Efficiently Representing Membership for Variable Large Data Sets

Jiansheng Wei, *Member, IEEE*, Hong Jiang, *Senior Member, IEEE*,
Ke Zhou, *Member, IEEE*, and Dan Feng, *Member, IEEE*

Abstract—Cloud computing has raised new challenges for the membership representation scheme of storage systems that manage very large data sets. This paper proposes DBA, a dynamic Bloom filter array aimed at representing membership for variable large data sets in storage systems in a scalable way. DBA consists of dynamically created groups of space-efficient Bloom filters (BFs) to accommodate changes in set sizes. Within a group, BFs are homogeneous and the data layout is optimized at the bit level to enable parallel access and thus achieve high query performance. DBA can effectively control its query accuracy by partially adjusting the error rate of the constructing BFs, where each BF only represents an independent subset to help locate elements and confirm membership. Further, DBA supports element deletion by introducing a lazy update policy. We prototype and evaluate our DBA scheme as a scalable fast index in the MAD2 deduplication storage system. Experimental results reveal that DBA (with 64 BFs per group) shows significantly higher query performance than the state-of-the-art approach while scaling up to 160 BFs. DBA is also shown to excel in scalability, query accuracy, and space efficiency by theoretical analysis and experimental evaluation.

Index Terms—Data management, fast index, membership representation, Bloom filter

1 Introduction

Cloud computing has become a popular paradigm for deploying and delivering services over the Internet [1], and it also raises new challenges for the membership representation scheme of storage systems that manage very large data sets [2], [3], [4]. For example, a key-value store needs to efficiently determine *whether a required data item exists* without having to exhaustively check all the elements. A straightforward approach to recording membership is to keep an ordered full index in memory. Once a membership query arrives, certain search algorithm will be activated to locate the target item. However, this primitive method faces two challenges when dealing with variable large data sets. First, it is cost-ineffective to maintain an ordered full index, as the logical/physical structure of the index must be frequently adjusted to accommodate the addition or deletion of elements. Commercial stores such as Amazon's Dynamo [5] and Microsoft's ChunkStash [6] allow complicated keys (i.e., opaque byte arrays and 20-byte SHA-1 hashes respectively) that can not be efficiently sorted. Second, as the amount of data grows, the whole index can become too large to be stored in the RAM in its entirety. HDFS [7] keeps the entire namespace for the distributed file system using 64 GB RAM in a special NameNode, which potentially limits the system scalability [8]. A storage node in Facebook's Haystack uses 48 GB RAM for a 9 TB RAID capacity without providing search features [9].

In general, we argue that indexing dynamically variable large data sets in storage systems requires the membership representation scheme to achieve (1) strong scalability to adapt the ever increasing amount of data, (2) high space efficiency to be contained in RAM for high access efficiency, (3) high query performance to resist the rising search complexity during the scaling-up process, (4) controllable query accuracy to resist errors introduced by compact data structures, (5) element location capability to inform the system where to locate the possible elements, and (6) element deletion capability to allow the storage system to recycle spaces.

Recently, a *compact hash table* is proposed in FlashStore [10] to index key-value pairs stored on flash. The hash table uses a (2-byte) compact signature combined with a (4-byte) pointer for each key (e.g., 20-byte SHA-1 hash), and it resolves collisions using a variant of cuckoo hashing [11] to achieve a high load factor. However, the capacity of the hash table must be predefined carefully according to the available flash and estimated number of keys. Another widely used technique is Bloom filter (BF) [12], which is a compact data structure that can efficiently record membership for a static set by hashing its elements into a fixed-length bit array. Clearly, both of the above techniques face challenges when dealing with a variable large data set, since either the existence of elements or the set cardinality can change over time. To the best of our knowledge, existing scalable membership representation schemes, such as *dynamic Bloom filters* (DBF) [13], *scalable Bloom filters* [14], [15], and *incremental Bloom filters* [16], generally lack a comprehensive consideration for the

- *J. Wei, K. Zhou, and D. Feng are with the Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Room B406, No. 1037, Luoyu Road, Wuhan 430074, Hubei, China.*
  E-mail: jianshengwei@gmail.com, {k.zhou, dfeng}@hust.edu.cn.
- *H. Jiang is with the Department of Computer Science and Engineering, University of Nebraska-Lincoln, 217 Schorr Center, 1101 T Street, Lincoln, NE 68588-0150. E-mail: jiang@cse.unl.edu.*

Manuscript received 17 Nov. 2012; revised 4 Feb. 2013; accepted 12 Feb. 2013; date of publication 6 Mar. 2013; date of current version 21 Feb. 2014.
Recommended for acceptance X. Tang.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2013.66

above requirements of storage systems, as they mainly focus on network applications.

To comprehensively consider the above requirements for a scalable fast index of membership representation, we propose DBA, a Dynamic Bloom filter array aimed at representing membership for variable large data sets in storage systems in a scalable way. DBA consists of groups of space-efficient Bloom filters, and its capacity can be extended by dynamically adding new BF groups. Within a group, homogeneous BFs are used and their structure is optimized at the bit level, so that dozens of BFs can be accessed in parallel to achieve high query performance. Using the compact BF structure introduces false positives, but the query accuracy can be effectively controlled by partially adjusting the error rate of the building-block BFs. Each BF is only responsible for representing an independent subset, which helps locate existing elements and confirm membership. Further, DBA supports element deletion by introducing a lazy update policy.

We extend our previous work [4] and comprehensively evaluate the DBA scheme in the MAD2 [17] deduplication storage system, which is a scalable high-throughput exact deduplication storage system that runs in the clustered backend of network backup services. In the MAD2 system, DBA records the membership of millions of chunk fingerprints and works as an in-memory scalable fast index, identifying unique data objects or indicating where to confirm a possible duplicate. Experimental results show that our multi-group-cardinality DBA approach can maintain a much higher query performance than the existing *dynamic Bloom filter* approach. For example, while scaling up to 160 BFs with an error rate threshold of $1/2^{14}$, a 64-group-cardinality DBA is able to query 4.46×10^5 items per second, a query performance that is about three times higher than DBF. The query accuracy of DBA can be effectively controlled by partially adjusting its member BFs with either enlarging their capacity or tightening their error rate. In our experiments, DBA reinserts over 4.52×10^5 items per second during the partial reconstruction process, which enables the fast index to erase deleted elements in batches. Mathematical analysis reveals that DBA is able to achieve all the above advantages over a BF with the same capacity and the same false positive rate at the expense of only a small fraction of more memory space of the latter.

The remainder of this paper is organized as follows: In the next section, we present the necessary background information to further motivate our DBA research. The DBA design is detailed in Section 3. Section 4 presents experimental results of our DBA prototype and mathematical analysis of the DBA model. Finally, Section 5 concludes the paper.

2 Motivation and Background

2.1 Challenges Arising from Variable Large Data Sets

Large scale data analysis applications have been widely deployed in the cloud environment [7], [8], [18]. One of the main challenges in managing a variable large data set is to provide a fast and scalable element query service. Distributed storage systems usually distribute metadata and/or data among multiple nodes based on (directory) subtree partitioning [19], [20], [21] or hash-based mapping [5], [22], [23]. In the former, a query process needs to traverse a tree-structured index to locate the target data item. The latter allows the query process to directly locate a responsible node/bucket, but the performance can be restricted by the sequential lookup within a node/bucket. Besides deterministic query schemes, probabilistic membership query approaches have also been proposed for metadata management purposes [24], [25]. In such approaches, a Bloom filter is used to record membership for each metadata server, and then the BFs are replicated among servers to construct a global probabilistic fast index. The above schemes focus on distributing and locating data elements among storage servers, but the overall query performance can also be restricted by the lookup efficiency of each server.

