The Dynamic Cuckoo Filter

Hanhua Chen*, Liangyi Liao*, Hai Jin*, Jie Wu†
*Services Computing Technology and System Lab
Cluster and Grid Computing Lab
Big Data Technology and System Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
†Center for Networked Computing, Temple University, Philadelphia, PA 19122, U USA
Email: {chen, liaoliangyi, hjin}@hust.edu.cn, jiewu@temple.edu

***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 dynamically changing 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 the 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, making 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 compared to the existing state-of-the-art designs, DCF achieves 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**

I. 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 some 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 [3]. 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 an item y is an element of S, we first obtain $h(y)$, the hash address of y, and then 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. Such a scheme, however, 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 [16], routers [17], and file backup systems [11]. 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 [4, 9].

A *standard Bloom filter* (SBF) [4] is the most popular approximate set representation structure. An SBF is essentially an array of m bits, initially all 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 a 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.

Real world applications commonly involve a highly dynamic data set with items joining and leaving dynamically and with an unpredictable size of the set. For example, in popular stream applications [18], an unbounded sequence of data tuples come with the data flow. This requires a set representation

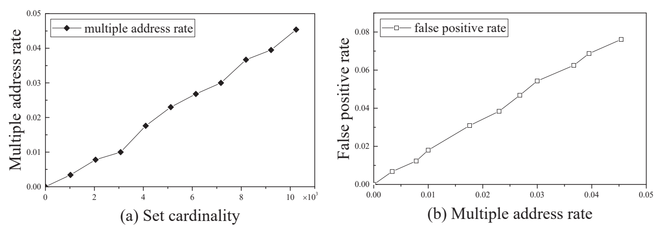

Fig. 1: Multiple address problem in DBF (The number of hash functions is set to seven. The number of CBFs varies from one to 10. Each CBF has 40,960 bits and can hold 1024 items. Totally 10,240 items are inserted in the experiment.)

data structure to have the ability to cope with sets with a changing cardinality. For another example, in a Web cache proxy [5], the cached set of Web pages is frequently updated according to the cache replacement strategies. This requires 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 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 $1 \leq i \leq k$. Fan *et al.* [9] recently proposed the *cuckoo filter* (CF) design to support the delete operation. In their design, each item x_i monopolizes a fingerprint ξ_{x_i} to represent itself and stores ξ_{x_i} in the data structure. Deleting an item x_i is performed by removing the fingerprint of x_i from the structure. The CF can achieve $1.5 \sim 2$ times query throughput with the same memory size compared to an SBF. However, the CF can not support the representation of sets with elastic sizes. We will review the CF design in more detail in Section II.B.

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]. This is 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 due to the multiple address problem [12], leaving the redundant bits information remained in the DBF.

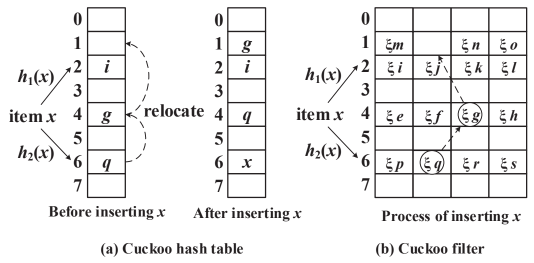

Fig. 2: Cuckoo hash table and cuckoo filter

This will result in much more serious false positives of the DBF. Figure 1 analyzes the multiple address problem of the DBF in 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, the DCF utilizes a monopolistic fingerprint for representing an item and thus enables reliable delete operation. Second, the 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 dataset 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 improves the speeds of inserting and membership testing by 50% and 80%, respectively. The implementation of the 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.

II. 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 the DBF design which are the most similar work related to our design.

A. *Cuckoo Hash Table*

A cuckoo hash table [15] 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 as below: 1) The cuckoo hash table computes the candidate bucket 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 placements 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.* [6] extended the cuckoo hash table by allowing multiple items stored in a single bucket.

