imDedup: a Lossless Deduplication Scheme to Eliminate Fine-grained Redundancy among Images

Cai Deng[#], Qi Chen[#], Xiangyu Zou[#], Erci Xu[*], Bo Tang[†], and Wen Xia[#]

[#]*Harbin Institute of Technology, Shenzhen* [*]*National University of Defense Technology*

[†]*Southern University of Science and Technology*

{*dengcai.319, chenqi8038, xdnzxy, jostep90*}*@gmail.com, tangb3@sustech.edu.cn, xiawen@hit.edu.cn*

***Abstract*—Images occupy a large amount of storage in data centers. To cope with the explosive growth of the image storage requirement, image compression techniques are devised to shrink the size of every single image at first. Furthermore, image deduplication methods are proposed to reduce the storage cost as they could be used to eliminate redundancy among images. However, state-of-the-art image deduplication methods either can only eliminate file-level coarse-grained redundancy or cannot guarantee lossless deduplication.**

In this work, we propose a new lossless image deduplication framework to eliminate fine-grained redundancy among images. It first decodes images to expose similarity, then eliminates fine-grained redundancy on the decoded data by delta compression, and finally re-compresses the remaining data by image compression encoding. Based on this framework, we propose a novel lossless similarity-based deduplication (SBD) scheme for decoded image data (called imDedup). Specifically, imDedup uses a novel and fast sampling method (called Feature Map) to detect similar images in a two-dimensional way, which greatly reduces computation overhead. Meanwhile, it uses a novel delta encoder (called Idelta) which incorporates image compression encoding characteristics into deduplication to guarantee the remaining deduplicated image data to be friendly re-compressed via image encoding, which significantly improves the compression ratio.

We implement a prototype of imDedup for JPEG images, and demonstrate its superiority on four datasets: Compared with exact image deduplication, imDedup achieves a 19%–38% higher compression ratio by efficiently eliminating fine-grained redundancy. Compared with the similarity detector and delta encoder of state-of-the-art SBD schemes running on the decoded image data, imDedup achieves a $1.8\times$–$3.4\times$ higher throughput and a $1.3\times$–$1.6\times$ higher compression ratio, respectively.

I. Introduction

With the rapid growth of the Internet, images consume a large amount of storage space in data centers. For example, the largest Chinese instant messaging service provider Tencent QQ reports that with a daily increase of 300 million new photos, it has already curated a total of 300PB images in 2017 [1]. To reduce storage consumption, every single image has been compressed (e.g., JPEG can provide a 10:1 compression ratio with a slightly perceptible loss in image quality [2]). However, existing image compression techniques do not consider the redundancy among images, which could be reduced significantly. Thus, data deduplication is considered to eliminate the redundancy among different images in the data center.

Data deduplication [3]–[8] is a classical method to eliminate the redundancy among large-scale data. Typically, existing deduplication methods can be classified into two categories: (i) exact methods and (ii) similarity-based methods. Specifically, the exact deduplication methods [9]–[11] compute a cryptographic fingerprint (e.g., SHA-1, MD5) for every deduplication unit (e.g., a file or a data chunk). For deduplication units with identical fingerprints, only one copy will be preserved. Exact deduplication methods only provide a coarse-grained redundancy elimination as they cannot eliminate redundancy whose sizes are smaller than the deduplication unit.

Instead, Similarity-Based Deduplication (SBD) methods [12]–[14] eliminate fine-grained redundancy (e.g., byte-level redundancy) and thus achieve a higher compression ratio. More specifically, the main advancement in SBD is usually using a fixed-size (e.g., 64 bytes) sliding window to walk through each byte and generate a corresponding hash value according to the content of the window. Then, SBD would generate one or several features (e.g., the smallest hash values) to quickly identify files/chunks similar in byte stream, which will be further processed by byte-level delta compression (e.g., Xdelta [15]) to eliminate redundancy for saving space.

Several previous works employ file-level exact deduplication techniques [16]–[18] for image deduplication (called exact image deduplication). However, they only eliminate identical images, which ignores fine-grained redundancy among similar (but not identical) images. Some other works [19]–[21] use fuzzy hash (e.g., average perceptual hash [22]) to achieve file-level image deduplication. They regard visual-similar images as duplicates, and only store one of those similar ones. However, they are lossy approaches, because visual-similar images may not be identical in byte stream so that they cannot restore images the same as that before deduplication. Another notable scheme is PXDedup [23], which decodes JPEG images back to pixels and eliminates the visual duplicate chunks. Generally speaking, PXDedup tries to eliminate more fine-grained redundancy in chunks (usually in KB-level) but could not eliminate redundancy smaller than chunks. This is also a lossy scheme since it does chunk-level visual deduplication.

While similarity-based deduplication (SBD) can provide lossless and fine-grained deduplication, applying SBD on images is challenging. The reasons are as follows. First, compressed images (e.g., JPEG format, the most widely used image compression standard in the world) are not friendly for deduplication. For example, because of image compression, even a small image editing (e.g., watermarking) may change the image binary data significantly. Second, if one tries to

decode all the compressed images to check redundancy, the size of decoded data can be several times of the original data. This casts a demanding requirement for compression technique as deduplication needs to cure data swelling. Third, decoded data increases the workload of running similarity detection and compression, which, along with the extra cost of image decoding, further reduces the whole deduplication throughput.