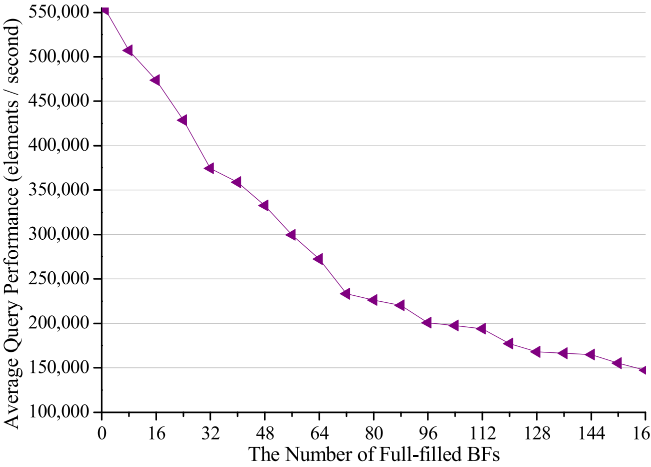


Fig. 1. The average query performance of a DBF.

DDFS [35] and ChunkStash [6] propose to build a fast index with a predefined capacity for each storage node by using BF and *compact hash table* respectively, but they lack considerations about the scaling-up process of the data sets. The limitations are obvious: first, the fast index is in risk of total reconstruction if the elements exceed its capacity; second, memory space is wasted if it takes a long time for the elements to fill the fast index; third, synchronizing the fast index with an on-disk backup copy is inefficient, since the fast index can be randomly updated by hash functions.

The *dynamic Bloom filter* scheme [13] uses multiple homogeneous BFs to index a scalable data set. This idea is similar to HBA [24] and G-HBA [25] in that they all create new BF as needed to extend the capacity, and it particularly emphasizes the use of homogeneous hash functions among BFs to reduce the computational overhead and enhance the scalability. However, we find that DBF faces a severe memory-access bottleneck as more BFs are added. Considering a DBF using k hash functions and r BFs, the memory-access complexity of querying an item can be expressed as $O(k \times r)$, which will increase linearly with the number of member BFs. Fig. 1 shows the experimental results in which a DBF is initialized using one BF with a capacity of 2^{19} elements. As the number of inserted elements increases to 83,733,597, totally 160 BFs are created and added to the DBF. Furthermore, 2,026,005,9277 membership queries are executed during the scaling-up process. For the stability of results, we measure the average query performance in a small window around the sampling point. As shown in the figure, the query performance goes down to about only 27 percent of the initial value as 160 BFs are created. The experiment is carried out in a real system where membership query is a regular operation, and the results show that the overall query performance can definitely be bottlenecked by the increasing memory-access complexity.

The proposed requirements of indexing variable large data sets in storage systems and the limitations of existing schemes motivate us to build a more flexible membership representation scheme. Next, we will briefly review the Bloom filter and its variations to present the necessary background information.

2.2 The Principles of Bloom Filters

A Bloom filter [12] for representing a static set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements consists of an array of m bits and a set of k independent hash functions $h_1, \ldots, h_k$ with range $\{1, \ldots, m\}$. At the beginning, all the bits of the array are initialized to 0, and then the bits $h_i(x)$ $(1 \leq i \leq k)$ are set to 1 for each element $x \in S$. To determine if an item y belongs to S, we just need to check whether all the bits $h_i(y)$ are set to 1. If not, y is definitely not a member of S. Conversely, y is assumed as a member of S with some error probability. This kind of false positive is caused by the hash collisions, which imply that the bits $h_i(y)$ can be occasionally set by some other elements indeed belonging to S. The probability that a fresh item is falsely identified as a member of a set by the associated BF is called the false positive rate or error rate. Fortunately, the error rate can be controlled by many tricks, and the high space efficiency and the high query performance make a BF attractive to many applications [24], [25], [26], [27].

Given m, n and k $(k \cdot n < m)$, the false positive rate of a BF can be derived as

$$f_{\mathrm{BF}} = (1 - (1 - 1/m)^{kn})^k \approx (1 - e^{-kn/m})^k.$$

It is shown that f_{BF} can be minimized to an expected value $(1/2)^{\ln 2 \cdot (m/n)}$ when $k = \ln 2 \cdot (m/n)$ [27]. Further, Kirsch and Mitzenmacher [28] propose that the computational complexity of the k hash functions can be reduced by using the technique of *double hashing*. For a static set with known cardinality n, if f_{BF} must be restricted to a threshold ε with minimal space overhead, the optimal number of hash functions should be

$$k_{\mathrm{opt}} = \log_2(1/\varepsilon), \tag{1}$$

and the required minimal space (in bits) is

$$m_{\min} = \log_2 e \cdot k_{\mathrm{opt}} \cdot n = \log_2 e \cdot \log_2 (1/\varepsilon) \cdot n. \tag{2}$$

It is easy to see that the error rate f_{BF} will not reach ε until all the n elements are inserted into the BF, thus n is also called the designed capacity and ε is also referred to as the target error rate, denoted as F_{BF}.

In particular, there is an interesting implementation of *segmented* BF (also referred to as *partitioned* BF) [27], [28], which consists of an array of m bits and k hash functions with each mapped to a range of m/k consecutive bits independently. This structure enables some applications, such as a hardware-based router [29], to access the bit array in parallel through the logic circuit. With the capacity of n and the space of m bits, the error rate of a *segmented* BF is

$$f'_{\mathrm{BF}} = (1 - (1 - k/m)^n)^k \approx (1 - e^{-kn/m})^k.$$

Clearly, f'_{BF} is slightly larger than f_{BF}, but both of them have the same asymptotic error rate. We will consider using *segmented* BFs as necessary without discussing the negligible error rate deviation in our solution.

Putze et al. [30] propose another variant called *blocked Bloom filter* (*blo*) that consists of b 512-bit blocks, where each block can fit into one cache-line of the CPU. For each element, *blo* uses the first hash function to select the accommodating block and the rest $k-1$ hash functions to set or test bits within the block. As a result, *blo* incurs at most one cache miss in querying an element, which benefits the insert/query performance. However, it can be a challenge for a *blo* to scale up and support element deletion.

Many improved schemes [13], [14], [15], [16], [31] have been proposed to represent dynamic sets, especially scalable sets. Due to the space limit, the detailed review is provided in the online supplemental material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.66.

In general, the existing improved Bloom filters mainly focus on network applications, and they lack considerations about how to index variable large data sets in storage systems, which motivates our DBA design that is detailed next.

3 The DBA Design

This section presents the design of Dynamic Bloom filter array, to carry out a general membership representation scheme and meet the requirements of a practical scalable fast index for variable large data sets in both distributed and single-node storage systems.

Fig. 2 shows a general framework of using DBA as a scalable fast index in a distributed storage system. In a storage cluster (Fig. 2a), a low-accuracy global DBA that summarizes the data membership of all the storage nodes can be used to filter out invalid queries or route the survived queries to the responsible nodes. The global DBA can be cooperatively maintained by multiple metadata servers if it becomes too large. Some distributed storage systems can also employ DHT-based routing, however, the studies of HBA [24] and G-HBA [25] have shown the potential benefit of using BF-based techniques in ultra-large scale file systems.