B. 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 a much smaller number of bits than x itself, a CF is much more space efficient than a traditional cuckoo hash table. Formally, a CF consists of a bucket array with the length of l. Each bucket has a number of b storage units, called entries. Each entry has a fixed size of f bits, which is in agreement with the size of the fingerprints generated by the hash functions. Thus the fingerprint information of an item can be stored in a single entry.

Furthermore, storing fingerprints instead of raw data raises challenge during relocation. Without the raw data information of an element stored in a CF, it is difficult for a victim element to find the hashing address of the alternative hosting bucket for relocation. To address the problem, the CF leverages a novel hash method called partial-key cuckoo hashing, which computes the address of the alternative bucket by performing an XOR operation based on the address of the current bucket and the fingerprint to be kicked out. Specifically, the addresses of the two candidate buckets to store the fingerprint of x are computed by Eq. (1).

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

where ξ_x is the fingerprint of x.

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. Querying an item y in set S, 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 against 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 Eq. (2) [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.

C. 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 inserts 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 the item x exists in the DBF. If only one CBF is found matched, the delete operation will be performed by decreasing

TABLE I: 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

all the $h_i(x)^{th}$ counters by one for $1 \leq i \leq k$ in the matched CBF. If more than one 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.

III. Dynamic Cuckoo Filter

A. Overview

A DCF leverages CF as building block and consists of a number of s linked homogeneous CFs CF_1,...,CF_s. Initially, a DCF consists of a single CF, CF_1, and extends its capacity by appending new CFs. Each CF_k is an array of l buckets $B_k(0)$,...,$B_k(l\text{-}1)$ and each bucket $B_k(\mu)$ has a number of b entries. The DCF leverages fingerprints to represent items. A fingerprint ξ_{x_i} $(1 \leq i \leq n)$ is a hash code with a fixed size of f bits generated from the item x_i. The fingerprint ξ_{x_i} is stored in one of the entries in the candidate bucket for x_i. Each item x_i has two candidate bucket addresses, μ_{x_i} and ν_{x_i}, computed by Eq. (1) , i.e., $\mu_{x_i} = h_1(x_i)$, $\nu_{x_i} = \mu_{x_i} \oplus h_1(\xi_{x_i})$. Each CF_k is associated with a counter to record the number of items stored inside. A CF is called active when the value of its counter is under a predefined capacity. Table 1 summarizes the notations for the definition of a DCF.

Algorithm 1 **Insert** (x)

```
1:  $\xi_x$ = fingerprint($x$);
2:  $i_1$ = hash($x$);
3:  $i_2$ = $i_1 \bigoplus$ hash($\xi_x$);
4:  **if** $curCF$ is full **then**
5:      $curCF \leftarrow$ 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 $\xi_x$ and fingerprint in $e$;
10:     $i = i \bigoplus$ hash($\xi_x$);
11:     **if** $curCF.B(i)$ has an empty entry **then**
12:         insert $\xi_x$ into $curCF.B(i)$;
13:         **return** true;
14: $victim \leftarrow$ the last item kicked out;
15: $nextCF \leftarrow curCF$; //Initialized as current CF
16: **for** $victim$ exists **do**
17:     $nextCF \leftarrow$ the next building block of $nextCF$;
18:     insert $victim$ into $nextCF$;
19: **return** true.
```

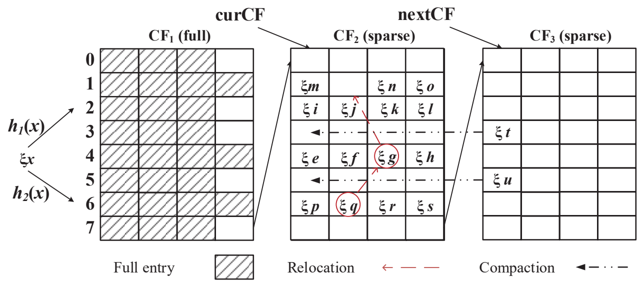

Fig. 3: An example of DCF

B. 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*.

For applications preferring fast insert speed, the DCF provides the *active extension* strategy. In *active extension*, the DCF appends 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 insert failure occurs.