With the above challenges in mind, we propose a new image deduplication framework in this paper. It first decodes images in a lossless way to expose the data similarity among images, then searches for similar images among the decoded images, and uses a fine-grained compressor to eliminate redundant data between similar image pairs, and finally re-compresses the non-redundant data by the original image compression coding.

Based on this framework, we present a lossless SBD scheme for decoded image data (called **imDedup**). To cure the data swelling caused by decoding, imDedup uses a novel compressor, Idelta, to eliminate fine-grained redundant data among decoded images. Idelta is compatible with image compression coding so that non-redundant data can be directly re-compressed by original image coding, and thus it achieves a much higher compression ratio. Besides, imDedup has a novel similarity detector, which detects similar image pairs for Idelta to compress. This detector is based on a fast sampling method called Feature Map (FM for short), which can fully utilize the information distribution of image data to extract features from a small amount of data, and thus significantly reduce the computation overhead.

In this work, we implement a prototype of imDedup for the most widely used image compression coding JPEG as an example. Note that the deduplication framework of imDedup is portable to other image formats. Extensive experiments on four simulated and real-world datasets show that imDedup detects and eliminates fine-grained redundancy among images successfully. In particular, it achieves a 19%–38% higher compression ratio than exact image deduplication. The evaluation also suggests that our proposed FM-based detector and Idelta compressor are more effective than state-of-the-art SBD schemes: The FM-based detector offers a 1.8×–3.4× higher system throughput; Idelta achieves a 1.3×–1.6× higher compression ratio and a 1.5×–2.6× faster compression speed.

To sum up, our work makes the following contributions:

1. We propose a new image deduplication framework, imDedup, which consists of three key steps: (i) decode images to expose similarity, (ii) find out similar images, (iii) eliminate fine-grained redundancy among those similar images, and then re-compress the remaining data using image encoding. And we implement a prototype of imDedup based on the most widely used JPEG. **To the best of our knowledge, imDedup is the first lossless similarity-based deduplication scheme for images**.
2. We introduce **a novel similarity detection method** using a fast sampling method called *Feature Map* for the 2nd step of imDedup framework. Feature Map fully utilizes the information distribution of image data (i.e., the first several elements of decoded JPEG blocks usually carry more information) to extract features from less data, which significantly reduces computation overhead.
3. We introduce **a novel delta compression method *Idelta*** for the 3rd step of imDedup framework. It eliminates fine-grained redundancy between similar image pairs by friendly incorporating image coding characteristics into delta compression (i.e., treat decoded JPEG data as a stream of two-dimensional blocks instead of bytes in deduplication), which makes our compressor compatible with image formats, and thus efficiently eliminates fine-grained redundancy and compresses non-redundant data.
4. We evaluate imDedup on four simulated and real-world datasets. Extensive experimental results confirm that imDedup achieves **a much higher compression ratio** than classic lossless image deduplication approaches while maintaining a comparable high throughput. On the decoded image data, imDedup also outperforms state-of-the-art SBD schemes at the metrics of compression ratio and system throughput.

The rest of this paper is organized as follows. Section II introduces preliminaries and related work. Section III motivates the design of imDedup. Section IV first describes the workflow of imDedup, then introduces the details of Idelta and the FM-based similarity detector, and lastly shows the implementation details of imDedup. Section V evaluates imDedup by using two simulated datasets and two real-world datasets. Finally, we conclude this work in Section VI.

II. Background and Related Work

A. JPEG based Image Compression

Image compression technology is widely used to reduce storage or transmission cost, and JPEG standard [24] is the most predominant one. The baseline method of JPEG standard is a lossy method based on Discrete Cosine Transform (DCT).

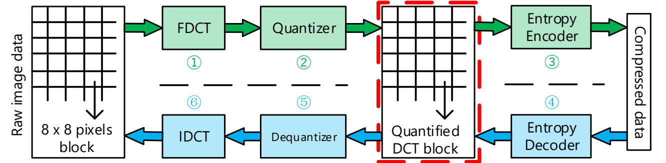
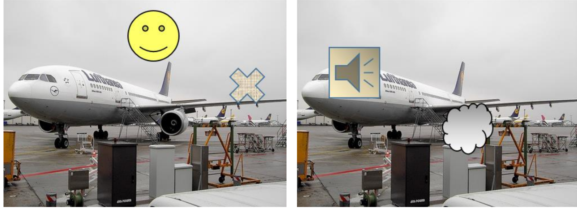


Fig. 1. The general encoding and decoding steps of DCT-based JPEG.

Figure 1 shows the key processing steps of the DCT-based JPEG standard (where FDCT refers to Forwarding DCT and IDCT refers to Inverse DCT). This figure illustrates a special case of single-component (grayscale) image compression. Color image compression can be roughly regarded as compression of multiple grayscale images [24]. In the encoding process, the input image data is regarded as a stream of 8×8 pixel blocks. After the first two steps in Figure 1, every pixel block is transformed into a quantified DCT coefficients block (DCT block for short), which is usually a short signed integer array of 8×8 elements (e.g., in the implementation of *libjpeg-turbo* library [25]). Finally, it compresses these DCT blocks by its entropy encoder further. Note that quantization is a lossy step while entropy encoding is lossless.