In a specific storage node (Fig. 2b), a high-accuracy DBA that maintains high space efficiency can be fully loaded in RAM to significantly improve the access efficiency. On receiving query requests, the DBA scheme

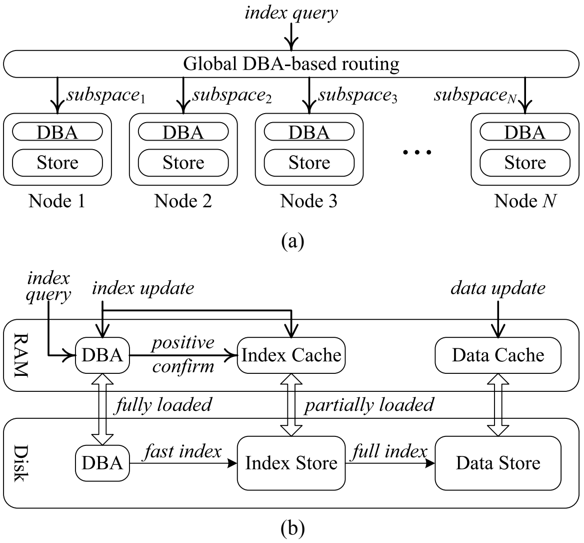


Fig. 2. Using DBA as a scalable fast index. (a) Using DBA in a storage cluster. (b) Using DBA in a single storage node.

will be first checked to quickly filter out nonexistent items. For possible existing elements, the DBA scheme directs the query process to search the most relevant subsets to confirm membership through an in-memory cache that maps to the on-disk full index. If a fresh item arrives, its metadata will be inserted into both the DBA and the full index, and the data content will be written to the on-disk data store accordingly through an in-memory data cache. As more fresh data is written, DBA scales along with the on-disk full index, and only the updated components (i.e., BFs) of the DBA need to be synchronized to the on-disk copy. Since the DBA structure is optimized at the bit level, which will be detailed in Section 3.1, it can effectively maintain high query performance while the data sets scale up in size.

3.1 The DBA Construction

A DBA is constructed of groups of compact BFs for space efficiency. BFs are homogeneous within each group to share the same hash functions and reduce computational overhead. Existing full-filled BFs will be sealed for query only and new BF can be created dynamically to expand the capacity.

Specifically, a DBA significantly differs from a *dynamic Bloom filter* in the bit layout. As shown in Fig. 3a, the r BFs sharing k hash functions in a DBF are independent of one another in space. To query an item, the r BFs will be checked sequentially until a positive or all negatives are produced, and the memory-access complexities are $O(k \times r/2)$ for a positive result and $O(k \times r)$ for a negative result respectively, according to [13, Algorithm 1] and shown on this page. Note that a lot of memory bandwidth is wasted while reading only one bit in each access. Fig. 3b shows the optimized bit layout of a DBA. In each group, bits belonging to g different BFs but with the same offset are mapped and stored together so that they can be simultaneously read or written in a single memory access. Algorithm 2 is an improved query process of Algorithm 1, and its memory-access complexities are $O(k \times r/(2g))$ for a positive result and $O(k \times r/g)$ for a negative result. Obviously, the memory access efficiency of a DBA is $O(g)$ times better than that of a DBF.

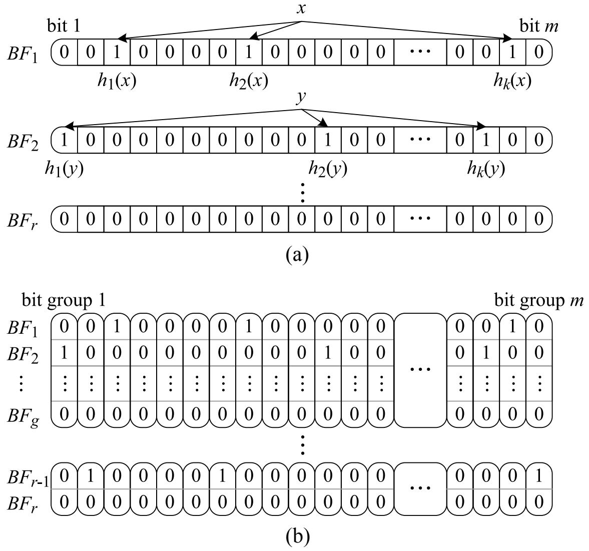

x
bit 1
bit m
BF_1 0 0 1 0 0 0 0 1 0 0 0 0 0 ··· 0 0 1 0
$h_1(x)$
$h_2(x)$
$h_k(x)$
y
BF_2 1 0 0 0 0 0 0 0 0 0 1 0 0 ··· 0 1 0 0
$h_1(y)$
$h_2(y)$
$h_k(y)$
BF_r 0 0 0 0 0 0 0 0 0 0 0 0 0 ··· 0 0 0 0
(a)
bit group 1
bit group m
BF_1 0 0 1 0 0 0 0 1 0 0 0 0 0 ··· 0 0 1 0
BF_2 1 0 0 0 0 0 0 0 0 0 1 0 0 ··· 0 1 0 0
BF_g 0 0 0 0 0 0 0 0 0 0 0 0 0 ··· 0 0 0 0
$BF_{r\text{-}1}$ 0 1 0 0 0 0 1 0 0 0 0 0 0 ··· 0 0 0 1
BF_r 0 0 0 0 0 0 0 0 0 0 0 0 0 ··· 0 0 0 0
(b)

Fig. 3. A bit-layout comparison between DBF and DBA, where a DBF stores the bits in a row-major order, while a DBA stores the bits in a column-major order to provide parallelism in memory access. (a) The bit layout of a DBF. (b) The bit layout of a DBA.

Algorithm 1. Query(x) in a DBF.

```
Require: x is not null
Note: hash_j(x) will only be calculated once, and the results can
be cached for homogeneous BFs.
for( i = 1; i <= r; i++ )
    counter = 0;
    for( j = 1; j <= k; j++ )
        if( BF[ i ][ hash_j(x) ] == 0 )
            break;
        else
            counter++;
    if( counter == k )
        return true;
return false;
```

Algorithm 2. Query(x) in a DBA.

```
Require: x is not null.
Note: BFs in different groups are assumed to be homogeneous
to facilitate the discussion.
for( i = 1; i <= $\lceil r/g \rceil$; i++ )
    bit_vector = 0xFF…F;
    for( j = 1; bit_vector && j <= k; j++ )
        bit_vector &= BfG[ i ][ hash_j(x) ];
    if( bit_vector != 0 )
        return true;
return false;
```

The maximum number g of BFs in a group is called group cardinality, which determines the parallelism in querying multiple BFs. Zhang and Guan [32] suggest that g should not exceed the bit width of a CPU register, so that a bit group can be hold in its entirety by a (32-bit) word. In our design, if g exceeds the bit width of a CPU register, a bit group will be vectorized to leverage the vector processing units [34] and the on-chip caches of modern CPUs to achieve high query performance. In practice, g is recommended to be a fraction of r (i.e., the total number of BFs)

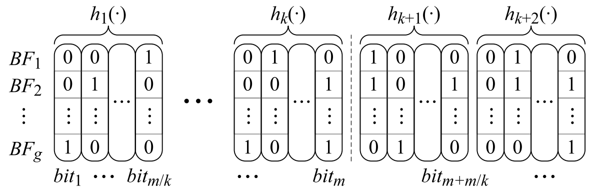


Fig. 4. Using segmented BF to facilitate error rate shifting.

for the consideration of controlling query accuracy, which will be discussed in Section 3.2. For example, define $g/r \in [1/8, 1/4)$, then a new BF group can be created as necessary when $g/r \geq 1/8$, and BF groups will be pair-wisely combined to increase the group cardinality when $g/r < 1/8$. This combining can be done efficiently by a background thread as only memory-copy operations are involved. A BF group can be created at once or extended on demand to accommodate new BFs. The latter can achieve better memory efficiency but at the cost of more combining operations than the former.

3.2 Control the Query Accuracy

A DBA can generate false positives as it is constructed of compact BFs. Considering a DBA that is initially designed to hold r BFs with each has a capacity of c_{BF}, the designed capacity of the DBA is $c_{\mathrm{DBA}} = c_{\mathrm{BF}} \times r$. Let ε denote the error rate threshold, F_{DBA} the target error rate of the DBA, and F_{BF} the target error rate of each BF, we have

$$F_{\mathrm{DBA}} = 1 - (1 - F_{\mathrm{BF}})^r = \varepsilon \tag{3}$$

and

$$F_{\mathrm{BF}} = 1 - (1 - F_{\mathrm{DBA}})^{1/r} = 1 - (1 - \varepsilon)^{1/r}. \tag{4}$$

The problem here is that the actual error rate can exceed ε if the number of elements exceeds c_{DBA}.