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, it 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.

Figure 3 shows an example of inserting element x into a DCF that currently has three building blocks. In the example, 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). 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. Assume we leverage the *passive extension* here, the next insert operation will still be performed in *curCF* until *curCF* is full.

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 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 but not full CFs. In order to release a CF with the least fingerprint movements, we leverage a greedy strategy, which moves the items out of the currently sparsest CF. Figure 3 illustrates an example where *CF*$_1$ is a full building block while *CF*$_3$ is the sparsest one. By moving ξ_t and ξ_n in *CF*$_3$ to the corresponding bucket addresses, 3^{rd} and 5^{th} bucket, in *CF*$_2$, the sparsest building block *CF*$_3$ 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.

Algorithm 2 Membership Query (x)

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

Algorithm 3 Delete (x)

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

Algorithm 4 Compact ( )

```
1: for $k = 1$ to $s$ do
2:     if $CF_k$ is not full then
3:         add $CF_k$ to $CFQ$;
4: sort $CFQ$ by ascending order;
5: for $i = 2$ to $m$ do // $m$ is the number of CFs in CFQ
6:     $curCF \leftarrow CFQ.element[i-1]$;
7:     for $j = 1$ to $l$ do
8:         if bucket $curCF.B(j)$ is not empty then
9:             for $k = m$ to $i$ do
10:                $CFQ.element[k].B(j) \leftarrow$ fingerprints of
                   $curCF.B(j)$;
11:        if $curCF$ is empty then
12:            remove $curCF$ from $DCF$;
13:            break;
14: return true.
```

IV. Analysis of DCF

A. 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,

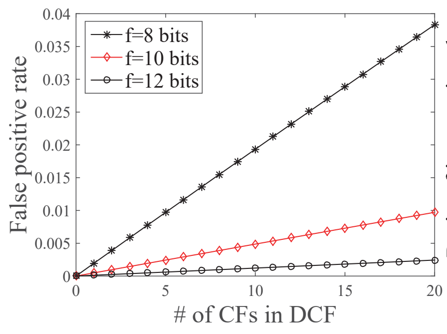

Fig. 4: The false positive rate of DCF

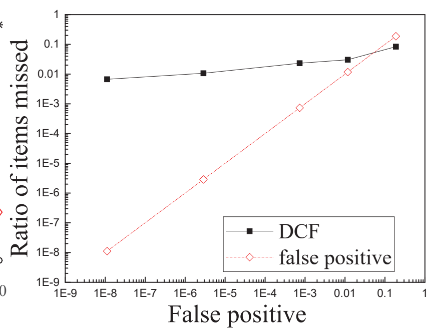

Fig. 5: Ratio of items missed under different false positive

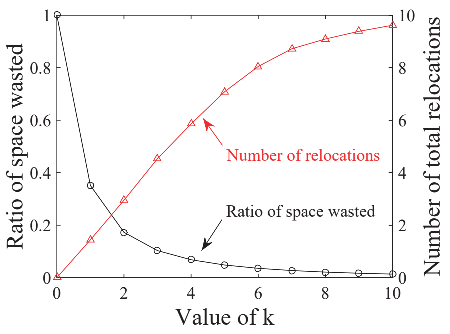

Fig. 6: Ratio of space wasted and number of relocations

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

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

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

We plot Fig. 4 according to Eq. (4), it 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}.

B. *Reliable Deletion*

In the following, 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, we call this multiple value. Considering the case that the items x and y share the same bucket address and happen to collide in the same fingerprint (i.e., $\xi_x = \xi_y$). According to Eq. (1), the addresses of the alternative buckets of items x and y are the same as well. If the deletion of 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 thanks to the monopolistic fingerprint. This guarantees that no false negative is introduced. Therefore we can conclude that compared with the DBF, our DCF design can support 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 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. Considering storing enormous duplicates will result in the decreasing 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 above analysis, we recommend not to filter the fingerprints when handling data sets without rare duplicates while filtering duplicates is considerable for data sets with large fraction of duplicates to achieve better space efficiency.