We summarize key characteristics of JPEG as below:

(a) A pair of similar images in real-world datasets.



(b) A pair of similar images in simulated datasets.

Fig. 2. Two pairs of similar images.

1. JPEG divides images into blocks and treats image data as a stream of blocks.
2. We can decode a JPEG file into quantified DCT blocks losslessly and use these DCT blocks to reconstruct the same original JPEG file.
3. With the same JPEG encoder, if two raw images have some identical pixel blocks, their corresponding DCT blocks will be also identical.

Now we can find that JPEG shrinks the sizes of JPEG files by eliminating redundant information within single JPEG files. There is still the potential to save more storage space by eliminating redundancy among JPEG files.

B. Data Deduplication

Different from traditional compression methods [26]–[28], data deduplication [3]–[8] is a technology that can detect and then eliminate redundancy among files or chunks. It can be used not only to save storage space [4], [9] by eliminating duplicate data but also to minimize the transmission of redundant data in low-bandwidth network environments [3], [29]. In general, a typical data deduplication system will calculate a cryptographic hash (e.g., SHA-1, MD5) for every deduplication unit (a file or a data chunk) to index and identify duplicates. The deduplication system then removes duplicate units and stores or transfers only one copy of them to save storage space or network bandwidth.

Data deduplication works on general files. All data will be seen as byte streams in deduplication systems. Such characteristic makes it portable and popular in backup storage.

C. Image Deduplication

With the explosive growth of image data size in cloud storage, many deduplication schemes focusing on image data have been proposed. Existing image deduplication schemes can be classified as exact deduplication [16]–[18], [30], [31] and visual deduplication [19]–[21], [32]. The exact image deduplicaiton scheme calculates a cryptographic hash (e.g., SHA-1, MD5) for every image file and seeks exactly identical copies of this image file by its hash. It will then remove each detected duplicate copy and use a pointer to represent it. Exact image deduplication is a coarse-grained scheme, for it can not eliminate redundancy smaller than the file size.

The visual deduplication scheme has a similar workflow to the exact image deduplication scheme. The difference is that visual deduplication calculates a fuzzy hash (e.g., average perceptual hash [22]) for every image file and uses it to detect visually similar images instead of exactly identical images. It then eliminates those detected similar copies and uses pointers to represent them. However, visual deduplication is a lossy scheme, because visual-similar images may not be identical in byte stream so that it may fail to restore images the same as that before deduplication.

One notable scheme is PXDedup [23], which tries to eliminate more fine-grained redundancy in chunk-level rather than file-level. But it is still a coarse-grained scheme because its chunk is still a relatively big unit for images (usually at the KB-level). Note that PXDedup is also a lossy scheme since it does visual deduplication.

To sum up, existing image deduplication schemes have at least one of the following shortcomings: (i) only eliminate coarse-grained redundancy; (ii) can not perform deduplication losslessly. Therefore, existing image deduplication schemes still have the potential to be improved by eliminating fine-grained redundancy losslessly.

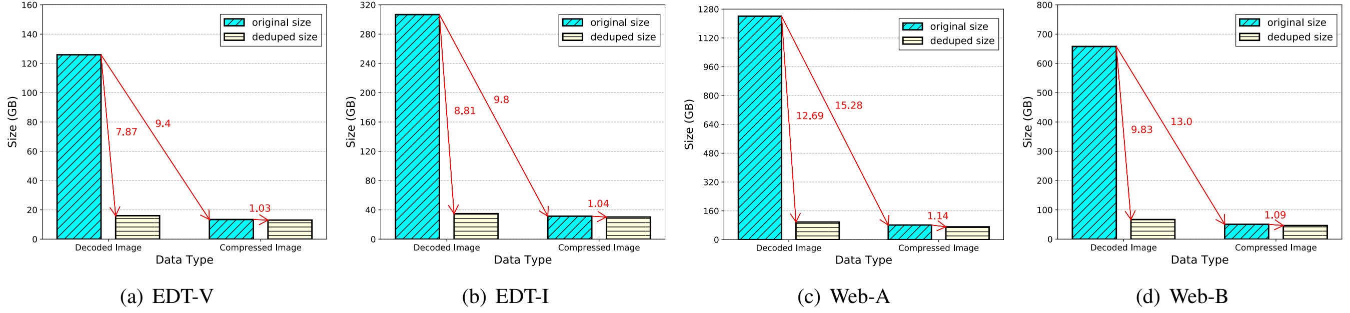
D. Similarity-Based Deduplication

Similarity-Based Deduplication (SBD) [12]–[14] is proposed to eliminate fine-grained redundancy in byte-level losslessly. SBD mainly includes two key techniques, namely, similarity detection (to find similar candidates) and delta compression (to compress similar candidates). In the following content, we will firstly introduce delta compression before similarity detection for the convenience of understanding.

Delta compression eliminates redundancy between the detected similar candidate pairs. In delta compression, the file or chunk to be compressed is called *target*. In similarity detection, the detector will find a similar candidate for target, which is called *base*. Delta compressor then compresses target by only storing the differences between target and base. The differences data stored here is called *delta*.

