Noname manuscript No.
(will be inserted by the editor)

EAD: Elasticity Aware Deduplication Manager for Datacenters with Multi-tier Storage Systems

Zhengyu Yang · Yufeng Wang · Janki Bhamini · Chiu C. Tan · Ningfang Mi

Received: date / Accepted: date

Abstract The popularity of Big Data applications places pressures on storage systems to efficiently scale to meet the demand. At the same time, new developments like solid-state drives have changed to traditional storage hierarchy. Cloud storage systems are transitioning towards a hybrid architecture consisting of large amounts of memory, solid-state disks (SSDs), and traditional magnetic hard disks (HD). This paper presents *Elasticity Aware Deduplication* (EAD), a data deduplication framework designed for multi-tier cloud storage architectures consisting of SSD and HD. EAD dynamically adjusts the deduplication parameters at runtime in order to improve performance. Experimental results indicate that EAD is able to detect more than 98% of all duplicate data, but it only consumes less than 5% of expected memory space. Additionally, EAD saves approximately 74% of overall IO access cost compared to the traditional design.

Keywords Deduplication Estimation · Scalability, Migration · Cloud Storage Systems · Fusion Disk · Adaptive Dynamical Sampling keyword · Cluster Computing · Cloud Computing

1 Introduction

Big Data applications require efficient storage systems to support them. The ever increasing amount of data generated (we are expecting data to reach 35 zettabytes by the year 2020), places stress on existing storage solutions [1]. The storage research community has responded by developing new techniques in both storage system design and hardware design.

Data deduplication has emerged as an important technique to manage this increase in data [2,3]. This technique is based on the observation that a lot of the data is not unique, and proposes efficient ways of identifying and eliminating duplicate data. Data deduplication has been shown as an essential and critical component in cloud backup, synchronization and archiving storage systems. It not only reduces the storage space requirements, but also improves the throughput of the backup and archiving systems by eliminating the network transmission of redundant data, as well as reduces the energy consumption by deploying fewer disks.

This paper focuses on *inline deduplication*, a type of deduplication system where the objectives are to minimize the data transfer between the client and server, as well as minimize the storage utilized at the backend server. This is done by having the client transmit metadata to the server to detect duplications and only the new data is going to be sent to the server. One of the challenges in inline deduplication is the need to maintain a large index of existing data fragments, or chunks, to avoid sending duplicated data over. Maintaining this type of index in magnetic hard disk (HD) is slow, because of the long disk IO time. Keeping the index in RAM is much faster, but the size of RAM in existing systems limits the number of index entries we can keep in memory. This is especially true when these systems have to handle tens of terabytes to petabytes of data.

In this paper, we present an *Elasticity Aware Deduplication* (EAD) system to address this problem. EAD is designed for cloud based storage systems. EAD improves the performance of the deduplication cache sys-

Zhengyu Yang · Janki Bhamini · Ningfang Mi
Northeastern University, 360 Huntington Ave, Boston, MA 02115
E-mail: yang.zhe,bhimani,ningfang@ece.neu.edu

Yufeng Wang · Chiu C. Tan
Temple University, 1801 N Broad St, Philadelphia, PA 19122
E-mail: y.f.wang, cctan@temple.edu

tem by taking advantage of multi-tier storage architecture that is increasingly becoming popular [4,5]. A multi-tier storage architecture has an extra tier between RAM and HD. This extra tier is faster than HD, but cheaper than RAM. This middle tier is commonly implemented using solid-state disks (SSDs) made from NAND flash memory. We make the following contributions.

1. EAD includes an adaptive sampling algorithm to decide the allocation of data chunks in each of the three tiers. Our adaptive algorithm is compatible with existing deduplication systems that use sampling to take advantage of locality [6][7], as well as systems that use content-based chunking techniques [8][9][10].
2. EAD takes advantage of the rapid scalability property of cloud computing systems to dynamically adjust the amount of RAM and SSD resources as needed to detect sufficient amount of duplicate data. The EAD algorithm will determine when to trigger the scaling up operation, and how many new resources to request. This avoids paying for additional resources that do not contribute to the performance of the system. The EAD algorithm can detect duplicated datasets from different VMs.
3. We present extensive theoretical analysis and experimental results to evaluate EAD. Results show that EAD can detect over 98% of the duplicated data with only 25% of existing approach's IO access cost.

The rest of the paper is organized as follows: Section 2 explores the background of the deduplication system. Section 3 describes the design of our EAD system, and Section 4 evaluates our solution. We discuss the related work in Section 5, and Section 6 concludes this paper.

2 Background

A cloud-based storage system that uses data deduplication has three main components, the `Client`, the `Dedup Server`, and the `Storage Pool`. The interaction of these three components is illustrated in Fig. 1. The client that wants to upload data to the cloud will first split the data into smaller *chunks*. These chunks can either have a pre-defined fixed size, or a variable size divided using known metric such as Rabin fingerprinting [11]. A group of chunks that exhibit locality is known as a *segment*.

2.1 Measuring Deduplication Performance

Key metrics for deduplication performance are the *deduplication ratio* (DR, i.e., removed redundancy data ratio) and *unique data ratio* (UR, i.e., unique data ratio) [12,13]. These are combined together as:

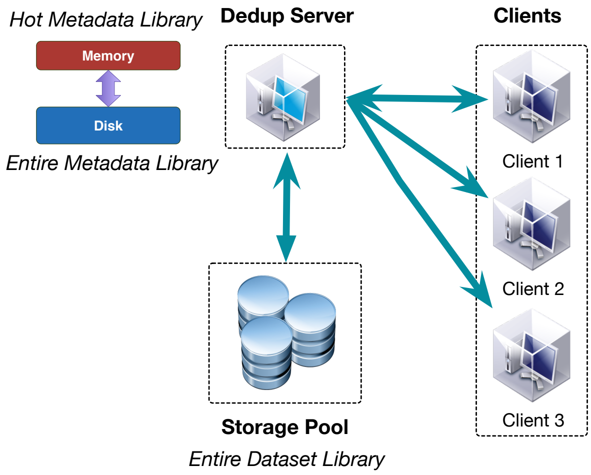


Fig. 1: A typical cloud-based deduplication system

$$DR = 1 - \frac{S_{Dep}}{S_{Org}} \leq 1 - \frac{S_{Unq}}{S_{Org}} = 1 - UR, \quad (1)$$

where S_{Org}, S_{Dep} and S_{Unq} are the size of the original dataset, the size of the dataset after deduplication, and the size of the actual unique dataset of the workload, respectively. UR is determined by the workload's redundancy and also caps the upper bound of the deduplication performance, while DR reflects the deduplication performance of the system. In general, the higher DR is, the less bandwidth and storage will be consumed, and thus the better the performance.

2.2 Impact of Storage Resources

As more data is stored in the storage pool, the size of the $IndexTable$ will also increase. Since storing the $IndexTable$ in disk incurs a bottleneck during the lookup process, parts of the $IndexTable$ need to be cached in RAM, as illustrated in left-top corner in Fig. 1. While it may appear that the solution to improving performance is to maximize the amount of RAM, this is not always correct.

To illustrate this, we conducted an experiment to show the relationship between RAM size and deduplication performance. We used two virtual machine images as our workload, a common workload used in deduplication research [7,14]. The first virtual machine image, VM1, contains majority of text files, mimicking an operating system in office use. The second virtual machine image, VM2, contains majority video data, representing an operating system for home use.

Fig. 2 shows the results. We see that the number of index entry slots indicates how much information of

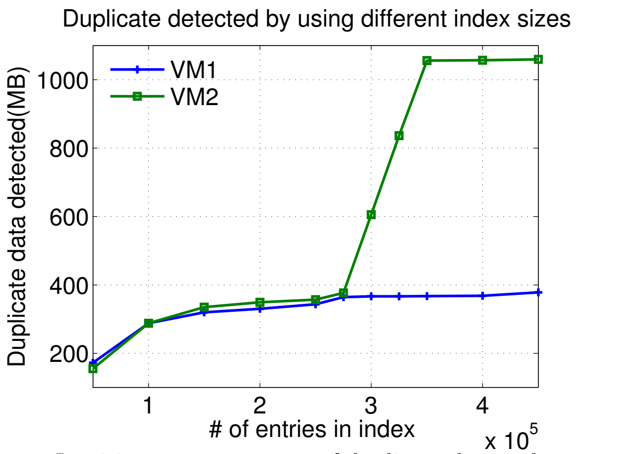


Fig. 2: Intuitive test on amount of duplicate detected on two equal-sized VMs by using equal-size indexes

already stored data the system can provide for duplicate detection. We set fixed number of index entries for duplicate detection and gradually increase it. We see that when index entry slots number increases to 270 thousand, both VMs exhibit the same amount of duplicate data. As we increase the index size, VM1 shows limited improvement, while VM2 shows much better performance. If we simply use VM1 to estimate memory demand, then it will lead to much less bandwidth savings, since buying too much memory is wasteful if most of the data resemble VM1.

3 Elasticity Aware Deduplication (EAD)

Our EAD system comprises of four components: (1) *EAD Client* runs on the client side and is responsible for file chunking, fingerprint computation and sampling; (2) *EAD Server* is executed by the cloud provider and controls the index management; (3) *Network Layer* connects a centralized *EAD Server* and multiple *EAD Clients*; and (4) *Storage Pool* stores all the data ($\mathbb{D}$). Each chunk that is detected as new unique data will be transmitted and stored within $\mathbb{D}$. Fig. 3 illustrates how these components work together. To better represent our algorithm, we summarize some frequently used notations in Table 1.

EAD is designed for a multi-tier storage architecture consisting of RAM, SSD, and HD. We store three types of data in the RAM of *EAD Sever*. The first type of data is the "Hot Metadata Library" ($\mathbb{T}_M$), which is the hot cache of $\mathbb{T}_D$. $\mathbb{T}_M$ will be shrinked and refined during downsampling, thus may not necessarily cover the entire $\mathbb{D}$. This is inevitable and will lead to re-send and re-storage the existing data in the *Storage Pool* (i.e., false negative error). The second type of data is the Estimation Base ($\mathbb{B}$), which is for downsampling and performance analysis. Each entry slot in $\mathbb{B}$ includes

Table 1: Summary of frequently used notations.

Notation	Description
DR, UR	Deduplication ratio and unique data ratios.
$\mathbb{C}$	Chunk cache.
$\mathbb{B}$	Estimation base.
$\mathbb{T}_M$, $\mathbb{T}_S$, $\mathbb{T}_H$	Hot, warm and entire metadata libraries.
$\mathbb{T}_D$	$\mathbb{T}_S \cup \mathbb{T}_H$, metadata library stored in fusion disk.
$\mathbb{D}$	Entire dataset library persisted in storage pool.
EDR	Esteemed deduplication ratio.
DR_{mr}	Measured accumulated data stream deduplication ratio.
DR_{er}	Deduplication ratio of entire dataset.
M_{New}, M_{Cur}	New and current memory sizes.
h_{Smp}	# of fingerprint hits in $\mathbb{B}$ from a sample hits in $\mathbb{B}$.
h_{Seg}	# of fingerprint hits in $\mathbb{B}$ from all the chunks uploaded during current deduplicaiton epoch.
R	Current sample rate.
Δ	Scaling up multiplier.
$\phi(d)$	Adjust function of downsampling operation counter d.

a fingerprint and two counters, h_{Smp} and h_{Seg}, where counter h_{Smp} records the number of fingerprint hits in $\mathbb{B}$ *from the sample set*, and h_{Seg} records the number of fingerprint hits in $\mathbb{B}$ *from all chunks uploaded during the current deduplication epoch*. The third type of data is the Chunk Cache ($\mathbb{C}$), which is a dedicated loading area for comparing the segments prefetching from $\mathbb{T}_D$ with the incoming fingerprints (FPs). This loading area will be emptied after each epoch. We mark $\mathbb{C}$ as "volatile" and mark others as "persistent" in Fig. 3.