V. 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.

A. *Maximum Number of Relocations*

Here we theoretically 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.

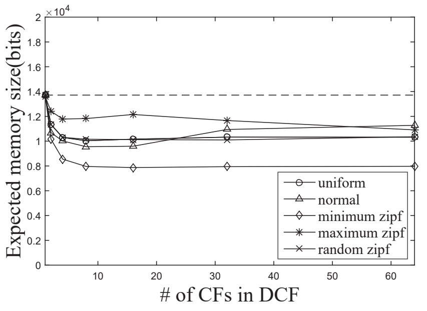

Fig. 7: The expected memory size under different distribution

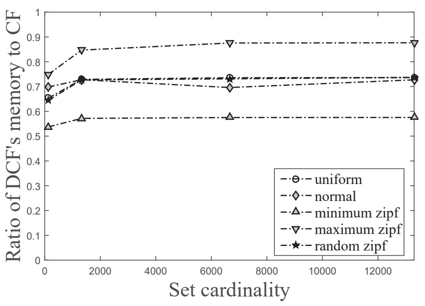

Fig. 8: The ratio of the optimal expected memory size of DCF to CF

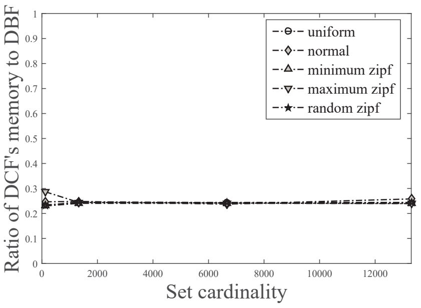

Fig. 9: The ratio of the optimal expected memory size of DCF to DBF

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,

$$\rho(n) = \frac{P_b^n \times P_{n-b}^{(l-1)b}}{P_n^{lb}}, n \in [0, lb] \tag{5}$$

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,

$$P(T = \tau | N = n) = \rho^{\tau}(n-1)[1 - \rho(n-1)], \tau \in [0, +\infty) \tag{6}$$

Accordingly, the probability that less than MNK relocations happen when successfully inserting the n^{th} item can be computed by Eq. (7),

$$\begin{aligned} P(T < \text{MNK} | N = n) &= \sum_{\tau=0}^{\text{MNK}-1} P(T = \tau | N = n) \\ &= 1 - \rho^{\text{MNK}}(n-1) \end{aligned} \tag{7}$$

The probability of successfully inserting the n^{th} item with a failure of inserting the $(n+1)^{th}$ item is quantified by Eq. (8), which quantifies the probability that the DCF only successfully inserts n items.

$$\begin{aligned} \Theta(N = n) &= \{\prod_{j=1}^{n} P(T < \text{MNK} | N = j)\} \\ &\quad \times P(T \geqslant \text{MNK} | N = n+1) \\ &= \{\prod_{j=1}^{n} [1 - \rho^{\text{MNK}}(j-1)]\} \times \rho^{\text{MNK}}(n) \end{aligned} \tag{8}$$

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

$$\begin{aligned} E[N] &= \sum_{i=0}^{lb} i \times \Theta(N = i) \\ &= \sum_{i=0}^{lb} \{i \times \prod_{j=1}^{i} [1 - \rho^{\text{MNK}}(j-1)] \times \rho^{\text{MNK}}(i)\} \end{aligned} \tag{9}$$

According to Eq. (9), 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. (6), the expected number of relocations of inserting the n^{th} item is computed by,

$$\begin{aligned} E[R](\tau, c) &= \sum_{\tau=0}^{\text{MNK}-1} \tau \times P(T = \tau | N = n) \\ &= \sum_{\tau=0}^{\text{MNK}-1} \tau \times \rho^{\tau}(n-1)[1 - \rho(n-1)] \end{aligned} \tag{10}$$

By leveraging the number of inserted items $E[N]$ obtained from Eq. (9), the total number of expected relocations of inserting a number of n items can be derived from Eq. (11),

$$\text{SUM}_E = \sum_{c=1}^{E[N]} E[R] = \sum_{c=1}^{E[N]} \sum_{\tau=0}^{\text{MNK}-1} \tau \times \rho^{\tau}(c-1)[1 - \rho(c-1)] \tag{11}$$

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 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 to the knee point to achieve an optimum tradeoff between space and time costs during the construction of the DCF.