Formula (3) suggests that there are two methods to control the error rate. The first is to enlarge the capacity of the BFs while keeping both r and F_{BF} constant, which can be implemented by reconstructing BFs in groups. We describe the main steps as follows:

1. If the number of BFs reaches r, find a group of BFs with capacity c_{BF} and reconstruct them using a larger capacity $c'_{\mathrm{BF}} = t \times c_{\mathrm{BF}}$ in the background, where t is an integer larger than or equal to 2. As a result, $(t-1) \times g \times c_{\mathrm{BF}}$ more elements can be accommodated.
2. If all the r BFs are enlarged, replace c_{BF} with c'_{BF} and update $c'_{\mathrm{BF}} = t \times c_{\mathrm{BF}}$.
3. Repeat step 1 and step 2 until the data set stops growing.

The second method is to increase r by suppressing F_{BF}. According to Formula (4), if r is increased to $r' = t \times r$, the target error rate of each BF will be

$$F'_{\mathrm{BF}} = 1 - (1 - F_{\mathrm{DBA}})^{1/r'} = 1 - (1 - \varepsilon)^{1/r'}. \tag{5}$$

The main steps of this method are detailed as follows:

1. If the number of BFs reaches r, find a group of BFs constructed with parameter F_{BF} and reconstruct them using F'_{BF} in the background. As a result, $(t-1) \times g$ more BFs can be created with a tighter target error rate F'_{BF}.
2. If all the BFs are constructed with parameter F'_{BF}, replace r with r' and update $r' = t \times r$, and replace F_{BF} with F'_{BF} and update F'_{BF} using Formula (5).
3. Repeat step 1 and step 2 until the data set stops growing.

Both of the methods employ background partial reconstruction to make sure that the actual error rate of a DBA is kept below the threshold while using a larger parameter t can reduce the reconstruction frequency. The chosen BF group can still serve membership queries until the reconstruction is completed in the newly allocated memory space. During the error rate adjustment period, a DBA allows at most two kinds of heterogeneous BFs, but hash functions can still be shared among homogeneous BFs and a group of BFs can always be queried in parallel to ensure high performance. Note that Algorithm 2 should be slightly modified to consider the case of heterogeneous BFs. Partial reconstruction has been implemented by re-inserting the raw elements of chosen BFs into reconfigured data structures in our prototype system, and we will present the experimental results in Section 4.4.

Furthermore, the second method of tightening the target error rate essentially requires increasing the number of hash functions for a BF group, which can be simplified by using *segmented* BFs. As Fig. 4 shows, a group of g *segmented* BFs initially share k hash functions with each being independently mapped to m/k bits, and the target error rate of each *segmented* BF being $F_{\mathrm{BF}} = (1/2)^k$. To tighten the target error rate to $F'_{\mathrm{BF}} = F_{\mathrm{BF}}/4$, we just need to add another two hash functions and insert the related elements into the incremental bit vectors. This is more efficient than rebuilding the whole BF group that must calculate $k+2$ hash functions for each element.

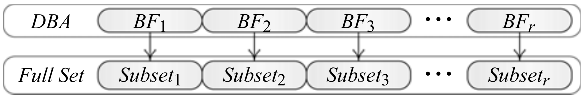


Fig. 5. Using DBA to locate elements.

3.3 Accelerate Element Location

In some storage systems, false positives are absolutely unacceptable and must be identified by checking the full set. For example, in a deduplication storage system [35], possible duplicate items must be confirmed by querying the on-disk full index, for otherwise the fresh data can be falsely eliminated. However, frequently searching an on-disk index can reintroduce the performance bottleneck.

Taking advantage of DBA's scalability, we propose to use different BFs to represent independent subsets (both in a distributed environment and in a single storage node), as shown in Fig. 5. A possible existing element can be located by simply searching the associated subset of a BF that generates a positive for it. Considering a DBA containing r BFs, the search range can be directly reduced to $1/r$ if only one of the BFs generates positive. Algorithm 2 can be easily modified to return the index number of a BF that generates a positive for the queried item. If multiple positives are generated for an item, the query process should check the corresponding subsets one by one until the item is finally identified. According to Formula (4), the maximum probability that i $(1 < i \leq r)$ BFs generate false positive for a given item can be derived as

$$(F_{\mathrm{BF}})^i = (1-(1-\varepsilon)^{1/r})^i \ < \varepsilon^i,$$

which is small and will not cause significant overhead.

In practice, the BFs in a DBA are created along with the subsets as the full set scales up. If a subset is relatively stable, its index structure can be further optimized to enhance the lookup efficiency [17].

3.4 Support Element Deletion

The delete operation is important for a storage system to recycle storage capacity. Since a DBA is constructed of groups of BFs, false negtives will be introduced if elements are deleted by directly clearing the corresponding bit cells. Considering the fact that large storage systems usually delete stale data in batches in off-peak time, we introduce a lazy update policy for the DBA scheme to support element deletion. Different from a DBF that maintains counters for *all the representing cells* [33], a DBA associates *each BF* with two counters that record the number of represented elements and the number of stale elements respectively. Whenever an element is deleted from a subset, the associated BF will increment the counter of stale elements. Only when the stale elements in a BF reach a predefined proportion, will a BF be refreshed to reflect the real-time membership of the subset, which implicitly eliminates deleted elements in a batch.

A negative effect of the lazy update policy is that a DBA can generate false positives while querying recently deleted elements. However, there will be no false negtives, and the target false positive rate is still well controlled. A storage system can control the refresh frequency of BFs by adjusting the allowed proportion of stale elements. Considering that a DBA can scale up and contain hundreds even thousands of BFs, refreshing a BF will not cause significant performance degradation, and the experimental results will be given in Section 4.4.

Furthermore, partially filled subsets can be combined by taking the *OR* of their associated BFs if their elements can be held by just one subset.

4 Evaluation and Analysis

We implement and evaluate our DBA scheme in the MAD2 [17] deduplication storage system. MAD2 distributes both file metadata and chunk contents among clustered storage nodes according to their fingerprints (i.e., 20-byte SHA-1 hash). Files or chunks with the same fingerprint will be identified as duplicates and only one instance of duplicates is stored. Specifically, the average chunk size is 4 KB, so that a 10 TB chunk store can hold at most 2.5×2^{30} chunks. To reduce the RAM requirement of maintaining a full index and improve the query efficiency, a DBA is used as a fast index to record the membership of chunk fingerprints in each storage node, and the full index is stored on disk and accessed through an efficient in-memory cache.

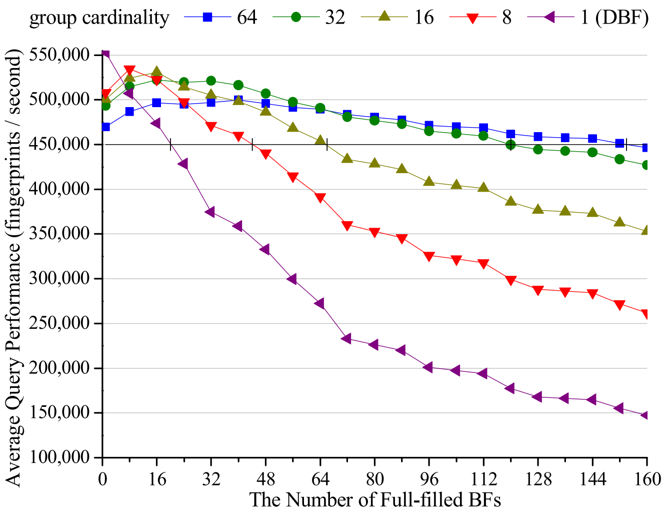


Fig. 6. Average query performance of a DBA under five different group cardinalities.

