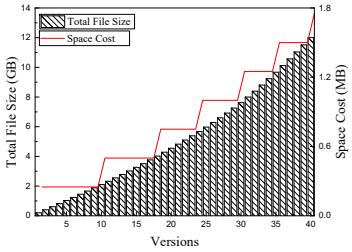


Fig. 22: Space Cost

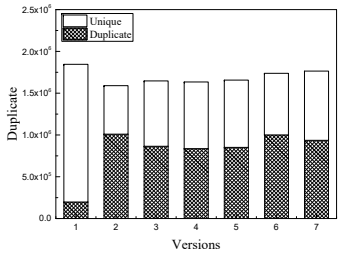


Fig. 23: Chunk Type

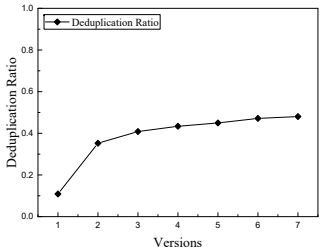


Fig. 24: Deduplication Ratio

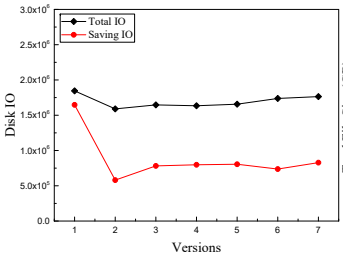


Fig. 25: Disk IO saving

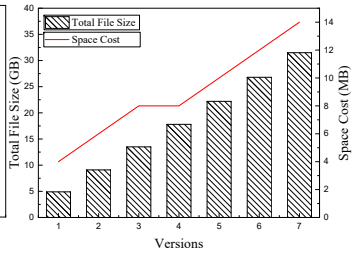


Fig. 26: Space Cost

the on-disk chuck index, our idea is to represent the set of all the stored chucks using DCF and store the succinct data structure in DRAM for chunk deduplication. When a new version of a file is uploaded, the system checks against the DCF to identify new chunks instead of checking the on-disk index. Therefore, the system can quickly determine new chunks and avoid storing duplicated chunks to save a potential huge amount of unnecessary disk I/O and backend storage.

Figure 18 presents the architecture of the prototype file backup system. In the system, a file uploaded by a user is divided into chunks based on the content [16]. The system computes a fingerprint for each chunk using a hash function. The *Chunk Digest*, which resides in the DRAM, represents the set of fingerprints for all the chunks stored in the backend physical storage system. By checking against the *Chunk Digest*, the system can identify new chunks without accessing the *Chunk Cache* in SDD [14] (the *Chunk Cache* caches the recently frequently requested chunks) and the *Chunk Index* in HDD. We deploy the dedeuplication module on a machine equipped with an octa-core 2.4GHz Xeon CPU, 32GB RAM and 1TB HDD.

Data chunks can be generated by three kinds of deduplication algorithms [18], i.e., file level chunking, fixed block chunking, and variable chunking algorithms. In file level chunking, data is partitioned according to file boundaries. By dividing data into fixed-size data chunks, fixed block chunking can offer a fined-grained deduplication method with high processing rate and low level CPU overhead. However, fixed block chunking suffers from the boundary-shifting problem. When considering two versions of the same file, version 1 and 2, the two versions of the file only has minor differences, which is caused by adding a single byte in the beginning of version 2. After exerting the fixed block chunking algorithm, chunks generated by version 2 will be totally different from version 1, because the corresponding data in version 2 has been shifted to the latter data chunks. Another type of chunking algorithm is called variable chunking, also called content-defined chunking [15]. It leverages a sliding window moving forward alone the data until a matched data pattern is found and the end of the data pattern is considered the boundary of a chunk. Obviously it can avoid boundary-shifting problem caused by data adding or removing, and only the first chunk of version 2 will be different from version 1 in the boundary-shifting issue in fixed block chunking.

Table 2 summarizes the datasets we use as backup files and their characteristics [21]. *Kernels* is the uncompressed source code of Linux kernel. It includes 903,000 files, 13GB in total and 40 versions spanning from version 2.6.0 to 2.6.39. *CentOS* is the installation files of seven different versions of CentOS Linux distribution spanning from 5.0 to 5.6. It includes 1,300,000 files and 31GB in total.

In the same time, variable chunking algorithm, i.e., Rabin-based CDC is also implemented to generate data chunks and the corresponding fingerprint. Rabin fingerprint can be computed in linear time because it is computed using polynomials over a finite field. Cooperating with a sliding window, Rabin fingerprint can quickly compute and judge the incoming data byte by byte [24].

In the two different datasets, the expected building block number is set to six according the estimation of the total chunk number and the false positive rate is limited to 0.001 The expected chunk size in Rabin chunking algorithm is set to 4KB.

Figures 19 and 23 show the duplicate and unique chunk number identified by DCF in two different datasets, *Kernels* and *CentOS*, respectively. In *Kernels* dataset, 80% to 95% chunks are duplicated compared with previous versions and in *CentOS* dataset the duplicate chunks takes up only 50% to 67% of the total chunk number.

Figures 20 and 24 show the deduplication ratio of the two datasets recorded by DCF. The deduplication ratio is computed by $1 - \frac{size\ after\ deduplication}{size\ before\ deduplication}$. Obviously, *Kernels* dataset leads to a higher deduplication ratio, 80% at most, and *CentOS* datasets only has a 45% deduplication ratio.