B. 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 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 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,

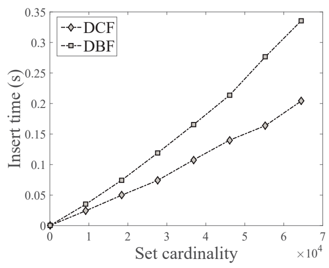

Fig. 10: Insert time with different value of N

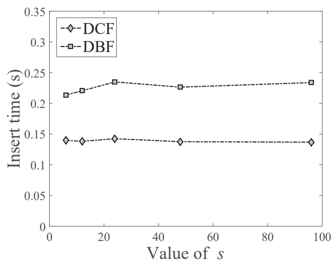

Fig. 11: Insert time with different value of s

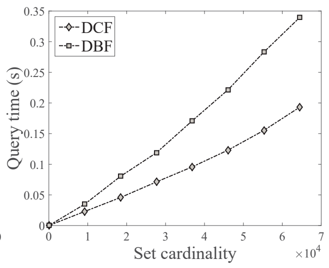

Fig. 12: Query time with different value of N

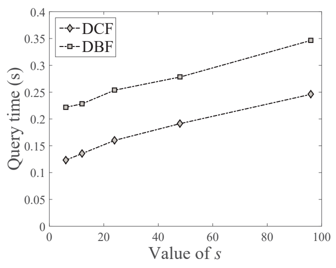

Fig. 13: Query time with different value of s

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 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,

$$\begin{aligned} \underset{s}{\text{MIN}} \quad & \sum_{i=1}^{s} r_i \times i \times (\frac{N}{s}) \times \frac{log_2(\frac{2b}{\epsilon_{\text{CF}}})}{\alpha} \\ \text{s.t.} \quad & \epsilon_{\text{CF}} = 1 - (1 - \epsilon_{\text{DCF}})^{\frac{1}{s}}, s > 0 \end{aligned} \tag{12}$$

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 the space allocated by the CF. It 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 the figure 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%.

VI. 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 real world datasets.

A. Experiment Setup

The DCF implementation uses SHA1 to generate hash values, where it uses the highest 32 bits and the lowest 32 bits to represent the fingerprint and one of the bucket address, respectively. In the DBF, the corresponding two parts of the hash value represent the value of $h_1(x)$ and $h_2(x)$, respectively. The DBF further computes the other $k-2$ hash values by Eq. (13) [13]. Thus the implementations of the DCF and the DBF consume nearly the same computation cost in hashing.

$$h_{i+2}(x) = h_i(x) + ih_{i+1}(x) + i^2 \tag{13}$$

In the experiment, we set the false positive rates of both the DCF and the DBF to a fixed value of 1.17×10^{-2}. We conduct two experiments. The first experiment compares the performance of the operations for the DCF and the DBF. We examine item insert, membership query, and the reliability of element delete. In the experiment, we configure both the DCF and the DBF with the space optimized parameter settings when varying the size of a dynamic set from zero to 64,512. We observe that the optimal space cost for the DCF and the DBF is obtained when the value of s is around six. Therefore, in the first experiment we fix the parameter s at six for both the DCF and the DBF. The table length of each building block (CF) varies from 0 to 2^{12} and is calculated automatically by DCF. The second experiment evaluates the influence of the parameter s. We fix the set cardinality at 46,080 and vary the value of s to examine how it affects the operation performance of item insert, membership query, and the reliability of item delete of both the DCF and the DBF. The table length of each building block varies from 2^{11} to 2^7 accordingly.