The experimental data set is a backup data set that was collected from an engineering group in a span of 31 days. Totally, there are 12.1 millions of files that amount to 6.0 TB. In the MAD2 system, these data generates 2,026,005,927 chunks and is finally deduplicated into 83,733,597 unique chunks totaling 367.7 GB in size. We disable the file level deduplication and use the fingerprints of these chunks to fill and evaluate a DBA in a chunk storage node. The system configuration includes a 32-bit Windows operating system, a quad-core CPU running at 2 GHz, 16 GB RAM, 2 gigabit network interface cards, and 16×1 TB hard disks organized as a RAID-5 partition.

4.1 Query Performance

According to our earlier analysis (in Section 3.1), the query performance of a DBA can be mainly affected by its group cardinality. In this section, we construct five different configurations of DBA (i.e., 1-, 8-, 16-, 32-, and 64-group-cardinality DBA) and measure their query performance in the MAD2 chunk-level deduplication environment. The error rate threshold of each DBA is specified as $1/2^{14}$ and the BFs of each DBA are designed to hold at most 2^{19} elements. In particular, the 1-group-cardinality DBA is functionally equivalent to a DBF, which is used for comparison between DBF and DBA, and it also returns the index number of the BF that generates a positive for the target chunk fingerprint.

Since the response time of querying a single chunk fingerprint is extremely short and can be disturbed by the instantaneous system workload level and the membership of the fingerprint, instead, we trace the numbers of the queried chunk fingerprints and the time stamps of the BF full-filled events to evaluate the average query performance.

As shown in Fig. 6, when there is just one BF being filled, the DBF scheme (i.e., the 1-group-cardinality DBA) shows the best average query performance of 5.5×10^5 fingerprints per second (fps), which is about 4.5×10^4 fps higher than the eight-group-cardinality DBA scheme. In general, we find that a DBA with larger group cardinality shows worse performance at the beginning when there is only one or a

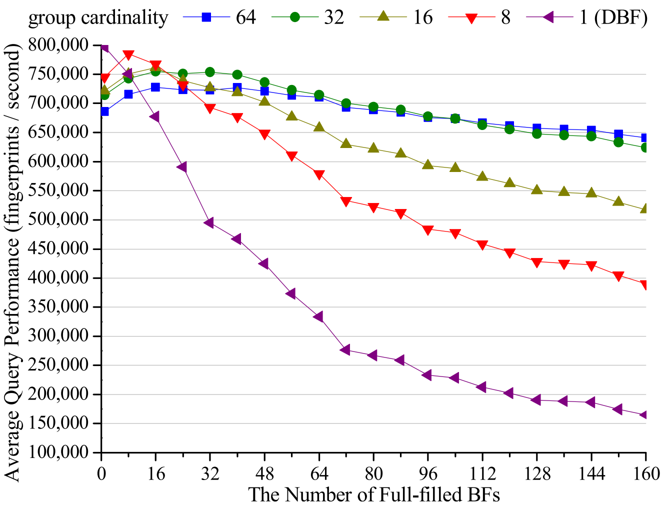


Fig. 7. Average query performance of a low-accuracy DBA that works as a global fast index in a distributed storage system.

small number of full-filled BFs. This reflects the fact that a DBA with smaller group cardinality incurs less management overhead (i.e., initialization of data structures, etc.) and can be more efficiently accessed through the CPU cache mechanism when the number of BFs is small. The results also suggest that the group cardinality of a DBA should increase slowly according to the number of required BFs, as discussed in Section 3.1. When the number of BFs grows to 16, the multi-group-cardinality DBA starts to outperform DBF in the average query performance. As the fingerprint set further scales up and more BFs are created, the average query performance of all the DBAs shows a similar declining trend, where a DBA with larger group cardinality declines slower than one with smaller group cardinality. When the number of BFs reaches 160, the average query performance of the DBF scheme declines to 1.5×10^5 fps, in clear contrast to the 64-group-cardinality DBA that maintains a much higher performance at 4.5×10^5 fps.

In particular, since MAD2 runs on a 32-bit operating system, the query operation of the 64-group-cardinality DBA scheme actually relies on the vectorized 32-bit calculation. Fig. 6 shows that the 64-group-cardinality DBA scheme outperforms the 32-group-cardinality DBA when there are more than 72 BFs. This result suggests that even if the group cardinality exceeds the bit width of the processing unit, large group cardinality is still effective in improving the RAM access efficiency and maintaining the query performance.

Although it is a natural choice to employ a DHT-based method to distribute chunk contents among multiple storage nodes in MAD2, we also evaluate the performance impact of using DBA as a global fast index in a simulated distributed storage system by relaxing the error rate threshold to $1/2^7$ that defines low-accuracy DBAs. Fig. 7 shows the average query performance of the DBAs with five different group cardinalities. Comparing to the high-accuracy DBAs, the low-accuracy DBAs show much higher average query performance that ranges from 6.9×10^5 fps to 8.0×10^5 fps at the beginning. While the average query performance declines as more BFs are created, we find that the 64-group-cardinality DBA maintains its average query performance at 6.4×10^5 fps when there are 160 BFs, which is about four times higher than that of the DBF (i.e., the 1-group-cardinality DBA) at 1.6×10^5 fps. In a real distributed environment, such as an ultra large-scale distributed file system ($\geq$ Petabytes or even Exabytes) [25], the metadata server (group) can maintain one or several BFs for each storage node according to their capacities. If new storage nodes are added, the metadata server (group) only needs to create new BFs accordingly.

4.2 Scalability

The experimental results of query performance show that DBA effectively maintains a high query performance by enabling parallel access and vectorized calculation over a group of BFs. Since the query performance tends to decline linearly as the number of BFs increases, we can evaluate the scalability of a DBA by comparing it to a DBF with the same BF composition. As Fig. 6 shows, when the query performance is 4.5×10^5 fps, there are 20 BFs in DBF, and 44, 66, 120 and 154 BFs in DBAs with 8-, 16-, 32-, and 64-group-cardinality respectively. Obviously, a DBA with larger group cardinality shows much better scalability in practice.

Now we derive a theoretical upper bound for the scalability of a DBA. Suppose that a DBF can scale up to r BFs while maintaining its query performance at or above the predefined threshold φ and the percentage of CPU time used to query the DBF is q in a real system, the speedup of querying a g-group-cardinality DBA over DBF with the same BF composition can be inferred according to Amdahl's law [36] as $Speedup = [(1-q)+(q/g)]^{-1}$. Thus, the DBA can hold at most $r \times g$ BFs in theory with the overall query performance no less than φ. In practice, the multi-group-cardinality DBA can introduce additional management overhead that slightly limits its scalability. On the other hand, the scalability of a DBF can be more severely limited by its significantly increased false positive rate (see Section 3.2).

4.3 False Positive Rate

This section evaluates the error rate control capability of the DBA scheme. Suppose that the maximum cardinality of the fingerprint set is unknown initially, we start with a DBA with a target error rate of $1/2^{14} \approx 6.1 \times 10^{-5}$ and a maximum capacity of 64×2^{19} elements. Let each BF hold 2^{19} elements, there are at most 64 BFs in the DBA and the target error rate of each BF can be derived as 9.5×10^{-7} based on Formula (4).

The real error rate is measured by counting the false positives in statistics windows near the sample points. As Fig. 8 shows, the real error rate grows to 5.7×10^{-5} and is close to the threshold when the number of fingerprints reaches the designed capacity of the DBA. Without adjusting the parameters, 160 BFs are created to hold all the fingerprints, and the uncontrolled error rate finally grows to 1.6×10^{-4} that is about 2.6 times larger than the threshold.