Figures 21 and 25 show the disk IO saved by implementing DCF and the original IO cost during deduplication process. After implementing DCF, unnecessary IO for KV index lookups can be saved by identifying those unique chunks. It is easy to obtain from Fig. 22 and Fig. 26 that DCF saves a steady portion of disk IO of the total IO, and the more unique chunks exists, the more IO DCF can save.

Figure 22 and Fig. 26 show the space cost of DCF alone with the increment of ingested files. The DCF extends its capacity dynamically and steadily alone with the increment of data size ingested and remains a extremely low space overhead compared with the file size.

7 Conclusion

In this paper, we propose the DCF design for approximate representation and membership testing for a dynamic set. To the best of our knowledge, the DCF is the first data structure to support both reliable element deletion and flexible structure extending/reducing for approximate dynamic set representation. We show that the DCF greatly reduces the space cost of the existing schemes as well as provide the reliable delete operation. Experiment results show that this DCF design greatly outperforms the state-of-the-art designs.

8 Acknowledgements

We thank the anonymous reviewers for the comments with care and insights. The preliminary result of this work was presented in IEEE ICNP 2017.

References

[1] The Dynamic Cuckoo Filter Toolkit, https://github.com/CGCL-codes/DCF, 2017.

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

[3] 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.

[4] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Cliffhanger: Scaling performance cliffs in web memory caches," in *Proceedings of NSDI*, Santa Clara, CA, March 2016, pp. 379–392.

[5] M. Dietzfelbinger and C. Weidling, "Balanced allocation and dictionaries with tightly packed constant size bins," *Theoretical Computer Science*, vol. 380, no. 1-2, pp. 47–68, 2007.

[6] P. C. Dillinger and P. Manolios, "Fast and accurate bitstate verification for SPIN," in *Proceedings of SPIN*, Barcelona, Spain, 1-3 April 2004.

[7] J. Dongarra and F. Sullivan, "Guest editors' introduction: The top 10 algorithms," *Computing in Science and Engineering*, vol. 2, no. 1, pp. 22–23, 2000.

[8] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *Proceedings of SIGCOMM*, Philadelphia, Pennsylvania, USA, 1-3 June 1999.

[9] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of CoNEXT*, Sydney, Australia, 2-5 December 2014.

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

[11] 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*, Santa Clara, CA, USA, 16-19 February 2015.

[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] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better bloom filter," *Random Structures and Algorithms*, vol. 33, no. 2, pp. 187–218, 2008.

[14] H. Ma, L. Liu, L. Pan, and J. Xu, "LSB page refresh based retention error recovery scheme for MLC NAND flash," *SCIENCE CHINA Information Sciences*, vol. 59, no. 4, pp. 042 408:1–042 408:11, 2016.

[15] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP*, Chateau Lake Louise, Banff, Alberta, Canada, 21-24 October 2001, pp. 174–187.

[16] A. Muthitacharoen, B. Chen, and D. Mazires, "A low-bandwidth network file system," *Acm Sigops Operating Systems Review*, vol. 35, no. 5, pp. 174–187, 2001.

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

[18] J. Paulo and J. Pereira, "A survey and classification of storage deduplication systems," *ACM Comput. Surv.*, vol. 47, no. 1, pp. 11:1–11:30, 2014.

[19] P. Reynolds and A. Vahdat, "Efficient peer-to-peer keyword searching," in *Proceedings of Middleware*, Rio de Janeiro, Brazil, 16-20 June 2003, pp. 21–40.

[20] H. Song, F. Hao, M. S. Kodialam, and T. V. Lakshman, "Ipv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards," in *Proceedings of INFOCOM*, Rio de Janeiro, Brazil, 19-25 April 2009, pp. 2518–2526.

[21] V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuenning, and E. Zadok, "Generating realistic datasets for deduplication analysis," in *2012 USENIX Annual Technical Conference*, Boston, MA, USA, 13-15 June 2012, pp. 261–272.

[22] 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*, UT, USA, 22-27 June 2014.

[23] J. Wei, H. Jiang, K. Zhou, and D. Feng, "DBA: A dynamic bloom filter array for scalable membership representation of variable large data sets," in *Proceedings of IEEE/ACM MSWiM*, Singapore, 25-27 July 2011, pp. 466–468.

[24] W. Xia, Y. Zhou, H. Jiang, D. Feng, Y. Hua, Y. Hu, Q. Liu, and Y. Zhang, "Fastcdc: a fast and efficient content-defined chunking approach for data dedupli-

cation," in *2016 USENIX Annual Technical Conference, USENIX ATC*, Denver, CO, USA, 22-24 June 2016, pp. 101–114.

[25] S. Zengin and E. G. Schmidt, "A fast and accurate hardware string matching module with bloom filters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 2, pp. 305–317, 2017.

[26] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proceedings of FAST*, San Jose, CA, USA, 26-29 February 2008, pp. 269–282.

Hanhua Chen received the PhD degree in computer science and engineering from the Huazhong University of Science and Technology, China, in 2010, where he is currently a professor at the School of Computer Science and Technology.

Liangyi Liao is currently a master student in the School of Computer Science and Technology, Huazhong University of Science and Technology.

Hai Jin received the PhD degree in computer engineering from the Huazhong University of Science and Technology, China, in 1994, where he is currently a professor at the School of Computer Science and Technology.

Jie Wu received the PhD degree in computer engineering from the Froida Atlantic University, USA, in 1989, where he is currently a professor at the computer and information science department at Temple University.