Delta compression is a special dictionary compression technique. The essence of delta compression is to treat the data as a one-dimensional long string, trying to find redundant substrings between target and base as much as possible. It uses *COPY* instructions to record the redundant strings, and directly stores the non-redundant strings by *INSERT* instructions. After collecting all generated instructions, then we get the final delta, which can be used to losslessly restore the original target.

For delta compression, many approaches are proposed, include Xdelta [15], Zdelta [33], Ddelta [34] and Gdelta [35], among which Xdelta is the most widely used by now.



(a) EDT-V (b) EDT-I (c) Web-A (d) Web-B

Fig. 3. The results of performing traditional SBD on decoded image data and compressed image data of four datasets.

Similarity detection finds similar candidates. A good detector can find candidates with higher similarity degrees, which will lead to more space saving via delta compression.

Currently, many similarity detection approaches have been proposed [12], [13], [36]–[43], among which the most commonly used one is N-Transform proposed by Broder [13], [44], [45]. N-Transform is based on Broder's Theorem [44], [46]:

Broder's Theorem. *Consider two sets A and B, with H(A) and H(B) being the corresponding sets of the hashes of the elements of A and B respectively, where H is chosen uniformly and randomly from a min-wise independent family of permutations. H maps an element in the set to an integer. Let min(S) denote the minimum element of the set of integers S. Then:* $Pr[min(H(A)) = min(H(B))] = \frac{|A \cap B|}{|A \cup B|}$.

Broder's Theorem states that the probability of the two sets *A* and *B* having the same minimum hash element is the same as their Jaccard similarity coefficient [47]. Based on this theorem, Broder proposed a similarity detection method that extracts several digital features from a file or a data chunk. The general workflow of this N-Transform approach [44] include:

1. Compute rolling hashes. Given a chunk (length = L), N-Transform uses a fixed length (e.g., 64 bytes) window to slide the chunk byte by byte, and calculates the Rabin fingerprint [48] of data contents in this window at every position (the Rabin fingerprint computed at position *j* is described as $Rabin_j$).
2. Compute features by linear transformations. To compute *n* features (denoted as $feature_i$, *i=1,2,...,n*), N-Transform needs *n* pairs of randomly pre-defined values (denoted as k_i *and* b_i, *i=1,2,...,n*) to execute linear transformations *n* times. Then we can get *n* features by selecting *n* minimum values from the results of linear transformations. For example, a 32 bits length feature can be calculated as

$$feature_i = Min_{j=1}^{L}\{(k_i \cdot Rabin_j + b_i) mod\ 2^{32}\}. \quad (1)$$

3. Match features to find similar candidates. Given two chunks *A* and *B*, N-Transform compares the ith feature of them respectively (e.g., compare $feature_1$ of *A* with $feature_1$ of *B*, $feature_2$ of *A* with $feature_2$ of *B* and so on). The more identical features they have, the more similar they are regarded as.

According to Broder's Theorem, the Jaccard Similarity of two items in N-Transform can be estimated by

$$Similarity(Item1, Item2) = \frac{k}{n}, \quad (2)$$

where *n* is the number of features and *k* is the number of matched identical features. It means that the more identical features the two candidates have, the more similar they are.

III. Observation and Motivation

Existing lossless image deduplication schemes are all coarse-grained schemes working in the file unit. It can only eliminate totally duplicate images, but can not find and eliminate fine-grained redundancy among similar images (e.g., images in Figure 2). SBD is an eligible lossless fine-grained redundancy elimination scheme, but our evaluation results on four JPEG image datasets suggest that SBD only achieves a compression ratio of 1.03–1.14 on compressed images (see Figure 3, the dataset characteristics will be detailed in Section V-A). This is because image compression coding has destroyed the original semantics of data, which causes the redundant information between compressed images no longer to be similar in binary data and makes finding and eliminating redundancy among images difficult.

To expose data similarity, we decode those images back to DCT blocks before deduplicating them. Then running SBD on the decoded data achieves a much higher compression ratio of 7.87–12.69 as shown in Figure 3, which further suggests that: **fine-grained redundancy among images are widespread but is corrupted by image encoding, and thus image decoding is a necessary step for fine-grained image deduplication**.

However, according to our observations, applying SBD on the decoded images brings two new challenges:

(i) Decoded image data requires a more efficient compressor. According to Figure 3, SBD achieves a high compression ratio on decoded data, but finally stores more data compared with directly re-encoding them by JPEG individually. This is derived from the data explosion caused by image decoding. As shown in Figure 3, data size will be amplified 9.4×–15.28× times after decoding. Thus we need a more efficient compressor with a high compression ratio to offset the extra storage cost caused by image decoding.

(ii) The explosion of data size puts forward a higher requirement to the deduplication system throughput. With

the explosion of data size caused by image decoding, our deduplication approach has to detect and compress redundancy in a much larger amount of data, which introduces performance challenges to the whole system throughput.

The first challenge motivates us to propose **a new deduplication framework for decoded images, which first eliminates redundant data among detected similar images and then uses image encoding to re-compress the remaining non-redundant data**. However, making the remaining non-redundant data compatible with image encoding needs some effort. For example, many lossy image encoding methods, like JPEG [24] and WEBP [49], usually configure two-dimensional blocks as the basic processing unit. However, traditional delta compressors always regard data as one-dimensional byte streams and eliminate redundancy at the byte-level. The mismatching on granularity makes the remaining delta-compressed data hard to be further re-compressed via image encoding. Therefore, our proposed approach no longer regards data as a byte stream but a block stream and eliminates redundancy in the block unit to make the remaining data compatible with image encoding, thus achieving a higher compression ratio.