We consider a high-speed SSD and a large-capacity HD that resemble a "fusion disk" [15] ($\mathbb{T}_D$) in *EAD Server* side to improve the access speed of the metadata library. The server does not store dataset content but only metadata. The "Entire Metadata Library" ($\mathbb{T}_H$, and we have $\mathbb{T}_D = \mathbb{T}_S + \mathbb{T}_H$) is maintained and stored in the HD, which maps the relationship between chunks and segments, and chunk's fingerprints with their physical addresses in the *Storage Pool*. Meanwhile, as a write-back cache of the HD, SSD stores the Warm Metadata Library ($\mathbb{T}_S$). Different caching replacement policies can be adopted in this fusion disk.

Notice that although we focus on a centralized EAD server cluster, it also supports multiple servers-clients cluster. In this case, EAD servers periodically sync updates of libraries to each other. Moreover, load balancing mechanisms are also adopted to balance the load.

3.1 EAD Deduplication Algorithm

EAD first applies the downsampling algorithms until the deduplication performance becomes unsatisfactory, and then triggers the scaling up process to obtain more RAM for better performance. Algs. 1, 2 and 3 describe the main procedure of EAD algorithm. There are three phases: phase 1 and 2 are as generic inline

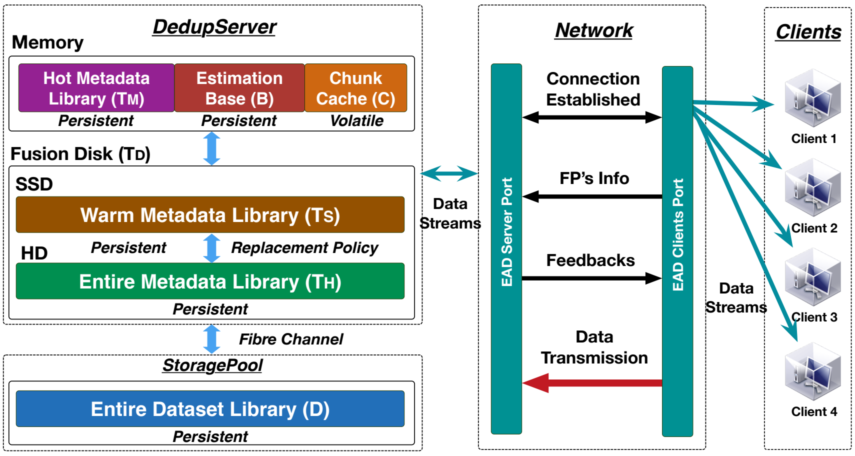


Fig. 3: EAD infrastructure.

Algorithm 1 Elastic deduplication strategy (Part 1)

1: $R = 1$ /* Initialize sampling rate */
2: **/* Dedup Phase 1: Identify duplicate chunks */**
3: **/* 1.1 Client sends FPs to server */**
4: EAD Client randomly selects κ $FP_{x_i}^{Smp}$s from S_{in} and mark them in S_{Smp}, then sends all FPs S_{in} with the S_{Smp} info to EAD Server
5: **/* 1.2 Deduplication prefetch process */**
6: **for all** $FP_{x_i} \in S_{in}$ **do**
7:   **if** $FP_{x_i} \in \mathbb{T}_M$ **then**
8:     Fetch all FPs from (at most C_{Max}) segment(s) in $\mathbb{T}_D$ that contain FP_{x_i} into $\mathbb{C}$
9:   **else**
10:     Add FP_{x_i} to $\mathbb{T}_M$ /* Record new FPs into $\mathbb{T}_M$ */
11: Add segment info of S_{in} to $\mathbb{T}_D$ /* Update $\mathbb{T}_D$ */

deduplication systems, and phase 3 is responsible for adjusting the sampling rate and triggering the scaling process.

[Phase 1.1] Client sends FPs to server: The first deduplication phase is to identify duplicate chunks. For each new segment S_{in}, the client first randomly selects κ samples from it and groups them into a sample set S_{Smp}. The client then sends all FPs of S_{in} together with S_{Smp} information to the server (Alg. 1 line 3 and 4). This integrates our estimation for downsampling process into the regular deduplication operations, so as to avoid the separate sampling and scanning phases as done by [16]. Thus, there is no extra overhead for our estimation purpose. Notice that in order to ensure the atomicity under concurrent accesses, a buffer is built to queue and sequentialize these incoming accesses in the centralized EAD server.

[Phase 1.2] Deduplication prefetch process: Once received S_{in} and S_{Smp}, the EAD server searches for each FP of S_{in} in the hot metadata library in RAM ($\mathbb{T}_M$). If found, then it prefetches the entire segment(s) stored in $\mathbb{T}_D$ from the disk to the chunk cache $\mathbb{C}$ (Alg. 1 line 5 to 8) in the RAM for later comparison. The purpose of prefetch is to take advantage of the spatial and temporal locality, although it may be possible that the prefetched segment(s) is(are) not as same as the incoming segment that contains the current FP. Meanwhile, to reduce overhead, we limit the max number (C_{Max}) of segments to be loaded since it is possible that more than one segment are found in $\mathbb{T}_D$ (an FP may appear in different segments during runtime). If the FP is not found, then EAD records this new FP in $\mathbb{T}_M$ (line 9 to 10). Finally, EAD adds the information of the current iterated fingerprint "$FP_{x_i} \in S_{in}$" to $\mathbb{T}_D$, regardless of whether it is found or not in $\mathbb{T}_M$ (Alg. 1 line 11).

[Phase 1.3] Sampling and estimation process: After the prefetching process, EAD uses the sample chunks in S_{Smp} to update the estimation base $\mathbb{B}$ and downsampling counter h_{Smp_i}. Specifically, EAD searches each received sample FP in S_{Smp} for each estimation base $\mathbb{B}$. If the FP is found in $\mathbb{B}$, the counter h_{Smp_i} of that FP stored in the $\mathbb{B}$ will be increased by one, otherwise this FP will be added to $\mathbb{B}$ with h_{Smp_i} and h_{Seg_i} counters being initialized to zero (Alg. 2 line 1 to 6).

[Phase 1.4] Duplication detection process: EAD then compares each fingerprint $x_i \in S_{in}$ with the prefetched segments in $\mathbb{C}$ during the current epoch, and marks them as "*dup*" or "*unq*" (indicating it is a duplicate or an unique chunk). Meanwhile, EAD updates $\mathbb{B}$ again – incrementing the counter h_{Seg_i} by one every time its correspondent fingerprint appears (Alg. 2 line 7 to 18). Once EAD finishes this process, it empties the $\mathbb{C}$ (Alg. 2 line 19).

[Phase 2] Data transmission:

The second deduplication phase is that *EAD Client* only transmits unique data chunks along with metadata of duplicate chunks to *EAD Server* (Alg. 3 line 1 to 3), which saves both bandwidth and storage space.

Algorithm 2 Elastic deduplication strategy (Part 2)

/* 1.3 Sampling and estimation process */
for all $FP_{x_i} \in S_{Smp}$ **do**
  if $FP_{x_i} \in \mathbb{B}$ **then**
    $h_{Smp_i} = h_{Smp_i} + 1$
  else
    Add FP_{x_i} to $\mathbb{B}$
/* 1.4 Duplication detection process */
for all $x_i \in S_{in}$ **do**
  $\forall x_k \in \mathbb{C}$ Compare FP_{x_i} with FP_{x_k}
  if $FP_{x_i} = FP_{x_k}$ **then**
    /* Found in prefetched segment in $\mathbb{C}$ */
    Set $x_i \in dup$ (x_i^{Dup})
  else
    /* Not found in prefetched segment in $\mathbb{C}$ */
    Set $x_i \in unq$ (x_i^{Unq})
  $\forall x_l \in \mathbb{B}$: Compare FP_{x_i} with FP_{x_l}
  if $FP_{x_i} = FP_{x_l}$ **then**
    $h_{Seg_i} = h_{Seg_i} + 1$
Empty $\mathbb{C}$

Algorithm 3 Elastic deduplication strategy (Part 3)

1: **/* Dedup Phase 2: Data transmission */**
2: **for all** $x_i \in S_{in}$ **do**
3:   Transmits x_i^{Unq} along with only metadata of x_i^{Dup}
4: **/* Dedup Phase 3: Downsampling and scaling */**
5: **if** Index is approaching the RAM limit **then**
6:   **if** $DR_{mr} < \Gamma \cdot EDR$ **then**
7:     **if** $R = 1$ **then**
8:       $\Gamma = \frac{DR_{mr}}{EDR}$ /* Calibration */
9:     **else**
10:       Scale up $RAM_{New} = RAM_{Cur} \cdot \Delta \cdot \phi(d)$
11:       Set $R = \Delta \cdot R$
12:       Refine($\mathbb{T}_M$)
13:   **else**
14:     $R = \frac{R}{\Delta}$ /* Downsampling */

[Phase 3] Downsampling and scaling:

The last deduplication phase (Alg. 3 line 4 to 14) is adaptively adjusting the sampling rate and triggering RAM scaling up process. We will discuss them in detail in Sec. 3.2.

3.2 Dynamically Adjust Sampling Rate

As one of the key features, EAD determines whether it is beneficial from scaling operation, by first investigating what causes current poor deduplication performance once the RAM limitation is reached (Alg. 3 line 4). In order to distinguish whether poor deduplication performance is due to overly aggressive downsampling or inherent within the dataset (e.g. data in multimedia or encrypted files), EAD needs to know the DR of the *entire dataset* (denoted as "DR_{er}"). If the RAM reaches the limitation and $DR_{er} < \Gamma$, then one needs to trigger the sampling rate or scaling up adjustment. In fact, the same methodology can be applied to SSD scaling, but in this paper we only focus on RAM scaling.

Unfortunately, obtaining the actual DR_{er} of the entire dataset is impractical, since it requires performing the entire deduplication process. Prior work from [16] provided an estimation algorithm to estimate the deduplication performance for static, fixed-size data sets. Their algorithm requires the actual data to be available in order to perform random sampling and comparisons. However, in our case, the dataset can be viewed as a stream of data, thus there is no prior knowledge of the size or characteristics of the data to be stored in advance. One more bad news is that it is not possible to perform back and forth scanning of the complete dataset for estimation. Therefore, we need to find a way to estimate the DR_{er}. One straightforward way to measure the deduplication ratio of stream accumulated from the beginning, note as DR_{mr}:

$$DR_{mr} = \frac{S_{mrDup}}{S_{curTtl}}, \tag{2}$$

where S_{mrDup} stands for the detected duplicated data size until current moment. S_{curTtl} is the total data size so far. The measured DR_{mr} may not accurately reflect the real duplication ratio of the workload since it is highly affected by the prefetching process. That is to say, a low S_{mrDup} may due to aggressive low sampling rate and low hit ratio in the prefetched set (C), rather than the workload's actual characteristics.