The first method to control the error rate of a DBA is to enlarge the designed capacity of BFs with their target error rate unchanged. When the number of fingerprints reaches 64×2^{19}, a group of 32 BFs are rebuilt to enlarge their capacity from 2^{19} to 3×2^{19}, which allows 64×2^{19} more fingerprints to be inserted. Similarly, when the number of elements

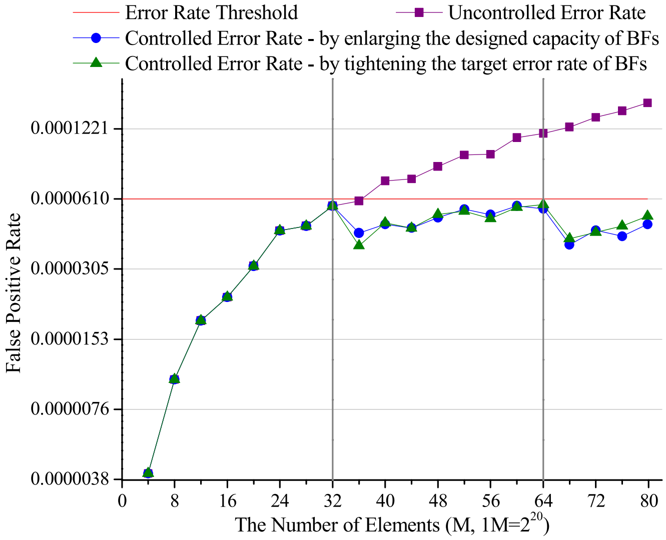


Fig. 8. Error rate control capability of a DBA.

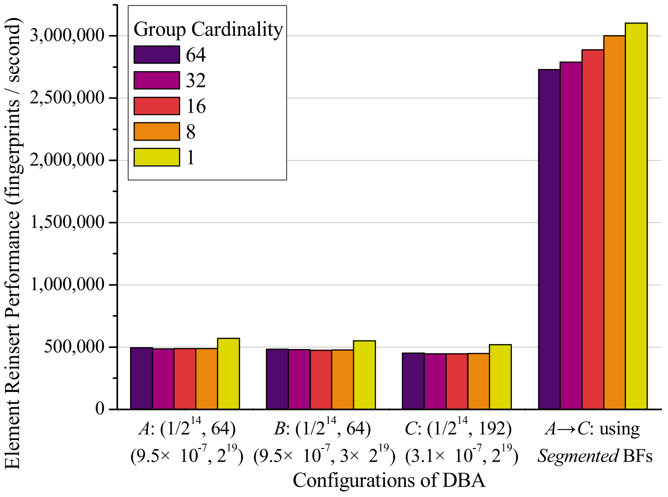


Fig. 9. Partial reconstruction efficiency of a DBA.

grows to 128×2^{19} that fills the DBA again, another group of 32 BFs are rebuilt with a larger capacity of 3×2^{19} elements. As shown in Fig. 8, the error rate of the DBA is effectively controlled below the threshold and finally shows 4.8×10^{-5} when all the fingerprints are represented.

The second method to control the error rate is to tighten the target error rate of BFs with their capacity unchanged. When the number of used BFs reaches 64, a group of 32 BFs are rebuilt to shift their target error rate from 9.5×10^{-7} to 3.1×10^{-7}, which allows 64 more BFs with the tighter target error rate to be created. Similarly, when the number of BFs reaches 128, which means that 128×2^{19} elements are inserted, another group of 32 BFs are rebuilt with a target error rate of 3.1×10^{-7}. Fig. 8 shows the error rate has also been successfully controlled below the threshold by using this method, and the error rate is 5.1×10^{-5} when all the elements are inserted.

The results show that both of the error rate control methods are effective in dealing with the real-world data set.

4.4 Partial Reconstruction Efficiency

A DBA will be partially reconstructed to control its error rate or delete stale elements. This section presents the partial reconstruction efficiency of a DBA under different configurations by measuring the element reinsert performance in terms of fingerprints per second.

As shown in Fig. 9, a DBA is initially designed with Configuration A: a target error rate of $1/2^{14}$ (6.1×10^{-5}), holding at most 64 BFs, and the designed capacity of each BF is 2^{19}. As a result, the target error rate of each BF can be derived as 9.5×10^{-7}, according to Formula (4), and the number of required hash functions is 20, according to Formula (1). The element reinsert performance is 5.7×10^5 fps for the 1-group-cardinality DBA (i.e., equivalently, DBF) and then decreases to about 4.9×10^5 fps as the group cardinality increases due to higher group management overhead.

Configuration B is derived from Configuration A by enlarging the designed capacity of each BF to 3×2^{19}. The element reinsert performance is between 4.8×10^5 fps and 5.5×10^5 fps for different group cardinalities, which is slightly lower than Configuration A's performance because the DBA has to access a range of space that is three times larger to reinsert an element.

Configuration C allows the DBA to hold at most 192 BFs, and the target error rate of each BF is tightened to 3.1×10^{-7} by using 22 hash functions. If a BF under this configuration is to be totally reconstructed, the element reinsert performance will be between 4.5×10^5 and 5.2×10^5 for different group cardinalities. Because the complexity of both the hash calculation and the RAM accesses has been increased, the results are further lower than those of Configuration A and Configuration B. However, if the DBA is configured to consist of *segmented* BFs, its error rate can be easily tightened by adding two more segments for each BF, along with the incremental hash functions. The results show that the element reinsert performance is between 2.7×10^6 fps and 3.1×10^6 fps for different group cardinalities, which is significantly higher than Configurations A-C that use standard BFs. While these results indicate that the element reinsert performance is somewhat negatively affected by the group management overhead introduced by the increasing group cardinality, the group management overhead of a DBA is generally negligible and the benefit of improving query performance far outweigh the overhead.

In practice, the partial reconstruction process can be executed by a background thread to ensure that the DBA maintain a reasonably high query performance. In particular, in a distributed environment that employs DBA as a global fast index, the building-block BFs can be reconstructed by their corresponding storage nodes and then grouped by the metadata server (group), which can prevent a performance bottleneck from being formed while processing metadata.

4.5 RAM Capacity Requirement

Considering the fact that the required space to construct a DBA depends essentially on the error rate threshold and the designed capacity, this section presents the RAM consumption of a DBA through mathematical analysis.

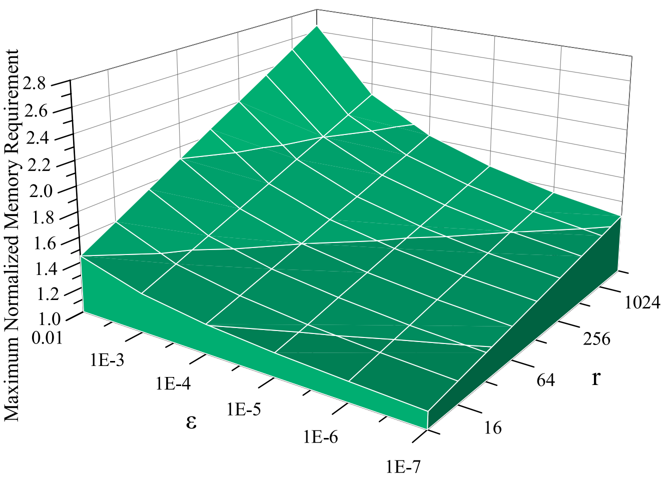


Fig. 10. Maximum memory requirement of a DBA (normalized to a BF).

For simplicity, first we consider a variable large data set S with a known maximum cardinality $c_{\max}$. Suppose that an error rate threshold ε is specified, we evaluate the memory requirement of a DBA, normalized to that required by a single BF that cannot be scaled at all. According to Formula (2), to represent membership for S, the number of bits required by a single BF is

$$m_{\text{BF}} = \log_2 e \cdot \log_2(1/\varepsilon) \cdot c_{\max}.$$

Alternatively, considering a DBA consisting of r homogeneous BFs, according to Formula (4), the error rate threshold of each member BF is $\varepsilon' = 1 - (1 - \varepsilon)^{1/r}$, and the number of bits required by each member BF is

$$m'_{\text{BF}} = \log_2 e \cdot \log_2(1/\varepsilon') \cdot c_{\max}/r.$$

If S finally grows to its maximum cardinality, the number of bits consumed by the DBA is

$$m_{\text{DBA}} = m'_{\text{BF}} \cdot r = \log_2 e \cdot \log_2(1/\varepsilon') \cdot c_{\max}.$$

Dividing m_{DBA} using m_{BF} and after some algebra manipulation, we have

$$m_{\text{DBA}}/m_{\text{BF}} = \log_\varepsilon(1 - (1 - \varepsilon)^{1/r}) \geq 1.$$

Thus, the maximum normalized memory requirement is independent of set cardinality, and a DBA can consume more memory space than a single BF once the represented set scales to its maximum cardinality.