The second challenge motivates us to design **an image SBD scheme with much higher throughput, including fast similarity detection and delta compression**. Our proposed method takes data as a two-dimensional block stream, which reduces the computation overhead of feature generation and delta compression by reducing the total number of units. It partly offsets the negative impact of data explosion. Besides, we analyze the most time-consuming part of deduplication (i.e., feature generation) and propose a new sampling method to detect similar images, significantly reducing computation overheads and memory accesses.

In this work, since JPEG is the most widely used image format, we take JPEG as an example to implement our image SBD system (called imDedup).

IV. Design and Implementation

A. Overview

In this paper, we propose a novel SBD scheme designed for JPEG files, imDedup. As mentioned in Section III, imDedup works on the decoded JPEG images, and runs a fine-grained SBD, which is different from all previous works. The overall workflow of imDedup is shown in Figure 4, and it mainly consists of the following three modules:

1) **JPEG decoder** decodes JPEG images into DCT blocks to expose data similarity in a lossless way, because DCT blocks can maintain the similarity between similar image pairs and can be used to reconstruct the original JPEG files losslessly (as described in Section II-A). It then sends the decoded images to similarity detector.
2) **Similarity detector** extracts several digital features for each input image (called target) and tries to find the most similar image (called base) for the target according to its features. It uses a novel sampling method, Feature Map, to reduce computation overhead and memory access during feature generation, which significantly improves the similarity detecting speed. It then sends base-target pairs to the Idelta encoder.
3) **Idelta encoder** compresses the input target images with their base images and output the final compressed data. Idelta is a novel delta compressor that eliminates redundancy in block units to make the remaining block data friendly to be re-compressed via JPEG encoding, and thus achieves a higher compression ratio and throughput.

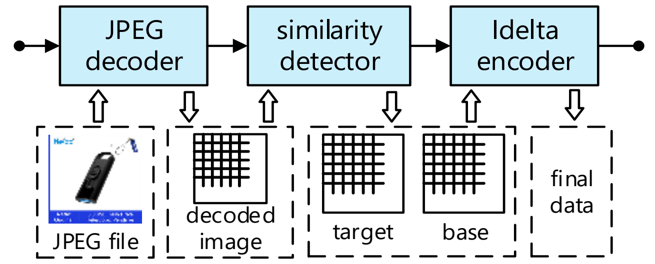


Fig. 4. The overall workflow of imDedup.

For the convenience of understanding, we will first introduce the Idelta encoder, then the similarity detector, and other implementation details in the following subsections, respectively.

B. Idelta Encoder

Overall. In the imDedup framework, the Idelta encoder compresses the input target image with the selected similar base and works in the block unit. As shown in Figure 5, the process of Idelta consists of three key steps: (1) computing the hash digest index for blocks in base; (2) producing the delta of target and base in block unit; (3) compressing the delta data.

In this way, Idelta first eliminates redundancy between similar image pairs in block units and makes sure the remaining data (i.e., delta) still keeps the block structure and is friendly to JPEG format. Then it re-compresses remaining blocks by JPEG encoding to further obtain space saving.

Details of Idelta can be seen in Algorithm 1.

In the first step, Idelta generates a **block index** to label blocks in the base image. Specifically, Idelta calculates a hash digest (e.g., Adler32) for each DCT block in the base image, then records <block's digest, block's position> in the index.

Then, in the second step, Idelta detects redundant blocks in the target image and generates delta data. In this process, Idelta traverses the whole target image and uses the base image's block index to locate duplicate blocks between target and base.

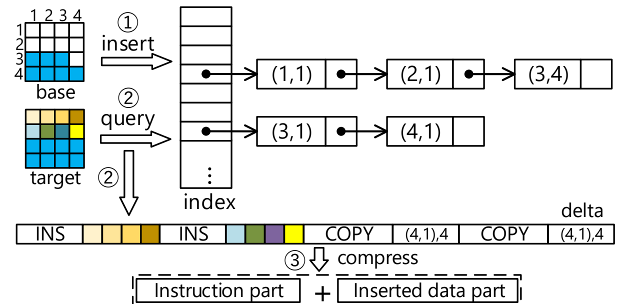


Fig. 5. The workflow of Idelta.

Algorithm 1: Idelta encoder.

```
Input: base & target image blocks, Base & Target; width
       & height of base & target images in block unit,
       w_base, h_base, w_tar & h_tar respectively
Output: compressed target
Index = ComputeIndex(Base, w_base, h_base);
for row = 0; row < h_tar; row++; do
    col ← 0;
    while col < w_tar do
        maxLength ← 0;
        positions ← 0;
        FP ← Adler32(Target[row][col]);
        Candidates ← LookUp(Index, FP);
        for block ∈ Candidates do
            /* Number of identical blocks. */
            length ← Compare(block, Target[row][col]);
            if length > maxLength then
                maxLength ← length;
                positions ← PositionsOf(block);
        if maxLength > 0 then
            WriteInstruction(Copy, positions, maxLength);
            col ← col + maxLength;
        else
            WriteInstruction(Insert, Target[row][col]);
            col ← col + 1;
part_1 ← EntropyEncode(Copy and Insert);
part_2 ← JPEGEncode(Inserted blocks);
return part_1 + part_2;
```

It uses "*COPY* {*starting point, length*}" instructions to replace duplicate blocks and uses *"INSERT* {*data*}*"* instructions to directly store non-duplicate blocks. For example, in Figure 5, we first use two *"INSERT"* instructions to directly store the first two non-duplicate rows from the target image. Then we use two *"COPY* {*(4,1), 4*}*"* instructions (i.e., copy 4 blocks from the 4th row and 1st column of the base image) to store the blue blocks in target. All generated instructions compose the final delta data.