We develop a more accurate method to estimate the DR_{er}, called "EDR" (Expectation of dataset's Duplication Ratio). The design goal is to directly conduct statistical analysis on the accumulated incoming stream and try best to be independent from the prefetching process and the cache system, so that EDR has higher chance to reflect and predict the entire DR_{er}. To achieve this goal, EDR uses the counters stored in $\mathbb{B}$:

$$EDR = 1 - \frac{1}{\kappa \cdot n_s} \sum_{i \in \mathbb{B}} \frac{h_{Smp_i}}{h_{Seg_i}}. \tag{3}$$

The intuition and correctness proof of EDR will be provided in next two sections. The computation of EDR happens while the index size is approaching the memory limit (the first threshold as shown in Alg. 3 line 4). EAD has a second threshold (Alg. 3 line 5) which is based on user-acceptable deduplication performance level $\Gamma \in (0, 1)$. This QoS (Qualify of Service) related parameter indicates that the lower the cost a user is willing to pay, the worse deduplication performance (a lower Γ) the user needs to tolerate as an exchange. Only if both these thresholds are reached, EAD enlargers RAM size, and adjusts the current sampling rate R (Alg. 3 line 9 to 12). Otherwise, EAD will not scale up and only apply downsampling on the index (Alg. 3 line 14). This is based on the fact that given a dataset that inherently exhibits poor deduplication characteristics [17], adding more RAM will incur some overhead with slight or even no improvement on deduplication performance.

3.3 EAD Parameters and Scaling Procedures

Our EAD algorithm has several parameters that can be adjusted by taking advantage of runtime observations to improve the overall performance. Here we discuss the key paramters and their adjustments.

■ **Adjusting Γ**

The parameter Γ is specified by the user, and indicates the user's desired level of deduplication performance. However, the user may sometimes be unaware of the underlying potential deduplication performance of the data, and set an excessively high Γ value, resulting in unnecessary scaling over time. We first need to adjust (calibrate) the user's Γ value to $\frac{DR_{mr}}{EDR}$, in the case that DR_{mr} has not reached the acceptable performance, even if the sampling rate is one (fully sampled). In fact, when the current sample rate $R = 1$, $\frac{DR_{mr}}{EDR}$ represents the current system's maximum deduplication ability. Later, EAD will tune down the sample rate if DR_{mr} is larger than $\frac{DR_{mr}}{EDR}$. Scaling up the RAM will be finally triggered if (1) $R = 1$, after several downsampling operations; and (2) DR_{mr} is worse than the $\frac{DR_{mr}}{EDR}$. In this way, EAD is able to elastically adapt variations on incoming data.

■ **Adjusting amount of RAM to scale up (Δ)**
A simple way to compute the amount of RAM is using a fixed scaling up multiplier Δ, as:

$$M_{New} = M_{Cur} \cdot \Delta \tag{4}$$

For example, we double the RAM each time ($\Delta = 2$) and reset the sampling rate back to 1, and start all downsampling process over again. However, workloads may not use up all exponentially scaled-up RAM space. Moreover, cost of adding RAM and overhead of corresponding migration are expensive. Therefore, we need to adaptively turn down the Δ, while satisfying the performance requirement and trying best to trigger the scaling up operation as less as possible.

Our solution considers both scaling RAM and the sample rate. For scaling RAM, we gradually decrease the value of Δ to slow down the exponentially scaling speed. For the sampling rate, instead of directly setting it back to $R = 1$ which will quickly occupy the RAM again, we adjust the sampling rate (Alg. 3 line 11) to the same sample rate before the latest downsampling operation (Alg. 3 line 14). Because this sampling rate is able to support a satisfying performance.

We propose the detail of conservative RAM incrementation policy. We introduce a parameter d (initialized as zero) to record occurrences of downsampling, every time the downsampling happens, d increases by one. We set the new RAM (M_{New}), after scaling up as a function of the current RAM size (M_{Cur}, refer Alg. 3 line 10):

$$M_{New} = M_{Cur} \cdot \Delta \cdot \phi(d), \tag{5}$$

where the (conservative) step $\Delta \cdot \phi(d) \in [1, \Delta]$ (i.e., $\phi(d) \in [\frac{1}{\Delta}, 1]$) and incremental step of each scaling up epoch should be less. Under this constraint, we design $\phi(d)$, as a monotonous decreasing function of downsampling operation counter d, as:

$$\phi(d) = \begin{cases} \Delta^{d-1}, & d < 2 \\ 1 - \sum_{i=2}^{d} \frac{1}{\Delta^i}, & d \geq 2 \end{cases}. \tag{6}$$

Thus, the RAM size can be calculated as (Alg. 3 line 10):

$$M_{New} = \begin{cases} M_0 \cdot \Delta^d, & d < 2 \\ M_{Cur} \cdot \Delta \cdot (1 - \sum_{i=2}^{d} \frac{1}{\Delta^i}), & d \geq 2 \end{cases}, \tag{7}$$

where M_0 is the original RAM size. We next prove the value space of $\phi(d)$ is in $[\frac{1}{\Delta}, 1]$. When $d < 2$, it obviously holds, as

$$\phi(d) = \Delta^{d-1} \in [1, \Delta]. \tag{8}$$

When $d \geq 2$, we compare $\phi(d)$ with the upper bound 1:

$$\phi(d) - 1 = 1 - \sum_{i=2}^{d} \frac{1}{\Delta^i} - 1 = -\sum_{i=2}^{d} \frac{1}{\Delta^i} < 0, \tag{9}$$

so it holds. For the lower bound $\frac{1}{\Delta}$, we also have:

$$\phi(d) - \frac{1}{\Delta} = 1 - \sum_{i=2}^{d} \frac{1}{\Delta^i} - \frac{1}{\Delta} = 2 - \sum_{i=0}^{d} \frac{1}{\Delta^i}. \tag{10}$$

Eq. 10 is a monotone decreasing function with the minimum value:

$$min[\phi(d) - \frac{1}{\Delta}] = \lim_{\Delta=\infty} (2 - \sum_{i=0}^{d} \frac{1}{\Delta^i}) \tag{11}$$

$$= \lim_{\Delta=\infty} \{2 - \frac{1[1 - (\frac{1}{\Delta})^n]}{1 - \frac{1}{\Delta}}\} = \frac{\Delta - 2}{\Delta - 1} \geq 0. \tag{12}$$

Therefore, our $\phi(d)$ satisfies the design constrains. Notice that when $\Delta = 2$, the RAM size will finally converge, while RAM sizes in all other cases are divergence. As the times of downsampling operation increase, EAD requires less amount of RAM for index table after scaling up. Comparing such optimization with always requiring Δ times of original RAM, such optimized approach is able to claim higher memory utilization efficiency.

■ **Managing size of $\mathbb{B}$**

One concern with our estimation scheme is that the size of $\mathbb{B}$ may become too large. If we need a large amount of RAM to store $\mathbb{B}$, we will be wasting RAM resources that could be used in caching the hot index $\mathbb{T}_M$. In practice, the size of $\mathbb{B}$ is relatively modest. Each entry in $\mathbb{B}$ consists of a fingerprint and two counters. Using SHA-1 to compute the fingerprint results in a 20 byte fingerprint. Additional four bytes are used for each counter. Thus, each $\mathbb{B}$ entry is 28 bytes, indicating that approximately the total size of $\mathbb{B}$ would be at most 33.38 MB to support 1 TB of data. In our experiment, it only requires 4.32 MB for estimating 163.2 GB dataset.

■ **Refining of $\mathbb{T}_M$ and $\mathbb{T}_D$**

Algorithm 4 Refine Index Table

1: **for** each selected index entry x (with FP_x) **do**
2: Locate its correspondent segment Seg_x
3: **if** Seg_x has not been processed **then**
4: Select new sample chunks from Seg_x based on current sampling rate
5: **for** *new selected sampled chunk* (with FP_y) **do**
6: **if** FP_y finds matching record in *IndexExt* ($FP_y = FP_j \in$*IndexExt*) **then**
7: Locate Seg_j and pull out its FPs to $\mathbb{C}$ for duplication re-detection
8: **else**
9: Add FP_y into *IndexExt* as a new entry
10: **for** $\alpha = 1$ to total number of chunks in Seg_x **do**
11: Compare FP_α with those in Seg_j in $\mathbb{C}$
12: **if** FP_α finds match **then**
13: Chunk α is duplicate and remove it from $\mathbb{T}_D$
14: Entry x will be removed from *IndexOrg*, *Bloom Filter* records information of FP_x

EAD removes certain amount of entries from the hot index $\mathbb{T}_M$ according to the new sample rate R during downsampling. After the scaling up operation is finished, we are left with the original index $\mathbb{T}_M$ and new RAM space for extending $\mathbb{T}_M$. EAD will do two things: (1) refining the original index by removing low hit ratio entries, and (2) adding new entries that are predicted to have better deduplication-detection ability than removed ones from $\mathbb{T}_D$ to $\mathbb{T}_M$. This adding operation is with new sample rate which is higher than original one. Here we partition $\mathbb{T}_M$ into two virtual zones: the original index as *IndexOrg*, and the extension part which consists of new appending entries as *IndexExt*. EAD compensates the poor deduplication performance due to previously too sparse sampling rate by index refining, which has two steps:

1. **Re-detect duplication:** Search through the *IndexOrg*, re-detect duplication chunks from already stored segments. Since extra IO operations may bring unexpected cost, EAD is only processing limited number of segments which are able to claim duplicate chunks.
2. **Evict duplication:** It is possible that not all index entries in the old index are useful, meaning that some entries contain FPs for chunks that are unlikely to be encountered again. Therefore, after duplication re-detection, these entries are removed from the *IndexOrg*.

Instead of reprocessing all the segments on the storage, EAD is able to select only part of them for detecting majority duplicate for evictions. We first introduce the additional information into index table to help estimating the expected number of entries to be evicted: a counter ($count_{FP}$, initialized as zero for new added entries) is used for each index entry to record its hit time in vector $\mathbb{T}$, which we call hit rate. Every time when an entry has been found a match, this counter increments by 1. Therefore the larger the counter is, the more duplicate chunks this entry can detect. Among those segments hooked by FPs with low hit rate, there exists "evictable" duplicate chunks. This conclusion is derived based on the following analysis of FPs in the index:

1. **Entries with high hit rate.** These FPs in the index indicate that segments have found matches and lots of chunks near the sampled chunks are identical, which is the natural result of chunk locality. Theoretically, more entries having high hit rate imply that more space can be saved.
2. **Entries with low hit rate.** Some segments themselves share few chunks with stored ones, naturally resulting in lower index matching rate. However, we also claim that low hit rate does not always imply that there is no chunk locality. For example, some segments share lots of chunks with stored segments, however they are not hooked by right FPs due to the sparse sampling rate. Thus not enough or even no matches from their sampled chunks' FPs are found in the index.