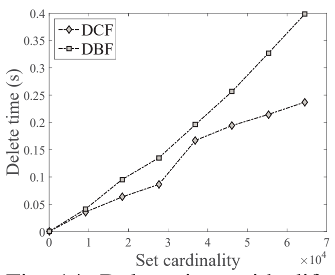

Fig. 14: Delete time with different value of N

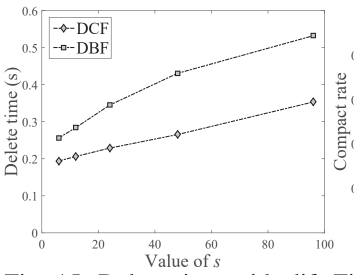

Fig. 15: Delete time with different value of s

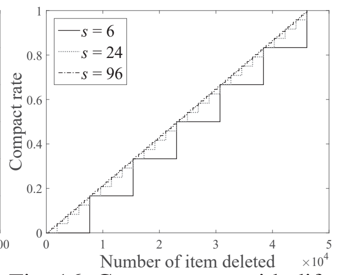

Fig. 16: Compact rate with different number of item deleted

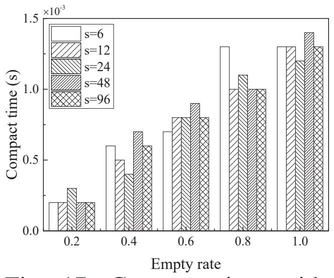

Fig. 17: Compact time with different empty rate

B. Experiment Results

Figure 10 shows that the DCF reduces the item insert time of the DBF by 35%, while improving the insert speed of the DBF by 50%. Figure 11 shows that the time consumed for item insert for both the DCF and the DBF changes slightly with the increase of s because the insert operation in the DCF and the DBF are operated in a certain CF or CBF. Figure 11 shows that the DCF nearly always outperforms the DBF by 50% in terms of insert speed.

Figure 12 shows that the DCF reduces the query time of the DBF by 45%, while improving the query speed of the DBF by 80% in average. Figure 13 shows that the membership query time of both the DCF and the DBF grow linearly with the increase of s. The increase of the membership query time with s is expected, since the membership query operation needs to probe every CF until a matched result is found.

We still consider the unreliable deletion of DBF as a baseline. Figure 14 shows that the DCF reduces the delete time of the DBF by 20%, while improving the delete speed of the DBF by 25% in average. Figure 15 compares the delete time of the DCF and the DBF under different settings of s. The result shows that a higher value of s results in a slower delete speed.

In the experiments, we find that the compact algorithm is highly sensitive to the number of deleted items. The algorithm can achieve a compact rate extremely close to the theoretical upper bound shown in Fig. 16. For example, in the experiment we test 46,080 items in a DCF with $s = 6$. Theoretically, when a fraction of 1/6 items have been deleted, 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 fraction of deleted items to 0.1666, which is slightly smaller than 1/6, and to 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 fraction of deleted items and the value of s.

Figure 17 shows that the compact time has no distinct relation with the value of s while it yields a linear relation to the empty rate. Moreover, the time consumed by the compact operation is quite short compared to the query time. The highest compact time takes only 1.12% of the membership query time for 46,080 items with the same configuration.

From the above results, it is clear that the value of s plays an important role in the computation and memory efficiency of the DCF. A higher value of s leads to a higher compaction rate with the lower speeds of membership query and element delete. Considering both space and time costs, we suggest the optimal value of s should be obtained by solving the linear program presented by Eq. (12).