For more details of the second step, Idelta calculates a digest D_i for each block in target and queries base image's block index with the digest D_i to find a base block with the same digest. Then, Idelta will compare them to make sure whether they are really identical. If they are, Idelta will keep comparing the following adjacent blocks of them to find as many duplicate blocks as possible until it meets a mismatch or the end of this row. Idelta will record the first matched block as *starting point* and the following successive matched numbers as *length* to construct a new *COPY* instruction. We call such an instruction a matched choice. Since there may be many blocks having the same digest, one block in target may have several matched choices. To record more duplicate blocks in a single instruction, Idelta will compare all matched choices and select the longest one to be the *COPY* instruction. If a block in target cannot find a duplicate one from the base index, it will be directly stored with an "*INSERT*" instruction.

Finally, in the third step, Idelta will compress the delta data further. It divides the delta data into the instruction

Algorithm 2: Calculate the base index in Idelta.

```
Input: image blocks, Base; width & height of the image, w
       & h
Output: the block index of Base, Index
Initialize(Index);
LastFP ← 0;
for row = 0; row < h; row++; do
    for col = 0; col < w; col++; do
        FP ← Adler32(Base[row][col]);
        if FP != LastFP then
            Insert(Index, FP, <row, col>);
            LastFP ← FP;
return Index;
```

part (i.e., meta data of *COPY* and *INSERT* instructions) and inserted data part (i.e., non-duplicate blocks stored by *INSERT* instructions). The instruction part will be compressed by an entropy encoder (e.g., FSE [50] or Huffman [26]), while the inserted data part will be re-compressed by JPEG encoding.

A Corner Case. Idelta's throughput may suffer from too many matched choices in the second step. For example, in Figure 5, when processing the 3rd row's 1st block of target, we should first find all duplicate blocks in base (i.e., (3,1), (3,2), (3,3), (4,1), (4,2), (4,3), (4,4)) according to the base index. Then, we have to take all of them as starting points for block matching to decide the longest matched choice, which is very time-consuming. However, we can only perform block matching from blocks (3,1) and (4,1) of the base, because the blocks behind them have the same content and will never have longer matched lengths than them.

Therefore, one way to optimize Idelta is that when generating the base index and encountering several adjacent identical blocks in a row, Idelta only inserts the first block into the index to reduce the number of starting points, as shown in Algorithm 2. Consequently, Idelta greatly reduces block matching operations and perform faster block matching. Such optimization may slightly decline the compression ratio of Idelta because of producing more *"COPY"* instructions, but will greatly improve the overall compression speed in this case (see our experiments in Section V-B).

Summary. What makes Idelta special is that it works in block units. Idelta eliminates fine-grained redundancy among images in block units to make sure the remaining delta data still keeps block structure so that it can be re-compressed by JPEG encoding, which brings significant storage saving.

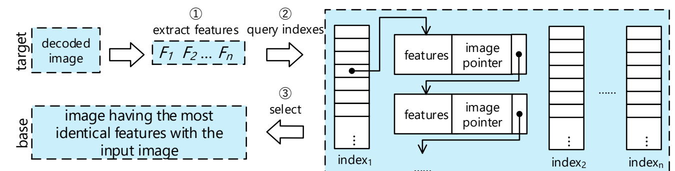


Fig. 6. The workflow of similarity detector in imDedup.

C. Similarity Detector

Overall. The similarity detector aims to find similar candidates for the input images. The workflow of the similarity

Algorithm 3: Similarity detection in imDedup.

Input: decoded image data, $target$; number of features, N; feature indexes, $Index[N]$

Output: the most similar image to $target$, $base$

$Feature[0,...,N-1] \leftarrow FeatureFunction(target)$;
$maxMatch \leftarrow 0$;
$base \leftarrow NULL$;
for $i = 0;\ i < N;\ i++;$ **do**
  $Candidates \leftarrow LookUp(Index[i], Feature[i])$;
  for $image \in Candidates$ **do**
    $match \leftarrow 0$;
    for $j = 0;\ j < N;\ j++;$ **do**
      if $Feature[j] == image.Feature[j]$ **then**
        $match \leftarrow match + 1$;
    if $match > maxMatch$ **then**
      $maxMatch \leftarrow match$;
      $base \leftarrow image$;
  $Insert(Index[i], Feature[i], target)$;
return $base$;

detector is shown in Figure 6, and details are shown in Algorithm 3. To find the most similar candidate, the detector first extracts *n* digital features for every input decoded image. Note that 'feature' is a digital integer extracted from the being detected data to represent its source data, which is commonly used to search for similar items (see 'similarity detection' part of Section II-D). Then the detector will find out the similar candidate who has the most number of identical features with the input image as the base image. To achieve such feature matching, the detector has to maintain a feature index for every feature, namely, a hash table maintaining a feature-images mapping relationship.