This index refining function is called in Alg. 3 line 12, and the detail of the refining function is illustrated in Alg. 4. The main idea is:

1. **Refine $\mathbb{T}_M$:** To solve the "low hit rate entries cannot represent hooked segment" problem, we give segments of those selected low hit rate entries one more chance. Specifically, we re-sample new entries from these segments and put them into the *IndexExt* (i.e., Alg. 4 line 9), and remove those low hit rate entries.
2. **Refine $\mathbb{T}_D$:** Those low hit rate entries whose segments have also been sampled by other entries in $\mathbb{T}_M$ with high hit rate will be deleted. Duplicated entries in their hooked segments in $\mathbb{T}_D$ will also be removed (i.e., Alg. 4 line 10-13).

Furthermore, to avoid adding them back to the index table, a *Bloom Filter* (BL) [18] (Alg. 4 line 14) is used to record hash information of removed FPs. By doing so, entries in *IndexOrg* will not be entirely kept and valuable space will be released for future use. When scaling up finishes, EAD will merge *IndexOrg* and *IndexExt*, and calculate the updated Deduplication Ratio. If the new Deduplication Ratio is still lower than $\Gamma \cdot EDR$ after duplication re-detection, EAD will reset the value of Γ, and make $\Gamma \cdot EDR$ to be equal to the value of current Deduplication Ratio. Therefore, the requirement on deduplication performance will not surpass the system ability. In order to select as few chunks as possible to conduct this expensive refining operation, we also propose a method to estimate, how many entries need to go through the duplication re-detection and then be removed from *IndexOrg* to release the space. We use entries with low hit rate to track their correspondent segments and detect evicted duplicate chunks. The threshold for labeling hit rate as *high* or *low* is not arbitrary. Suppose that we have n chunks come in for a backup process, the measured Deduplication Ratio is DR_{mr} ($DR_{mr} < \Gamma \cdot EDR$). At the meantime, we have the value of hit counters as $\{0, 1, \cdots, c, \cdots, m\}$ (c and m are two positions that will be used later), and their correspondent amount of entries are $\{n_0, n_1, \cdots, n_c, \cdots n_m\}$ (i.e., there are n_0 entries whose counter values are zero, etc.). Assume that the sampling rate before the latest downsampling operation is R_0, thus we claim that the minimum number of index entries to be selected is:

$$n_{evt} = R_0 \cdot (\Gamma \cdot EDR - DR_{mr}) \cdot \sum_{i=0}^{m} n_i. \qquad (13)$$

To evict this amount of chunks, based on above calculation, EAD starts picking index entries with counter value as zero (n_0), if $n_0 < R_0 \cdot n_{evt}$, EAD picks entries with hit rate as one and vice versa, until it satisfies (at cth counter position, $0 \le c \le m$):

$$\sum_{i=0}^{c} n_i \ge n_{evt}. \qquad (14)$$

The intuition is that the system should perform as close to the real deduplication ratio of the workload $\Gamma \cdot EDR$ (QoS adjusted) as possible, but it can only detect DR_{mr}. It is highly possible that the missing part of the detection is due to no enough space for *right* sample entries in the index. Therefore, we need to remove space of this part of "junk" or "duplicated" entries for other more useful entries. Lastly, since the sample rate before last downsampling (R_0) is able to handle the current workload, we use this sample rate to pick entries in the $(\Gamma \cdot EDR - DR_{mr}) \cdot \sum_{i=0}^{m} n_i$ to help decrease the overhead.

4 Analytical Analysis of EAD

In this section, we focus on the design intuition of the core technique of EAD: how to estimate the expectation of dataset's duplication ratio (EDR). Notice that more detailed proof is presented in Appendix. If we randomly pick one sample from a segment, then the number of chunks that are same with the picked sample in the segment can somehow reflect the duplication ratio of segment. Our EDR is inspired by this idea and further extends to the case that randomly picking multiple samples from multiple segments. In detail, EDR uses the repeating time of sampled chunks in the stream to estimate the *duplication data amount*, and uses the repeating time of samples in the estimation base to get the *unique sample number*. The fraction of them reflects the duplication ratio as well as the expected deduplication ratio of the stream received so far (DR_{Cur}). This process is independent of the prefetching and is purely relying on the workload, so it would be more accurate than the measured system-performed deduplication ratio DR_{mr}.

To further explain this intuition, we model the problem in Fig. 4. The incoming stream can be divided into multiple same size (T chunks) segments. We introduce the concept of "dupSet" which is a group of duplicated chunks with same FP in each segment. It is a virtual "group" for analysis purpose, and same-FP chunks do not need to come continuously as long as they are in the same segment. Let T be the total size of a segment, and d_i be the ith dupSet (we also use d_i to refer to as $dupSet_i$ for convenience). Since segments consist of these duplicated dupSets, $|d_i| \in [1, T]$ and $\sum_{i=1}^{n} |d_i| = T$. $|d| = 1$ means this dupSet is a non-duplicated chunk in the segment, and $|d| = T$ means the entire segment is fulfilled by chunks with same FP. We then show how EDR is equal or close to the actual DR_{Cur} in three levels. Notice that all the analysis below are after the warming up period (i.e., "first time" indexing).

[Level 1] DupSet: Denote the number of samples from each dupSet as κ_i ($\in [0, d_i]$). When $\kappa_i = 0$, this dupSet is not sampled, and when $\kappa = d_i$ all chunks in this dupSet are sampled. We also have $\sum_{i=1}^{n} \kappa_i = \kappa$, where κ is the sample number of each segment. The exact deduplication ratio of d_i can be calculated as:

$$DR(d_i) = 1 - UR(d_i) = 1 - \frac{1}{d_i}. \qquad (15)$$

Since EAD is based on the sampling counters, it cannot estimate and will ignore those non-sampled dupSet.

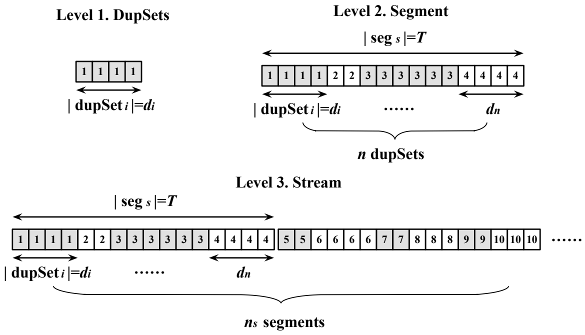


Fig. 4: Example of dupSet, segment and stream.

EDR of a **sampled** dupSet is calculated as:

$$EDR(d_i) = 1 - \frac{1}{\kappa_i}\Big(\sum_{j \in \mathbb{B}(d_i)} \frac{h_{Smp_j}}{h_{Seg_j}}\Big) \tag{16}$$

$$= 1 - \frac{1}{\kappa_i} \cdot \frac{\kappa_i}{d_i} = 1 - \frac{1}{d_i} = DR(d_i), \tag{17}$$

where $\mathbb{B}(d_i)$ means chunks in $\mathbb{B}$ that belong to d_i. Specifically, Eq. 17 holds due to the following facts: (1) all chunks in d_i are exactly same, thus the sample counter h_{Smp_j} is equal to sample size κ_i; and (2) the total hit-in-sample counter h_{Seg_j} in a dupSet should be the same as the total chunk number of the dupSet (d_i). Notice that if there are more than one sample from this dupSet, only one of them will be stored in $\mathbb{B}(d_i)$ (Alg. 2 line 2 and 3). To sum up, in level 1, $DR_i = EDR_i$ stands, which means EDR_i can accurately reflect the actual DR_i. For further comparison, we can accurately calculate the UR_i by using these two counters, as:

$$UR(d_i) = \frac{1}{\kappa_i}\Big(\sum_{j \in \mathbb{B}(d_i)} \frac{h_{Smp_j}}{h_{Seg_j}}\Big). \tag{18}$$

[Level 2] Segment: Segment s has T chunks and is assembled by n dupSets. Its $DR(Seg_s)$ can be exactly calculated as:

$$DR(Seg_s) = 1 - UR(Seg_s) = 1 - \frac{n}{T} = 1 - \frac{1}{\bar{d}}. \tag{19}$$

We can calculate EDR of Seg_s (can contain some non-sampled dupSets) as:

$$EDR(Seg_s) = 1 - \frac{1}{\kappa}\Big(\sum_{i \in \mathbb{B}(Seg_s)} \frac{h_{Smp_i}}{h_{Seg_i}}\Big) \tag{20}$$

$$= 1 - \frac{1}{\kappa}\Big[\sum_{\mathbb{B}(d_i) \in \mathbb{B}(Seg_s)} \Big(\sum_{j \in \mathbb{B}(d_i)} \frac{h_{Smp_j}}{h_{Seg_j}}\Big)\Big] \tag{21}$$

$$= 1 - \frac{1}{\kappa}\Big\{\sum_{\mathbb{B}(d_i) \in \mathbb{B}(Seg_s)} [\kappa_i \cdot UR(d_i)]\Big\} \tag{22}$$

$$= 1 - \sum_{\mathbb{B}(d_i) \in \mathbb{B}(Seg_s)} \Big(\frac{\kappa_i}{\kappa} \cdot \frac{1}{d_i}\Big), \tag{23}$$

where $\mathbb{B}(Seg_s)$ is the estimation base $\mathbb{B}$ of Seg_s. Eq. 21 divides the sum in Eq. 20 into multiple subsums from each **sampled** dupSet. We then use Eq. 18 and 17 to calculate those subsums, and finally get Eq. 23. To solve Eq. 23, we hereby introduce the following approximation based on the assumption that sampled dupSet can reflect the entire segment:

$$\sum_{\mathbb{B}(d_i) \in \mathbb{B}(Seg)} \Big(\frac{\kappa_i}{\kappa} \cdot \frac{1}{d_i}\Big) \approx UR(Seg_s) = \frac{1}{\bar{d}}. \tag{24}$$

Thus, Eq. 23 can be calculated as:

$$EDR(Seg_s) \approx 1 - \frac{1}{\bar{d}} = 1 - UR(Seg_s) = DR_(Seg_s). \tag{25}$$

This approximation has two error sources: (1) EDR may not accurately reflect duplication status of those dupSets with zero samples ($\kappa_i = 0$); and (2) Different dupSets may have different sizes. EDR ignores this and assigns them same weights in Eq. 24. We provide more detailed analysis on them later.