Fig. 10 shows the maximum memory requirement of a DBA normalized to a single BF with the same capacity and the same error rate threshold. We can see that for the given error rate threshold, a DBA consisting of more BFs also requires more space to restrict its error rate. On the other hand, the maximum normalized memory requirement decreases as the error rate threshold becomes tighter. It indicates that if the error rate threshold is sufficiently small, a DBA consisting of dozens or even hundreds of BFs will consume only a fraction of more memory space than a single BF. For example, to achieve a target error rate of $1/10^5$, a DBA consisting of 1,024 BFs requires only 1.6 times of the memory of a single BF.

On the other hand, it usually takes a long time for a variable large data set to reach its maximum cardinality with a certain probability. Denote the rate of the mean cardinality and the maximum cardinality of a set as p, the expected memory requirement of a DBA can be derived as

$$m'_{\text{DBA}} = m_{\text{DBA}} \cdot p = \log_2 e \cdot \log_2(1/\varepsilon') \cdot c_{\max} \cdot p,$$

and the expected normalized memory requirement is

$$m'_{\text{DBA}}/m_{\text{BF}} = p \cdot \log_\varepsilon(1 - (1 - \varepsilon)^{1/r}).$$

If the set cardinality has a long tailed distribution, it is possible that the upper bound is dozens of times larger than the mean cardinality and the expected memory requirement of a DBA can be conversely much smaller than that of a single BF. Previous research [33] shows that memory spaces can be saved under five popular set cardinality distributions: uniform distribution, normal distribution, random Zipf distribution, maximum Zipf distribution and minimum Zipf distribution.

For a variable large data set without known maximum cardinality, a DBA is able to enlarge its capacity as necessary with the error rate effectively controlled and the memory consumption will increase as needed.

In general, a DBA inherits the high space-efficiency of a BF, and it is more flexible than a BF in allocating memory space, controlling error rate and supporting membership update. In particular, if an in-memory fast index must be periodically synchronized to an on-disk copy to prevent accidental data loss, only the updated BFs have to be rewritten for a DBA.

4.6 Comparison between DBA and DBF

We compare our DBA scheme to the DBF scheme in this section, since both of them are aimed at dynamic sets and employ homogeneous BFs to avoid computational bottlenecks of hash operations. The most significant difference lies in that a DBA optimizes the bit layout for each BF group to maintain high query performance while a DBF is more likely to encounter the RAM access bottleneck while scaling up.

Table 1 (see supplemental material, available online) lists the comparison details in terms of scalability, query efficiency, error rate control, element location, element deletion and RAM consumption. Theoretical analysis shows that both the memory-access efficiency and the scalability of a DBA are $O(g)$ times better than those of a DBF. Experimental results indicate that using larger group cardinality indeed enables a DBA to significantly outperform a DBF in query performance while representing a variable large data set. A DBA can effectively control its error by either enlarging the designed capacity or tightening the target error rate of member BFs, while a DBF lacks a proactive solution. By using each member BF to represent an independent subset, a DBA consisting of r BFs can narrow the searching range to

$1/r$ if only one positive is generated for a target item. In contrast to a DBA that supports element deletion by employing a lazy update policy, a DBF [33] proposes to replace constructing BFs using *counting Bloom filters* [37] that consume more RAM space and more computational overhead. In general, a DBA outperforms a DBF and is more flexible and more space-efficient in representing variable large data sets.

5 Conclusions

This paper presents the DBA scheme to represent membership for variable large data sets in storage systems. A DBA consists of groups of homogeneous Bloom filters that can be created dynamically as needed. Inside each group, the bit layout is optimized to improve RAM access efficiency, so that multiple BFs can be queried in parallel. The query accuracy of a DBA can be proactively controlled by adjusting the error rate of its member BFs. By associating each BF with an independent subset, a DBA is capable of locating possible existing elements. Adopting a lazy update policy, a DBA also effectively supports element deletion, an important operation in data management.

The DBA scheme is prototyped and evaluated in the MAD2 deduplication storage system. Experimental results show that a DBA can maintain high query performance while using large group cardinality and the query accuracy can be effectively controlled by either enlarging the capacity or tightening the target error rate of its member BFs. Moreover, the element reinsert performance is high enough to support error rate adjustment and reflect membership update. Theoretical analysis reveals that a DBA consumes only a fraction of more memory space than a single BF with the same capacity and the same target error rate to achieve its high flexibility, scalability and performance. In general, a DBA outperforms existing schemes and is more applicable to representing variable large data sets as a scalable fast index in storage systems.

Acknowledgments

This work was supported in part by the National Basic Research Program (973 Program) of China under Grant No. 2011CB302305, the National Natural Science Foundation of China under Grant No. 61232004 and No. 61025008, and the US National Science Foundation (NSF) under Grants NSF-IIS-0916859, NSF-CCF-0937993, NSF-CNS-1016609 and NSF-CNS-1116606. Dr. Ke Zhou is the corresponding author. The authors are grateful to the anonymous reviewers for their constructive comments.

References

[1] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," Technical Report UCB/EECS-2009-28, EECS Dept., U.C. Berkeley, Feb. 2009.

[2] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud Computing: State-of-the-Art and Research Challenges," *J. Internet Services and Applications*, vol. 1, no. 1, pp. 7-18, 2010.

[3] S. Tomazic, V. Pavlovic, J. Milovanovic, J. Sodnik, A. Kos, S. Stancin, and V. Milutinovic, "Fast File Existence Checking in Archiving Systems," *ACM Trans. Storage*, vol. 7, no. 1, article 2, June 2011.

[4] J. Wei, H. Jiang, K. Zhou, and D. Feng, "DBA: A Dynamic Bloom Filter Array for Scalable Membership Representation of Variable Large Data Sets," *Proc. 19th IEEE Int'l Symp . Modeling, Analysis, and Simulation of Computer and Telecomm. Systems (MASCOTS)*, pp. 466-468, July 2011.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," *Proc. 21st ACM Symp. Operating Systems Principles (SOSP)*, pp. 205-220, Oct. 2007.

[6] B. Debnath, S. Sengupta, and J. Li, "ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory," *Proc. USENIX Ann. Technical Conf. (ATC)*, pp. 215-229, June 2010.

[7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," *Proc. 26th IEEE Symp. Mass Storage Systems and Technologies (MSST)*, May 2010.

[8] C. Maltzahn, E. Molina-Estolano, A. Khurana, A.J. Nelson, S.A. Brandt, and S. Weil, "Ceph as a Scalable Alternative to the Hadoop Distributed File System," *USENIX; login:*, vol. 35, no. 4, pp. 38-49, 2010.

[9] D. Beaver, S. Kumar, H.C. Li, J. Sobel, and P. Vajgel, "Finding a Needle in Haystack: Facebook's Photo Storage," *Proc. Ninth USENIX Symp. Operating Systems Design and Implementation (OSDI)*, pp. 47-60, Oct. 2010.

[10] B. Debnath, S. Sengupta, and J. Li, "FlashStore: High Throughput Persistent Key-Value Store," *Proc. 36th Int'l Conf. Very Large Data Bases (VLDB)*, pp. 1414-1425, Sept. 2010.

[11] R. Pagh and F.F. Rodler, "Cuckoo Hashing," *J . Algorithms*, vol. 51, no. 2, pp. 122-144, 2004.

[12] B.H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Comm. ACM*, vol. 13, no. 7, pp. 422-426, 1970.

[13] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and Network Applications of Dynamic Bloom Filters," *Proc. 25th IEEE INFOCOM*, pp. 1-12, Apr. 2006.