Feature Generation. The critical step in similarity detection is generating features for images. Ideally, we expect to generate features that can adequately reflect the characteristics of images as quickly as possible. Figure 7 shows a simple method to calculate features. It uses a sliding window to walk through the whole image and computes the hash (e.g., Rabin) of each window. Then it executes *n* different linear transformations on the generated hashes, and results of each linear transformation are collected in different sets. Finally, the minimum one of each set is selected as a feature, and there will be *n* features calculated for every image.

Fig. 7. A two-dimensional way to generate features.

Fig. 8. FM-based hash calculation for similarity detection.

Here we keep the two-dimensional structure of image data and use a two-dimensional window to extract features because simply treating image data as a byte-stream will lose characteristics on the other dimension. By using a bigger processing unit (i.e., using block units instead of byte units), the calculation of linear transformations is greatly reduced and thus the detector could be faster. Specifically, the DCT block is usually an integer array with 128 bytes (e.g., in the implementation of *libjpeg-turbo* [25]). Suppose there are a total of $m \times m$ blocks in an image. Therefore, the window would slide at most $m \times m$ times in block units but $128 \times m \times m$ times in byte units. So the number of hashes generated in byte units will be 128 times that in block units, which brings more computational overhead in linear transformations. At the same time, the detector will not lose detection accuracy, because redundancy and differences are also appearing in block units according to JPEG's characteristics (see Section II-A).

Hash Calculation. The way to calculate hashes for each sliding window is also critical since these operations are also related to heavy computations and have a great impact on the efficiency of similarity detection. It is feasible to apply the traditional one-dimensional byte-wise hash methods to the two-dimensional window, but it is inefficient. As shown in Table I, the classic Rabin hash [48] used by N-Transform [44] has a complicated calculation process, and it has two significant shortcomings. (1) It has to read the whole block from memory to complete the hash calculation of a block. (2) By sliding the window, most blocks will be included into different windows several times. Therefore, these blocks have to be read from memory repeatedly, which is a heavy overhead.

Here we propose a calculation method based on a designed data structure, Feature Map (FM for short), which works more efficiently. Figure 8 provides an example of how FM-based method works. In this example, the detector calculates *n* features of 16 bits length for every image. First, the detector calculates Feature Map according to the decoded image blocks. Feature Map is a two-dimensional array, and every element in Feature Map is corresponding to a block in the decoded input image (*we call such an element the agent of its corresponding block*). Each agent is determined according to its corresponding block. In Figure 8, we sample the Least Significant Bit of each block's first integer as an agent. After generating Feature Map, the detector uses a sliding window to walk through the Feature Map to acquire hashes according to the content of the window. The agents in a window will be flattened to generate a hash value. As the example in Figure 8, the first window includes agents '0', '0', '0', '1', '0', '1', '0', '0', '1', '0', '0', '0', '0', '0', '0', '0', and thus, the hash value

TABLE I
THE COMPARISON BETWEEN DIFFERENT HASH CALCULATION APPROACHES ON A BLOCK.[a]

Hash Method	Pseudo-code	Computation Overhead	Memory Access
Rabin	$for(i=0;\ i<128;\ i++)$ $tmp = hash_{old} \wedge U(a_i);$ $hash_{new} = (tmp \ll 8) \mid b_i \wedge T(tmp \gg s_1);$	128 ORs, 256 XORs, 256 SHIFTs	256 ARRAY LOOKUPs, 128 READs
Feature Map	$for(i=0;\ i<s_2;\ i++)$ $Map[r][c] = (Map[r][c] \ll 1) \mid (b_i\ \&\ 1);$ $hash_{new} = (hash_{old} \ll s_2) \mid Map[r][c];$	(s_2+1) ORs & SHIFTs, s_2 ANDs	1 ARRAY LOOKUP, s_2 READs

[a]Here a and b denote the sliding out and sliding in byte respectively; s_1 and s_2 are also pre-defined, s_2 is equal to sampling length; U and T are pre-defined arrays; r and c denote the processing row and column respectively; Map is the two-dimensional array of Feature Map.

of the first window is '0001010010000000', which is 0x1480.

Note that this method is scalable as we can use different sizes of window and sampling length for agents. For example, in our real evaluation, we calculate 64 bits length features. So we can choose a 8×8 window with 1 bit length agent, or a 4×4 window with 4 bits length agent (by sampling the LSBs of the first 4 integers of each block to get 4 bits length agent) to generate 64 bits hashes. Algorithm 4 gives the details of how FM-based feature method works with given sizes of window and agent.