[Level 3] Stream: Since the stream is assembled by multiple (even unlimited) segments, we can accurately calculate DR_{Stream} at moment T_{Cur} as:

$$DR_{Cur}(Stream) = 1 - \frac{\sum_{s=1}^{n_s} T \cdot UR(Seg_s)}{T \cdot n_s}$$

$$= 1 - \frac{\sum_{s=1}^{n_s} UR(Seg_s)}{n_s}, \tag{26}$$

where n_s is number of segments in the stream. We further calculate EDR as:

$$lhs = 1 - \frac{1}{n_s \cdot \kappa}\Big(\sum_{i \in \mathbb{B}(n_s Seg)} \frac{h_{Smp_i}}{h_{Seg_i}}\Big)$$

$$= 1 - \frac{1}{n_s \cdot \kappa}\Big[\sum_{\mathbb{B}(Seg) \in \mathbb{B}(n_s Seg)} \Big(\sum_{i \in \mathbb{B}(Seg)} \frac{h_{Smp_i}}{h_{Seg_i}}\Big)\Big]$$

$$= 1 - \frac{1}{n_s}\Big[\sum_{\mathbb{B}(Seg) \in \mathbb{B}(n_s Seg)} \frac{1}{\kappa}\Big(\sum_{i \in \mathbb{B}(Seg)} \frac{h_{Smp_i}}{h_{Seg_i}}\Big)\Big]$$

$$= 1 - \frac{1}{n_s}\Big\{\sum_{\mathbb{B}(Seg) \in \mathbb{B}(n_s Seg)} [1 - EDR(Seg_s)]\Big\}. \tag{27}$$

Here we use the "average" segment's $UR(Seg_s)$ (Eq. 25) to regress "$1 - EDR(Seg_s)$" in Eq. 27 with a slight regression error, as:

$$lhs \approx 1 - \frac{1}{n_s}\Big[\sum_{\mathbb{B}(Seg) \in \mathbb{B}(n_s Seg)} UR(Seg_s)\Big] \tag{28}$$

$$= 1 - \frac{\sum_{s=1}^{n_s} UR(Seg_s)}{n_s} = DR_{Cur}(Stream). \tag{29}$$

5 Evaluation

We created a realistic dataset to evaluate our solution. The dataset comprises of virtual machine images. These images have different types of programs installed, as well as different types of data drawn from Wikimedia Archives [19] and OpenfMRI [20]. For the following results, we denote our EAD solution as `Elastic`, and compare against two alternatives. The first alternative, denoted as `FullIndex`, represents an ideal situation where there is unlimited RAM available. This will serve as an upper bound on the total amount of space savings. The other alternative is denoted as *DownSample*, which is a recent approach [7] that dynamically adjusts the sampling rate to deal with insufficient RAM. Table 2 summarizes the configuration of our testbed.

Table 2: Testbed configuration.

Component	Specifications
Processor	Intel i3-2120T at 2.60GHz
Processor Cores	4 Cores
Memory Capacity	8GB
RAID Controller	LSI SAS 2008
Network	10 Gigabit Ethernet NIC
Operating system	Ubuntu 12.04.5
Linux Kernel	3.14 Mainline
SSD Made	Intel and Samsung NVMe SSDs
SSD Capacity	500GB - 4 TB
HDD Made	Western Digital
HDD Capacity	2 - 8 TB

5.1 Performance of EAD

■ **Deduplication ratio.** Deduplication ratio is the standard metric used to evaluate deduplication systems [13, 21]. Here, we use the *normalized deduplication ratio* as our metric for evaluation (normalized by the full index as the theoretical upper bound). The normalized deduplication ratio is defined as the ratio of measured *Deduplication Ratio* to *Deduplication Ratio* of `FullIndex` deduplication.

As shown in Fig. 5, although EAD does not claim equally high ratio compared to FullIndex and Downsample, the performance of `Elastic` is always higher than 98% and the gap between it and the other two is less than 2%. When about 5% of data has been processed, `Elastic` has a thriving performance. Because of trivial size of initial index size, `Elastic` cannot detect enough duplicate chunks, leading to a poor performance. Fig. 7 and Fig. 8 further show how sampling rate and number of index slots used vary above cases. Both `DownSample` and `Elastic` have comparatively very low memory cost, which shows that EAD is able to use less RAM space to achieve a satisfying deduplication ratio.

■ **Deduplication efficiency.** Another metric we use is the deduplication efficiency. The deduplication efficiency is the ratio of the duplicat data detected to the number of index entry slots, which reflects a deduplication algorithm's ability to cache most valuable index entry slots.

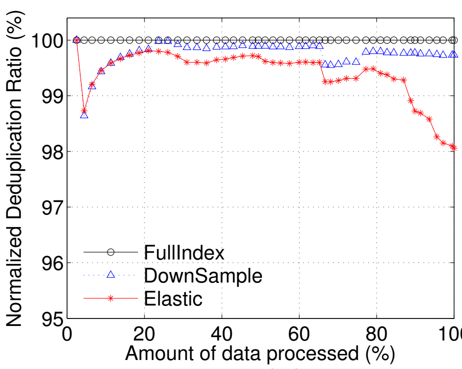


Fig. 5: Norm. dedup ratio.

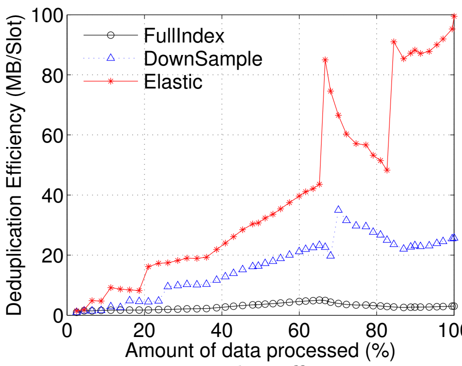


Fig. 6: Dedup efficiency.

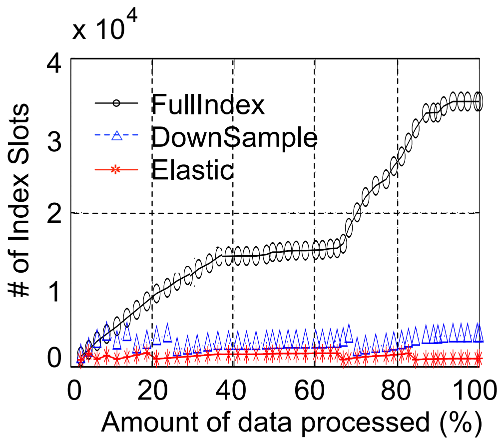


Fig. 7: Index usage comparison.

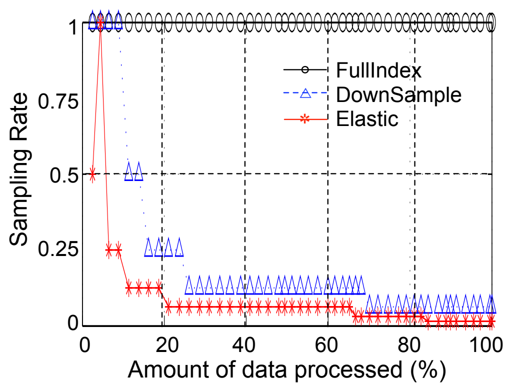


Fig. 8: Sampling Rate comparison.

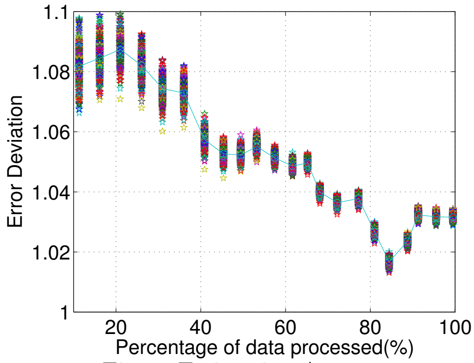


Fig. 9: Estimation Accuracy.

By using this criterion, we make more fairly comparisons among EAD and the other two solutions, as shown in Fig. 6. It shows that `Elastic` outperforms both `Downsample` and `FullIndex` on efficiency. Notice that `Elastic` always yields a higher efficiency, almost 4 times of that from `Downsample` and 30 times of that from `FullIndex`. This is because that its elastic feature enables it to utilize as little memory space as possible to detect enough duplicate data as required, avoiding memory waste as the other two do.

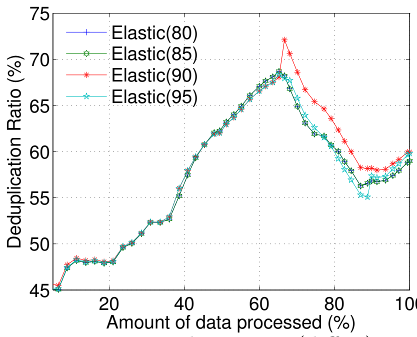


Fig. 10: Dedup Ratio (diff Γ).

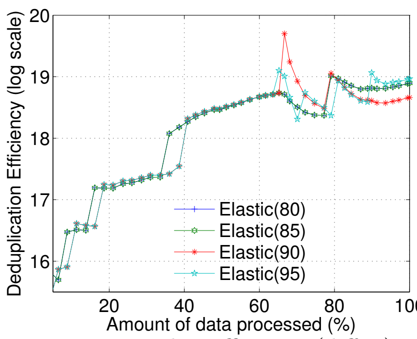


Fig. 11: Dedup efficiency (diff Γ).

5.2 Monitoring Accuracy

EAD can work properly only when it is able to accurately monitor the real time deduplication efficiency. As the criteria of judging deduplication performance, estimated duplication rate is supposed to be as accurate as possible. Otherwise, elasticity might bring unexpected effect on the performance if it makes an inappropriate decision for index scaling up. Fig. 9 shows the accuracy of monitored deduplication ratios during the backup process. 500 independent tests were conducted on the dataset. We consider the ratio of estimated deduplication ratio in EAD to that in `FullIndex` as error deviation, which indicates the real time accuracy of monitoring. We can see that initially the error deviation is at most 10% , but as more data comes in, the deviation reduces to 2%, which offers a reliable criterion for evaluation on system performance.

5.3 Performance of Different EAD Parameters

We also conducted experiments to determine the impact of different parameters on our EAD algorithm under multiple clients running heterogeneous workloads such as video streaming, file backup, big data processing applications, and etc.

■ **Impact of Γ.** The parameter Γ represents the system's tolerance to missing duplicate data. The higher

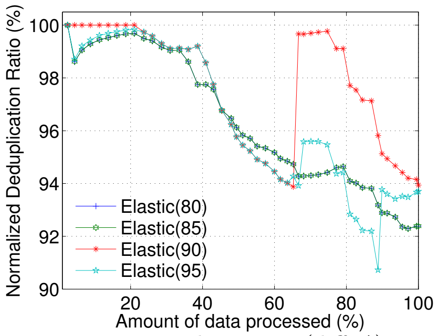


Fig. 12: Dedup Ratio (diff Δ).

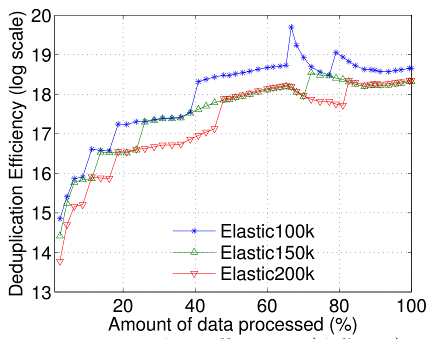


Fig. 13: Dedup efficiency (diff M_I).

the Γ is, more sensitive it will be to trigger the RAM scaling up, and vice versa. However, an inappropriately large Γ will also lead to a high scale-up frequency due to workload bursties[22]. Thus, in order to investigate how to balance the performance and sensitivity, we conduct sensitivity analysis on different values of Γ, as shown in Fig. 10, where the initial index entry slots are 100K and $\Delta = 2$. We see that a higher Γ has a higher deduplication ratio, though $\Gamma = 0.95$ case also yields the highest overall efficiency. This implies that EAD is able to achieve both good deduplication ratio and efficiency (Fig. 11). This can be seen from Fig. 10, where there is a nearly 30% of difference on deduplication ratio between the case of trigger value as 60% and 90%.

■ **Impact of Δ.** The Δ parameter indicates the magnitude of both sampling rate and RAM incrementation. A Δ value of two, for example, means that the sampling rate doubles during the scaling operation. Fig. 12 shows the improvement on deduplication ratio under different scaling parameter policies. In general, a higher Δ will aggressively expand RAM space during scaling up and will also help to explore more duplicate data after scaling up as shown in Fig. 12. This is because of its higher sampling rate after resampling.