C. 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. Instead of relying on 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 [14]. 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 (the *Chunk Cache* caches the recently

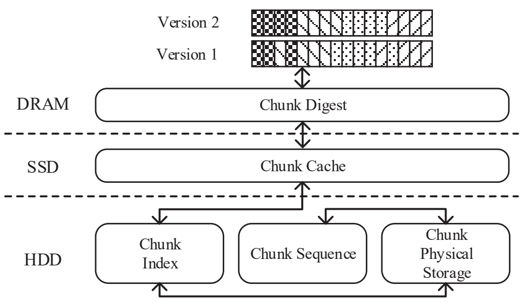

Fig. 18: Prototype system of data deduplication

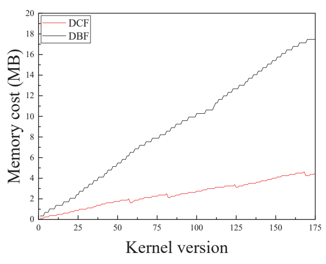

Fig. 19: Memory cost

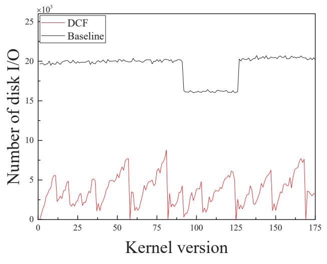

Fig. 20: Number of disk I/O

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. In the experiment, we use the 18GB dataset which includes 175 versions of the compressed source codes of Linux kernel [2]. We implement both our DCF and the previous DBF designs to support the *Chunk Digest* in the system. We evaluate the consumption of memory space by comparing the system with our DCF design to that with the DBF. We also evaluate the disk I/O of the file backup system with and without our DCF design, which is implemented as *Chunk Digest*.

In the experiment, we input the compressed Linux source codes into the file backup system from version 1 to version 175. During the backup process, we conduct four times of eliminating outdated backup chunks in version 55, 80, 125, and 170. We randomly select 50% of the stored chunks in previous 20 versions as outdated chunks. After eliminating outdated chunks, compact operation will be activated to release the vacant space of DCF.

Figure 19 compares the memory consumption of the systems with our DCF design and the DBF. The result shows that the system can save 75% memory space by using our DCF design compared to the DBF scheme. The slight decrease in version 55, 80, 125, and 170 shows that the compact operation dynamically adjust the space according to the cardinality of chunk data set.

Figure 20 compares the disk I/O for checking cache and chunk index of the system with the DCF as *Chunk Digest*. We use the disk I/O without *Chunk Digest* as baseline. The result shows that by leveraging the DCF to support the *Chunk Digest* in a file backup system, the disk I/O is greatly reduces by 62.5% in the best case. The disk I/O becomes zero in version 55, 80, 125, and 170 because the compaction after eliminating outdated chunks does not involve disk I/O. Other fluctuations in Fig. 20 imply the versions at those points have a significant update compared with previous ones.

VII. 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 provides the reliable delete operation. Experiment results show that this DCF design greatly outperforms the state-of-the-art designs.

VIII. Acknowledgement

This research is supported in part by The National Key Research and Development Program of China under grant No.2016QY02D0202, NSFC under grants Nos. 61422202, 61370233, 61433019, Foundation for the Author of National Excellent Doctoral Dissertation of PR China under grant No.201345, and Research Fund of Guangdong Province under grant No.2015B010131001. Jie Wu's research is supported in part by NSF grants CNS1629746, CNS 1564128, CNS 1449860, CNS 1461932, CNS 1460971, and CNS 1439672.

References

[1] The Dynamic Cuckoo Filter Toolkit, https://github.com/CGCL-codes/DCF, 2017.
[2] https://www.kernel.org/pub/linux/kernel/, 2017.
[3] A. P. Batson, "The organization of symbol tables," *Communications of the ACM*, vol. 8, no. 2, pp. 111–112, 1965.
[4] 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.
[5] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Cliffhanger: Scaling performance cliffs in web memory caches," in *NSDI*, Santa Clara, CA, March 2016, pp. 379–392.
[6] 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.
[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 *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 *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 *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] 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.
[15] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.
[16] P. Reynolds and A. Vahdat, "Efficient peer-to-peer keyword searching," in *Middleware*, Rio de Janeiro, Brazil, 16-20 June 2003, pp. 21–40.
[17] 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 *INFOCOM*, Rio de Janeiro, Brazil, 19-25 April 2009, pp. 2518–2526.
[18] 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 *SIGMOD*, UT, USA, 22-27 June 2014.