[14] K. Xie, Y. Min, D. Zhang, J. Wen, and G. Xie, "A Scalable Bloom Filter for Membership Queries," *Proc. 50th IEEE GLOBECOM*, pp. 543-547, Nov. 2007.

[15] P.S. Almeida, C. Baquero, N. Preguica, and D. Hutchison, "Scalable Bloom Filters," *Information Processing Letters*, vol. 101, no. 6, pp. 255-261, 2007.

[16] F. Hao, M. Kodialam, and T.V. Lakshman, "Incremental Bloom Filters," *Proc. 27th IEEE INFOCOM*, pp. 1741-1749, Apr. 2008.

[17] J. Wei, H. Jiang, K. Zhou, and D. Feng, "MAD2: A Scalable High-Throughput Exact Deduplication Approach for Network Backup Services," *Proc. 26th IEEE Symp. Mass Storage Systems and Technologies (MSST)*, May 2010.

[18] D.J. Abadi, "Data Management in the Cloud: Limitations and Opportunities," *IEEE Bull. on Data Eng.*, vol. 32, no. 1, pp. 3-12, Jan. 2009.

[19] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "NFS Version 3: Design and Implementation," *Proc. Summer 1994 USENIX Technical Conf.*, pp. 137-151, June 1994.

[20] S.A. Weil, K.T. Pollack, S.A. Brandt, and E.L. Miller, "Dynamic Metadata Management for Petabyte-Scale File Systems," *Proc. ACM/IEEE Conf. Supercomputing (SC)*, pp. 523-534, Nov. 2004.

[21] S.A. Weil, S.A. Brandt, E.L. Miller, and D.D.E. Long, "Ceph: A Scalable, High-Performance Distributed File System," *Proc. Seventh USENIX Symp. Operating Systems Design and Implementation (OSDI)*, pp. 307-320, Nov. 2006.

[22] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "HYDRAstor: A Scalable Secondary Storage," *Proc. Seventh USENIX Conf. File and Storage Technologies (FAST)*, pp. 197-210, Feb. 2009.

[23] S.A. Brandt, E.L. Miller, D.D.E. Long, and L. Xue, "Efficient Metadata Management in Large Distributed Storage Systems," *Proc. 20th IEEE /11th NASA Goddard Conf. Mass Storage Systems and Technologies (MSST)*, pp. 290-298, Apr. 2003.

[24] Y. Zhu, H. Jiang, and J. Wang, "Hierarchical Bloom Filter Arrays (HBA): A Novel, Scalable Metadata Management System for Large Cluster-Based Storage," *Proc. IEEE Int'l Conf. Cluster Computing (CLUSTER)*, pp. 165-174, Sept. 2004.

[25] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, "Scalable and Adaptive Metadata Management in Ultra Large-Scale File Systems," *Proc. 28th Int'l Conf. Distributed Computing Systems (ICDCS)*, pp. 403-410, June 2008.

[26] A.W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E.L. Miller, "Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems," *Proc. Seventh USENIX Conf. File and Storage Technologies (FAST)*, pp. 153-166, Feb. 2009.

[27] A.Z. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485-509, 2005.

[28] A. Kirsch and M. Mitzenmacher, "Less Hashing, Same Performance: Building a Better Bloom Filter," *Proc. 14th Ann. European Symp. Algorithms (ESA)*, pp. 456-467, Sept. 2006.

[29] F. Chang, W. Feng, and K. Li, "Approximate Caches for Packet Classification," *Proc. 23rd IEEE INFOCOM*, pp. 2196-2207, Mar. 2004.

[30] F Putze, P. Sanders, and J. Singler, "Cache-, Hash- and Space-Efficient Bloom Filters," *Proc. Workshop Experimental Algorithms*, pp. 108-121, 2007.

[31] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath, "Cheap and Large CAMs for High Performance Data-Intensive Networked Systems," *Proc. Seventh USENIX Symp. Networked Systems Design and Implementation (NSDI)*, pp. 433-448, Apr. 2010.

[32] L. Zhang and Y. Guan, "Detecting Click Fraud in Pay-Per-Click Streams of Online Advertising Networks," *Proc. 28th IEEE Int'l Conf. Distributed Computing Systems (ICDCS)*, pp. 77-84, June 2008.

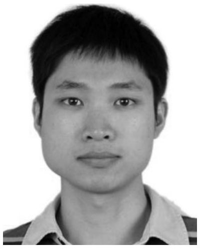
[33] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The Dynamic Bloom Filters," *IEEE Trans. Knowledge and Data Eng.*, vol. 22, no. 1, pp. 120-133, Jan. 2010.

[34] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner, "SIMD-Scan: Ultra Fast in-Memory Table Scan Using on-Chip Vector Processing Units," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 385-394, 2009.

[35] B. Zhu, K. Li, and H. Patterson, "Avoiding the Disk Bottleneck in the Data Domain Deduplication File System," *Proc. Sixth USENIX Conf. File and Storage Technologies (FAST)*, pp. 269-282, Feb. 2008.

[36] G.M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *Proc. Spring Joint Computer Conf. (AFIPS Conf. Proc.)*, pp. 483-485, 1967.

[37] L. Fan, P. Cao, J. Almeida, and A.Z. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Trans. Networking*, vol. 8, no. 3, pp. 281-293, June 2000.



Jiansheng Wei received the BE degree in computer science and technology and the PhD degree in computer architecture from Huazhong University of Science and Technology, Wuhan, China, in 2005 and 2012, respectively. His research interests include computer architecture and storage systems. He received Best Paper Award at IEEE NAS 2011. He is a member of the IEEE.

Hong Jiang received the BSc degree in computer engineering in 1982 from Huazhong University of Science and Technology, Wuhan, China, the MASc degree in computer engineering in 1987 from the University of Toronto, Toronto, Canada, and the PhD degree in computer science in 1991 from the Texas A&M University, College Station, Texas. Since August 1991, he has been at the University of Nebraska-Lincoln, Lincoln, Nebraska, where he served as a vice chair of the Department of Computer Science and Engineering (CSE) from 2001 to 2007 and is a professor of CSE. At UNL, he has graduated more than 10 PhD students who upon their graduations either landed academic tenure-track positions in PhD-granting US institutions or were employed by major US IT corporations. His present research interests include computer architecture, computer storage systems and parallel I/O, high-performance computing, big data computing, cloud computing, performance evaluation. He serves as an associate editor of the *IEEE Transactions on Parallel and Distributed Systems*. He has more than 200 publications in major journals and international Conferences in these areas, *including IEEE-TPDS, IEEE-TC, ACM-TACO, JPDC*, ISCA, MICRO, USENIX ATC, FAST, LISA, ICDCS, IPDPS, MIDDLEWARE, OOPLAS, ECOOP, SC, ICS, HPDC, ICPP, etc., and his research has been supported by US National Science Foundation (NSF), DOD and the State of Nebraska. He is a senior member of the IEEE, and a member of the ACM.

Ke Zhou received the BE, ME, and PhD degrees in computer science and technology from Huazhong University of Science and Technology (HUST), China, in 1996, 1999, and 2003, respectively. He is a professor of the School of Computer Science and Technology, HUST. His main research interests include computer architecture, cloud storage, parallel I/O and storage security. He has more than 50 publications in journals and international conferences, including *Performance Evaluation*, *FAST*, MSST, ACM MM, SYSTOR, MASCOTS, and ICC. He is a member of the IEEE.

Dan Feng received the BE, ME, and PhD degrees in computer science and technology from Huazhong University of Science and Technology (HUST), China, in 1991, 1994, and 1997, respectively. She is a professor and a vice dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 100 publications in journals and international conferences, including *IEEE-TPDS, IEEE-TC, ACM-TOS, JCST*, USENIX ATC, FAST, ICDCS, HPDC, SC, ICS, IPDPS and ICPP. She is a member of the IEEE.