■ **Impact of M_I.** The M_I parameter is the initial size of the memory for the index. Fig. 13 shows the *Deduplication Efficiency* of EAD with different initial memory sizes, which indicates that the most conservative RAM initialization case claims the highest deduplication efficiency. Thus, EAD provides well balance between RAM and storage savings.

■ **Impact of SSD and RAM sizes.** To evaluate the effectiveness of assignment of SSDs into the EAD storage system, the RAM size is fixed to hold 10k index entries, and the SSD size is varying from 125GB to 4000GB. Different caching algorithms (e.g., LRU [23], CLOCK [24], ARC [25], CAR and CART [26]) are used to manage pages in SSDs. Fig. 14 shows IO hit ratios under different SSD sizes when RAM is set to store at most 10k index entries. Fig. 15 shows the corresponding normalized IO operation costs under different SSD sizes, where the cost under our RAM-HD design is used as the baseline.

We first observe that advanced algorithms like ARC, CAR and CART have higher hit ratio and lower IO cost than naive algorithms like LRU and CLOCK. We can see that our design is able to save almost 75% of IO operation costs compared to our RAM-HD design. This is because of their enhanced methods to avoid being flushed by IO spikes.

We also observe that, although in general, the larger the SSD is, high hit ratio and low IO cost it will obtain (i.e., the red rectangle in Fig. 14 and Fig. 15). However, the performance improvement brought by larger SSD size is not a linear function of SSD size and price. This implies that in real implementation, it is necessary to find a sweet spot, where EAD has sufficient capacity to hold active working sets of all traces.

We further investigate the impact of both RAM and SSD sizes during the scaling up process under more than 120 virtual machines. Fig. 16 and Fig. 17 show

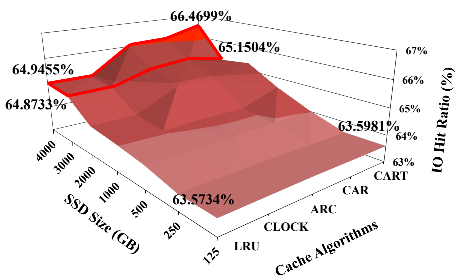


Fig. 14: Hit ratios (diff cache algorithms).

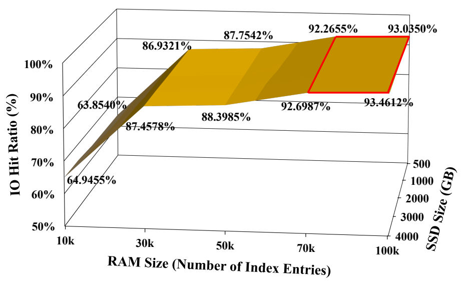


Fig. 16: Hit ratios (diff SSD & RAM sizes).

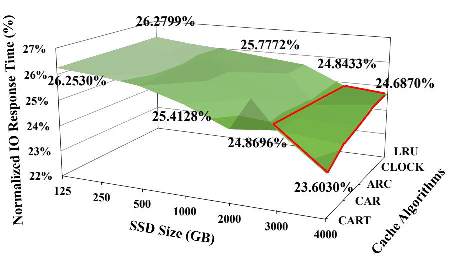


Fig. 15: Norm. IO resp. time costs (diff cache algorithms).

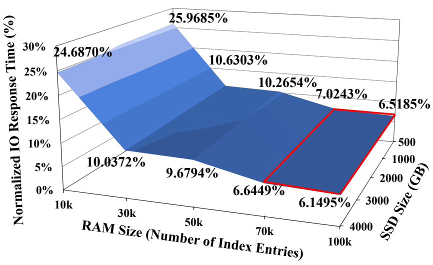


Fig. 17: Norm. IO resp. time costs (diff SSD & RAM sizes).

IO hit ratios and normalized IO operation costs (i.e., RAM-HD design is used as the baseline) under different amounts of RAMs and SSDs during a real scaling up progress, respectively. We see that increasing RAM and SSD size can dramatically improve IO hit ratios. We also observe that the IO cost is more sensitive to the change of RAM size than SSD. In the future, this observation can be used to develop a metics (e.g., a weight function of performance improvement and scaling up cost), such that EAD can use that to dynamically make decisions to add more RAMs or SSDs during runtime.

6 Related Work

Numerous research has been done to improve the performance of finding duplicate data. Work by [27] focused on techniques to speed up the deduplication process. Researchers have also proposed different chunking algorithms to improve the accuracy of detecting duplicates [28][29][30][31][32]. Other research considers the problem of deduplication of multiple datatypes [33][12].

These of researches are complementary to our work, can be easily incorporated into our solution. ChunkStash [34] also attempted to speedup inline storage deduplication by indexing a small fraction of chunks per container in the Flash memory. However, it does not consider the adaptive sampling rate to help to absorb bursties from large-scale datacenter I/O streams. [35] further explored the effectiveness of deduplication for large host-side caches in virtualized datacenter environments running dynamic workloads. Nitro [36] is an SSD cache design with adjustable deduplication, compression, and large replacement units. It evaluates the trade-offs between data reduction, RAM requirements, SSD writes, and storage performance. Our previous work [37] presented a basic model of centralized data deduplication using elasticity feature of cloud computing, however we did not utilize Flash resources in that work to improve the I/O performance. In this paper, we present a deduplication framework that takes advantage of a multi-tier storage architecture consisting of RAM, SSD and HD. We further provide mathematical proof of our algorithm.

The ever increasing amounts of data coupled with the performance gap between in-memory searching and disk lookups, mean that increasingly, disk IO has become the performance bottleneck. Recent deduplication researches have focused on addressing the problem of limited memory. Study [38] proposed integrated solutions which can avoid disk IOs on close to 99% of the index lookups. However, [38] still puts index data on the disk, instead of memory. Estimation algorithms like [39] can be used to improve the performance by reducing total number of chunks, but the fundamental problem remains as the amount of data increases. Other existing research in this area have proposed different sampling algorithms to index more data using less memory: [6] introduces a solution by only keeping part of chunks' information in the index; and [7] proposes a more advanced method based on the work in [6] by deleting chunks' fingerprints (FPs) from the index when it's approaching fullness.

Many research works have been done to investigate the problem about how to best utilize the SSD resources as a cache-based secondary-level storage system or integrated with HDD as a hybrid storage system. Some conventional caching policies [23][40][41][42] such as LRU and its variants maintain the most recent accessed data for future reuse while some other works intended to design a better cache replacement algorithm by considering frequency in addition to recency [25][43]. [44] presented a SSD-based multi-tier solutions to perform dynamic extent placement using tiering and consolidation algorithms. [45] presented a new VMware Flash Resource Manager, named, which considers of both performance and incurred cost for managing Flash resources, and updates the content of SSDs in a lazy and asynchronous mode. Studies [46] [47] [48][49][50][51][52] investigated SSD and NVMe storage-related resource management problems, such as how to reduce the total cost of ownership and how to increase the Flash device utilization. Recently, [53] proposed a hybrid elasticity approach that takes into account both the application performance and the resource utilization to leverage the benefits of both approaches. Study [54] designed a secure ciphertext deduplication scheme based on a classical CP-ABE scheme by modifying the construction with a recursive algorithm, eliminating the duplicated secrets and adding additional randomness to some certain ciphertext. Study [55] investigated the difference between inline and offline deduplication algorithms, and proposed a collective inline memory contents deduplication proposal algorithm

7 Conclusions

This paper presents a deduplication framework that takes advantage of a multi-tier storage architecture consisting of RAM, SSD, and HD, and the rapid scalability capabilities of a virtualized cloud environment. Our elasticity-aware deduplication (EAD) solution balances both deduplication performance and memory size allocation to ensure effective use of cloud resources. We evaluated EAD using real trace driven experiments, and the results indicate that EAD save at least 74% of overall IO access cost compared to the traditional design.

Meanwhile, our EAD is able to detect more than 98% of all duplicate data, but it only consumes less than 5% of expected memory space. In the further, we plan to implement EAD to a larger cluster with multiple deduplication servers. We also plan to improve EAD by adding one more NVMe disks tier.

Acknowledgements

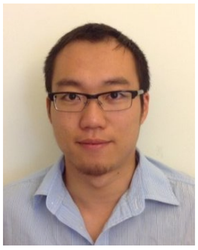
This research was supported in part by National Science Foundations grants CNS-1527346, CNS-1618398, and CNS-1452751, and AFOSR grant FA9550-14-1-0160.

References

1. D. Geer, IEEE Transactions on Computer (2008)
2. M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhang, Y. Tan, in *13th USENIX Conference on File and Storage Technologies (FAST 15)* (2015), pp. 331–344
3. Y.J. Fu, N. Xiao, X.K. Liao, F. Liu, Journal of Computer Science and Technology (JCST) **28**(6), 1012 (2013)
4. B. Berliner. Multi-tier cache and method for implementing such a system (1998). US Patent 5,787,466
5. R.P. Spillane, P.J. Shetty, E. Zadok, S. Dixit, S. Archak, in *Proceedings of the 2nd ACM Symposium on Cloud Computing* (ACM, 2011), p. 1
6. M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, P. Camble, in *Proccedings of the 7th conference on File and storage technologies* (2009)
7. F. Guo, P. Efstathopoulos, in *Proceedings of the 2011 USENIX conference on USENIX annual technical conference* (2011)
8. A. Adya, B. di, et al., ACM SIGOPS Operating Systems Review (2002)
9. G. Forman, K. Eshghi, S. Chiocchetti, in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining* (2005)
10. U. Manber, et al., in *Proceedings of the USENIX winter 1994 technical conference* (1994)
11. A.Z. Broder, in *Sequences II* (Springer, 1993), pp. 143–152
12. D. Bhagwat, K. Eshghi, D.D. Long, M. Lillibridge, in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems* (2009)
13. M. Dutch, in *SNIA Data Management Forum* (2008), p. 7
14. C. Kim, Park, et al., in *Networked Computing and Advanced Information Management (NCM)* (2011)
15. Q. Yang, J. Ren, in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on* (IEEE, 2011), pp. 278–289
16. D. Harnik, O. Margalit, D. Naor, D. Sotnikov, G. Vernik, in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on* (2012)
17. M. Hibler, L.d. Stoller, et al., in *USENIX Annual Technical Conference, General Track* (2003)
18. B.H. Bloom, Communications of the ACM (1970)
19. Wikimedia Downloads Historical Archives (Accessed in 04/2013). `http://dumps.wikimedia.org/archive/`
20. OpenfMRI Datasets (Accessed in 05/2013). `https://openfmri.org/data-sets`
21. G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, W. Hsu, in *FAST*, vol. 12 (2012), vol. 12
22. R. Zhou, M. Liu, T. Li, in *workload characterization (IISWC), 2013 IEEE International Symposium on* (IEEE, 2013), pp. 98–108
23. E. O'Neil, P. O'Neil, G. Weikum, in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data* (Washington, DC, 1993), pp. 297–306
24. L.A. Belady, IBM Systems journal **5**(2), 78 (1966)
25. N. Megiddo, D. Modha, in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (San Francisco, CA, 2003), pp. 115–130
26. S. Bansal, D.S. Modha, in *Proceedings of the 2th USENIX Conference on File and Storage Technologies*, vol. 4 (2004), vol. 4, pp. 187–200
27. A. Sabaa, P.d. Kumar, et al. Inline Wire Speed Deduplication System (2010). US Patent App. 12/797,032
28. L.L. You, C. Karamanolis, in *Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies* (2004)
29. E. Kruus, C. Ungureanu, C. Dubnicki, in *Proceedings of the 8th USENIX conference on File and storage technologies* (2010)
30. J. Min, D. Yoon, Y. Won, IEEE Transactions on Computers (2011)
31. A. Muthitacharoen, B. Chen, D. Mazieres, in *ACM SIGOPS Operating Systems Review* (2001)
32. K. Eshghi, H.K. Tang, Hewlett-Packard Labs Technical Report TR (2005)
33. W. Xia, H.d. Jiang, et al., in *Proceedings of USENIX annual technical conference* (2011)
34. B.K. Debnath, S. Sengupta, J. Li, in *USENIX annual technical conference* (2010)
35. J. Feng, J. Schindler, in *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)* (IEEE, 2013), pp. 1–6
36. C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, G. Wallace, in *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (2014), pp. 501–512
37. Y. Wang, C.C. Tan, N. Mi, in *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on* (IEEE, 2014), pp. 785–792
38. B. Zhu, K. Li, H. Patterson, in *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008)
39. G. Lu, Y. Jin, D.H. Du, in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)* (2010)
40. M. Kampe, P. Stenstrom, M. Dubois, in *Proceedings of the 1st conference on Computing frontiers* (Ischia, Italy, 2004), pp. 181–191
41. T. Johnson, D. Shasha, in *Proceedings of the 20th International Conference on Very Large Data Bases* (San Francisco, CA, 1994), pp. 439–450
42. Y. Zhou, J. Philbin, K. Li, in *Proceedings of the 2001 USENIX Annual Technical Conference* (Boston, MA, 2001), pp. 91–104
43. D. Lee, J. Choi, J.H. Kim, S. Noh, S.L. Min, Y. Cho, C.S. Kim, IEEE Transactions on Computers **50**(12), 1352 (2001)
44. J. Guerra, H. Pucha, J. Glider, W. Belluomini, R. Rangaswami, in *Proceedings of the 9th USENIX Conference on File and Storage Technologies* (San Jose, CA, 2011)
45. J. Tai, D. Liu, Z. Yang, X. Zhu, J. Lo, N. Mi, Cloud Computing, IEEE Transactions on (99), 1 (2015)
46. Z. Yang, M. Awasthi, M. Ghosh, N. Mi, in *2016 IEEE 8th International Conference on Cloud Computing Technology and Science* (IEEE, 2016)

47. Z. Yang, J. Tai, J. Bhimani, J. Wang, N. Mi, B. Sheng, in *35th IEEE International Performance Computing and Communications Conference (IPCCC)* (IEEE, 2016)
48. Z. Yang, M. Hoseinzadeh, A. Andrews, C. Mayers, D.T. Evans, R.T. Bolt, J. Bhimani, N. Mi, S. Swanson, in *36th IEEE International Performance Computing and Communications Conference (IPCCC)* (IEEE, 2017)
49. J. Roemer, M. Groman, Z. Yang, Y. Wang, C.C. Tan, N. Mi, in *2014 IEEE 11th International Conference on Mobile Ad Hoc and Sensor Systems* (IEEE, 2014), pp. 702–707
50. J. Bhimani, Z. Yang, N. Mi, J. Yang, Q. Xu, M. Awasthi, R. Pandurangan, V. Balakrishnan, in *IEEE Transactions on Multi-Scale Computing Systems (TMSCS)*
51. J. Bhimani, J. Yang, Z. Yang, N. Mi, N.K. Giri, R. Pandurangan, C. Choi, V. Balakrishnan, in *36th IEEE International Performance Computing and Communications Conference (IPCCC), Poster Paper* (IEEE, 2017)
52. Z. Yang, M. Hoseinzadeh, P. Wong, J. Artoux, C. Mayers, D.T. Evans, R.T. Bolt, J. Bhimani, N. Mi, S. Swanson, in *36th IEEE International Performance Computing and Communications Conference (IPCCC)* (IEEE, 2017)
53. S. Farokhi, P. Jamshidi, E.B. Lakew, I. Brandic, E. Elmroth, Future Generation Computer Systems **65**, 57 (2016)
54. H. Tang, Y. Cui, C. Guan, J. Wu, J. Weng, K. Ren, in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (ACM, 2016), pp. 59–70
55. B. Nicolae, in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on* (IEEE, 2013), pp. 19–28

Zhengyu Yang is a Ph.D. candidate at Northeastern University (Boston USA), under the supervision of Prof. Ningfang Mi. He graduated from the Hong Kong University of Science and Technology with a M.Sc. degree in Telecommunications, and he obtained his B.Eng. degree in Communication Engineering from Tongji University (Shanghai, China). His research interests are cache algorithm, deduplication, cloud computing, datacenter storage and scheduler, and Spark optimization.



Yufeng Wang graduated from Temple University with a master degree majoring in Computer and Information Sciences under the supervision of Dr. Chiu C. Tan. He graduated from University of Electronic Science and Technology of China in 2012, majoring in Communication Engineering. His current research mainly focuses on efficient and secure data processing in cloud computing, mobile cloud computing, elastic memory management and security enhancement.

Janki Bhimani is Ph.D. candidate working with Prof. Ningfang Mi at Northeastern University, Boston. Her current research focuses on performance prediction and capacity planning for parallel computing heterogeneous platforms and backend storage. She received her M.S. (in 2014) from Northeastern University in Computer Engineering. She received her B.Tech. (in 2013) from Gitam University, India in Electrical and Electronics Engineering.

Chiu C. Tan is an Assistant Professor in the Department of Computer and Information Sciences at Temple University. He is a member of the Center of Network Computing. He is also the director of the Temple University CIS Department REU program. His research interests lies in next generation computing systems, mobile and embedded sensor devices, high bandwidth wireless coverage, and cloud computing infrastructure.

Ningfang Mi is an Associate Professor at Northeastern University, Boston. She received her Ph.D. degree in Computer Science from the College of William and Mary, VA. She received her M.S. in Computer Science from the University of Texas at Dallas, TX and her B.S. in Computer Science from Nanjing University, China. Her current research interests are capacity planning, MapReduce/Hadoop scheduling, cloud computing, resource management, performance evaluation, workload characterization, simulation and virtualization.

0.1 **Appendix Proof and Error Analysis of Expectation of Duplication Ratio (EDR)**

Real-world workloads in enterprise environments have different I/O behaviors, but their pattern can be regresses to some long-term predicable streams. This can be regarded as a super set of number of n_s homogeneous T-size segments. This stream can be modeled by an *average* segment (Seg), such that any results which holds on this average segment also holds for the entire stream, i.e., $EDR_{Cur}(Stream) \approx DR_{Cur}(Stream)$

$$\lim_{n_s \to \infty} DR_{Cur}(Stream) = DR(Seg). \tag{1}$$

Therefore, we can reduce the problem from "level 3" to "level 2". In this section, we demonstrate how $EDR(Seg)$ is close to $DR_(Seg)$ during runtime. We first make the following definitions:

Table 1: Four cases with different sampling degrees and *dupSet* sizes

dupSet Size	Fully Sampled	Partially Sampled
Equal Size	(1) Accurate	(3) Accurate
Non-equal Size	(2) Approximate	(4) Approximate

[Definition 1] dupSet: We first define duplicated set ("dupSet") as the set of same-value chunks in a segment. For example, in Fig. 1 case 2, there are four dupSets, i.e.,

$$dupSet_1 = \{1, 1, \}, dupSet_2 = \{2, 2, 2, 2, 2, 2\},$$

$$dupSet_3 = \{3, 3, 3\}, dupSet_4 = \{4, 4, 4, 4, 4\}. \tag{2}$$

[Definition 2] Fully and Partial Sampled Segments: For each dupSet d_i among T chunks in a segment, if at least one chunk per dupSet is sampled in $\mathbb{B}$, then this segment is considered *fully sampled*; otherwise it is *partially sampled*. Note that the worst case of the partially sampled segment is non-samples, which will happen if $\kappa > 0$. Based on this definition, we can divide the problem into four cases as shown in Tab 1. Fig. 1 also shows examples for each case. Later we prove that Case 1 and 3 are accurate, while Case 2 and 4 are with known estimation errors. Here we also show the probabilities of each cases. Given the size of each segment T, size of each dupSet d_i, sample number $\kappa (\geq n)$, and total number of dupSets n, the probability that a segment is fully sampled is:

$$P_{FullSmp} = \frac{\prod_{i=1}^{n} (C_1^{d_i})}{C_\kappa^T} = \frac{(C_1^d)^n}{C_\kappa^T} = \frac{d^n}{C_\kappa^T}, \tag{3}$$

and the probability that a segment is partial sampled is:

$$P_{PartSmp} = 1 - \frac{d^n}{C_\kappa^T}. \tag{4}$$

[Definition 3] Estimation Error: To evaluate the estimation accuracy, we define the estimation error δ which is the distance between EDR and DR:

$$\delta = \left| EDR(Seg) - DR(Seg) \right|, \tag{5}$$

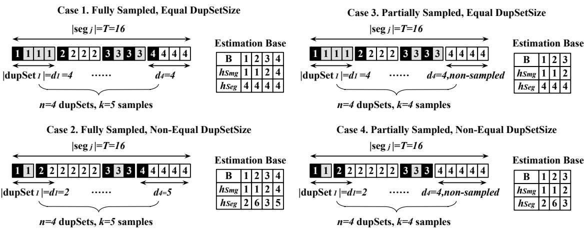


Fig. 1: Examples of different sampling scenarios.

where $\delta \in [0, +\infty)$, and the less δ the more accuracy EDR is. When $\delta = 0$, the estimation is fully accurate. Based on these assumption and definitions, we now calculate the δ of each case:

[Case 1] Fully Sampled, Equal DupSet Size:
We first calculate the actual DR_{Cur} of a segment:

$$DR_(Seg) = 1 - \frac{n}{T} = 1 - \frac{1}{d}. \tag{6}$$

We then investiage the EDR. Since the "fully sampled" case ensures there is at least one sample per dupSet, we can denote $\kappa = n + \varepsilon$, where ε ($\in [0, T - n]$) is the redundant samples that are duplicated with existing one sample per dupSet. We further let each dupSet d_i being assigned $\kappa_i = 1 + \varepsilon_i$ samples, where ε_i is the number of redundant samples of d_i, and $\sum_{i \in \mathbb{B}} \varepsilon_i = \varepsilon$. Fig. 1 Case 1 illustrates an example, where $\kappa_3 = 1 + \varepsilon_3 = 1 + 1 = 2$. Therefore, $EDR(Seg)$ can be calculated as:

$$EDR(Seg) = 1 - \frac{1}{\kappa} \sum_{i \in \mathbb{B}} (\frac{h_{Smp_i}}{h_{Seg_i}}) = 1 - \frac{1}{\kappa} (\sum_{i \in \mathbb{B}} \frac{1}{d_i} + \sum_{i \in \mathbb{B}} \frac{\varepsilon_i}{d_i}). \tag{7}$$

Since in Case 1 all dupSets have the same size ($d_i = d$), Eq 7 can be simplified as:

$$lhs = 1 - \frac{1}{\kappa} (\frac{1}{d} \cdot n + \frac{1}{d} \cdot \varepsilon) = 1 - \frac{1}{\kappa d} (n + \varepsilon) = 1 - \frac{\kappa}{\kappa d} = 1 - \frac{1}{d}. \tag{8}$$

δ=0, which proves that our EDR can reflect the duplication ratio of the accumulated workload with 100% accuracy in Case 1.

[Case 2] Fully Sampled, Non-Equal DupSetSize:
It is straightforward to get the real $DR_(Seg)$ of one segment as:

$$DR(Seg) = 1 - \frac{n}{T} = 1 - \frac{1}{\bar{d}}. \tag{9}$$

However, to calculate EDR, we need to divide the "fully sampled" case into two sub cases: (2.1) Exact fully sampled: **each** dupSet has **exactly one** sample in $\mathbb{B}$, and $\kappa = n$; and (2.2) Redundantly fully sampled: **at least one** dupSet has **more than one** samples in $\mathbb{B}$, i.e., $\forall \kappa_i \geq 1, i \in [1, n]$ and $\kappa > n$, where κ_i is the sample number of d_i.

[Case 2.1] Exact Fully Sampled:
Since $\kappa = n$ and $\kappa_i = 1$, we have:

$$EDR(Seg) = 1 - \frac{1}{\kappa} \sum_{i \in \mathbb{B}} (\frac{h_{Smp_i}}{h_{Seg_i}}) = 1 - \frac{1}{n} (\sum_{i=1}^{n} \frac{1}{d_i}). \tag{10}$$

Based on Eq 9 and Eq 10, the estimation error is:

$$\delta = |EDR(Seg) - DR(Seg)| = \left| \frac{1}{\bar{d}} - \frac{1}{n}(\sum_{i=1}^{n} \frac{1}{d_i}) \right|$$

$$= \left| \frac{1}{A(D)} - \frac{1}{H(D)} \right| = \frac{1}{H(D)} - \frac{1}{A(D)}. \quad (11)$$

In Eq 11, we use notation $A(D)$ and $H(D)$ to represent the arithmetic mean and harmonic mean of sample set $D = \{d_i | d_i \in Seg\}$ respectively. It is always true that $0 \leq A(D) \leq H(D)$, so we remove the absolute value sign. Aiming for better performance, we further investigate under what case δ can be minimized. We conduct several experiments tuning dupSet's sizes under the "exact fully sampled in non-equal dupSet" case.

Fig. 2 shows seven representative δ curves with different size of one subSet (subSet A). There are three dupSets A,B,C in the segment with size of T. For each curve, we iterate the size of dupSet B. Obviously dupSet C's size is $(T - |A| - |B|)$. We observe that (1) when the remaining two dupSets (B and C) have exactly same sizes, the δ will be the lowest of that curve; and (2) when all of three dupSets have the same sizes, the δ is the global lowest among all curves. That is to say, the more equalikely of size of each dupSet is, the lower estimation error will be.

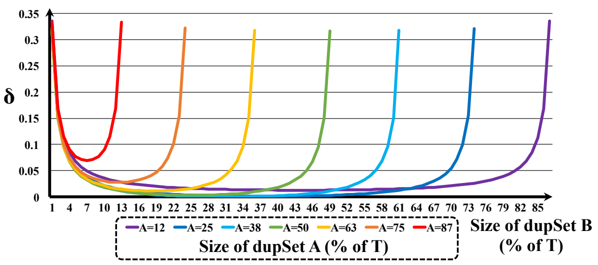


Fig. 2: δ of a three-dupSet segment with different dupSet sizes.

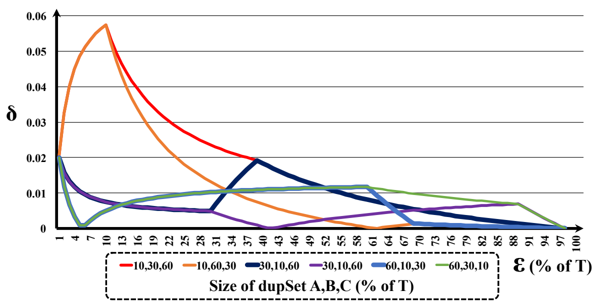


Fig. 3: δ of three-dupSet segment with different redundant samples.

[Case 2.2] Redundantly Fully Sampled:
All dupSets are sampled and some of them have more than one sample, i.e., $\kappa = n + \varepsilon$. We have:

$$EDR(Seg) = 1 - \frac{1}{\kappa} \sum_{i \in \mathbb{B}} (\frac{h_{Smp_i}}{h_{Seg_i}}) = 1 - \frac{1}{n + \varepsilon} (\sum_{i \in \mathbb{B}} \frac{1 + \varepsilon_i}{d_i}). \quad (12)$$

The estimation error is:

$$\delta = |EDR(Seg) - DR(Seg)| = \left| \frac{1}{\bar{d}} - \frac{1}{n + \varepsilon} (\sum_{i \in \mathbb{B}} \frac{1 + \varepsilon_i}{d_i}) \right|. \quad (13)$$

Here we introduce a super set D_{Sup}, which is extension of D, i.e., $D_{Sup} = \{d_i | x_j \in \mathbb{B}, x_j \in d_i\}$. For example, in Fig. 1 Case 2, $D = \{1, 2, 3, 4\}$ and $D_{Sup} = \{1, 2, 3, 3, 4\}$. Therefore, we can use the harmonic mean of D_{Sup} to help to represent the second part of Eq 13. We further use $\Delta(D_{Sup}) = \sum_{i \in \mathbb{B}} (\varepsilon_i \cdot d_i)$, to represent the total size of dupSets that those redundant samples are associated with, where ε_i is the number of redundant samples from d_i. For example, in Fig. 1 Case 2, we have $\Delta(D_{Sup}) = 1 \times 3 = 3$. Therefore, Eq 13 equals to:

$$\delta = \left| \frac{1}{A(D)} - \frac{1}{H(D_{Sup})} \right|$$

$$= \left| [\frac{1}{A(D_{Sup})} - \frac{1}{H(D_{Sup})}] + [\frac{1}{A(D)} - \frac{1}{A(D_{Sup})}] \right|$$

$$= \left| [\frac{1}{A(D_{Sup})} - \frac{1}{H(D_{Sup})}] + [\frac{n}{T} - \frac{n + \varepsilon}{T + \Delta(D_{Sup})}] \right|. \quad (14)$$

As shown in Eq 14, the non-equal case estimation error comes from two parts: (1) the difference between arithmetic and harmonic mean (same as Eq 11); and (2) the number of picked samples which are not necessarily proportional to each corresponding dupSets' sizes (i.e., dupSets' weights in the segment). To further investigate δ, Fig. 3 shows the relationship between δ and different number and distribution of redundant samples in a three-non-equal-dupSet segment example. In this experiment, $|dupSetA| = 10\%T$, $|dupSetB| = 30\%T$,and $|dupSetC| = 60\%T$. We increase the number of samples from each dupSet in different orders. For example, curve with "10, 30, 60" means that firstly each dupSet has one sample, and then we keep adding one more sample of $10\%T$ dupSet until it is fully indexed. Later we repeat it for $30\%T$ and $60\%T$ dupSets until the entire segment is fully indexed. We observe that, if we pick lots of samples from $10\%T$ dupSet at beginning, the error will reach the worst. More the samples from one dupSet, more the weight of that dupSet will be in the final estimation. It will be more accurate if the number of picked samples are proportional to each dupSet, or from a dupSet who shares a relatively big fraction of the segment. Notice that, we also conduct experiments where picking samples with random or some distributions, and consecutively we end up with the same conclusion. Thus, we show part of results to demonstrate the bounds of δ.

[Case 3] Partially Sampled, Equal DupSetSize:

Let T_S be the sampled part of a segment, n' be the number of sampled dupSets, and ε' be the total redundant samples from those sampled dupSets. We have $n' + \varepsilon' = \kappa$. Then EDR of the segment is:

$$EDR(Seg) = 1 - \frac{1}{\kappa} \sum_{i \in \mathbb{B}} (\frac{h_{Smp_i}}{h_{Seg_i}})$$

$$= 1 - \frac{1}{n' + \varepsilon'} (\sum_{i \in \mathbb{B}(T_S)} \frac{1}{d_i} + \sum_{i \in \mathbb{B}(T_S)} \frac{\varepsilon'_i}{d_i}) \quad (15)$$

$$= 1 - \frac{1}{n' + \varepsilon'} (\frac{n'}{d} + \frac{\varepsilon'}{d}) \quad = 1 - \frac{1}{d} = DR(Seg). \quad (16)$$

Therefore, like Case 1, EDR can 100% reflect the $DR(Seg)$ in Case 3.

$$\delta = |EDR(Seg) - DR(Seg)| = 0. \quad (17)$$

[Case 4] Partially Sampled, Non-Equal DupSetSize:

Similar to Case 2, the δ can be calculated as:

$$\delta = |EDR(Seg) - DR(Seg)| = \left| \frac{1}{\bar{d}} - \frac{1}{n' + \varepsilon'} (\sum_{i \in B} \frac{1 + \varepsilon'_i}{d_i}) \right| . \quad (18)$$

Also, Eq 18 can further be divided into two sub cases:
[Case 4.1] Exact Partially Sampled:
To differentiate with fully sampled case, we use the notation $\mathbb{B}(\mathbb{T}_{\mathbb{S}})$ to represent the estimation base that has partial samples of all dupSets (T_S of segment T and T_S is a subset of T). Similar to Case 2.1, we let $D^{T_S} = \{d_i | i \in \mathbb{B}(\mathbb{T}_{\mathbb{S}})\}$ be the sampled dupSet's set. Since each sampled dupSet has only one sample, so $\varepsilon' = 0$, and:

$$\delta = \left| \frac{1}{A(D)} - \frac{1}{H(D^{T_S})} \right|$$

$$= \left| [\frac{1}{A(D^{T_S})} - \frac{1}{H(D^{T_S})}] + [\frac{1}{A(D)} - \frac{1}{A(D^{T_S})}] \right| . \quad (19)$$

[Case 4.2] Redundantly Partially Sampled:
Similar to Case 2.2, let $D^{T_S}_{Sup} = \{d_i | x_j \in \mathbb{B}(\mathbb{T}_{\mathbb{S}}), x_j \in d_i\}$ represent the super set of D^{T_S}. For those sampled dupSets, there exist redundant samples, i.e., $\varepsilon' \neq 0$, as a result we have:

$$\delta = \left| \frac{1}{A(D)} - \frac{1}{H(D^{T_S}_{Sup})} \right|$$

$$= \left| [\frac{1}{A(D^{T_S}_{Sup})} - \frac{1}{H(D^{T_S}_{Sup})}] + [\frac{1}{A(D)} - \frac{1}{A(D^{T_S}_{Sup})}] \right| . \quad (20)$$

We can see from Eq 19 and Eq 20, the estimation error of partially sampled case comes from two source: arithmetic and harmonic means difference, and using partial dupSets to estimate the entire segment. The former has already been explained in Case 2, and the latter is simply based on the coverage of $\mathbb{B}(T_S)$ over the entire segment.