Algorithm 4: FM-based feature generation.

```
Input: image blocks, Block; image width & height in block
       unit, w & h; sliding window size, m; sampling agent
       length, s; pre-defined value pairs for linear
       transformation, K[N] & B[N]
Output: N features of m×m×s bits length, Feature[N]
Feature[0, ..., N-1] ← 0;
/* initialize Feature Map. */
for row = 0; row < h; row++; do
    for col = 0; col < w; col++; do
        tmp ← 0;
        for k = 0; k < s; k++; do
            tmp ← (tmp ≪ 1) | (Block[row][col][k] & 1);
        Map[row][col] ← tmp;
/* calculate features by FM. */
for row = 0; row <= h - m; row++; do
    Hash ← 0;
    /* calculate the first hash. */
    for j = 0; j < m; j++; do
        for k = 0; k < m; k++; do
            Hash ← (Hash ≪ s) | Map[row + k][j];
    /* calculate features. */
    for col = 0; col <= w - m; col++; do
        for k = 0; k < N; k++; do
            /* linear transformation. */
            Transform[k] ←
              (K[k] * Hash + B[k]) mod 2^(m×m×s);
            /* minimum selection. */
            if Transform[K] < Feature[k] then
                Feature[k] ← Transform[k];
        /* calculate the next hash. */
        for k = 0; k < m; k++; do
            Hash ← (Hash ≪ s) | Map[row+k][col+m];
return Feature[N];
```

Summary. FM-based detector has four main advantages: (1) It generates features in block unit, which greatly reduces computation overhead by reducing the number of needed linear transformations. (2) FM-based hash calculation doesn't have to read the whole block data from memory. It only has to read the first s (i.e., length of agents) elements of the block when producing the Feature Map, which is memory efficient. (3) FM-based hash could well reflect characteristics because it fully utilizes the information distribution of DCT blocks. In DCT blocks, the first several elements carry more information, since the farther away an item is from the DC term (i.e., the first value) in DCT block, the higher the frequency its corresponding waveform will have and the smaller its amplitude will be [51], and the quantization operation in JPEG encoding will discard these high frequencies and small amplitude coefficients [52]. Therefore sampling from the first s elements will not decline the detection accuracy. (4) In FM-based detector, the decoded image data will be read from memory only once during the feature generation. Specifically, the decoded image block data will be read only when calculating Feature Map. When calculating features, the window is sliding on Feature Map instead of original data so it won't access original data again.

D. Implementation Details

In this subsection, we will introduce more implementation details for imDedup.

The JPEG decoder used in imDedup is based on *libjpeg-turbo* [25], which uses SIMD (Single Instruction Multiple Data) instructions to speed up the JPEG decoding and encoding process. imDedup decodes a JPEG file into two parts: JPEG header and DCT blocks. JPEG header includes some necessary information for decoding and encoding, whose size is much smaller than DCT blocks.

The similarity detector of imDedup extracts 64 bits length features. Note that we pay more attention to luminance components just like how JPEG does and only extract features from luminance data to reduce the computation overhead.

Idelta encoder actually not only compresses the DCT blocks of targets but also uses traditional delta compression to compress the JPEG header of target with that of base. Besides, Idelta will package multiple compressed images before writing them into external storage to avoid excessive I/O requests.

Pipeline concurrency is exploited to speed up deduplication. We use multi-threads for pipelining. Adjacent two pipes maintain a queue together. The upper pipe acts as a producer and the lower one acts as a customer.

TABLE II
THE WORKLOAD CHARACTERISTICS OF FOUR DATASETS.

Nmae	Size	Counts	Type	Average Size
EDT-V	13.4 GB	120,521	simulated	117 KB
EDT-I	31.3 GB	251,725	simulated	130 KB
Web-A	81.2 GB	376,154	real-world	226 KB
Web-B	50.6 GB	2,352,026	real-world	23 KB

V. EVALUATION AND DISCUSSION

In this section, we evaluate our system to select reasonable parameters and explore the comprehensive performance.

A. Experimental Setup

Environment. The experiments are running on a Dell PowerEdge R740 server with 2 Intel Xeon Gold 6130 2.1GHz CPUs. The server is equipped with 128GB DDR4 memory and 4TB SSD and installed with Ubuntu 18.04.2 operating system.

Metrics. Evaluations focus on the following metrics:

- **Compression Ratio** reflects the effect of redundancy elimination. It is defined as $\frac{original\ data\ size}{data\ size\ after\ deduplication}$. A larger compression ratio on the same dataset indicates a better redundancy elimination effect.
- **Deduplication/Restore Throughput** reflects the system throughput and is defined as $\frac{original\ data\ size}{deduplication/restore\ time}$. A larger deduplication throughput on the same dataset indicates a higher efficiency of redundancy elimination.
- **Success Rate** is defined as $\frac{size\ of\ detected\ images}{original\ data\ size} \times 100\%$. It reflects the percentage of detected similar images.
- **Accuracy** measures the similarity of the detected images. We define it as the compression ratio of *the detected images* (i.e., $\frac{size\ of\ detected\ images}{size\ of\ compressed\ detected\ images}$). The larger the accuracy is, the more similar the detected images are.

Datasets. We use four datasets to evaluate imDedup, whose characteristics are shown in Table II.

- **EDT-V:** This dataset is based on the PASCAL VOC2012 dataset [53]. For each original image in VOC2012, we generate a random number of similar copies for it by randomly adding some small patterns. Collect all the generated edited copies (excluding the original image) then we get the EDT-V dataset. A pair of similar images in EDT-V are shown in Figure 2(b).
- **EDT-I:** The generation way of EDT-I is the same as that of EDT-V, except that its original images come from ImageNet 2012 dataset [54]. Note that we only sampled 50,000 images from ImageNet 2012 to generate EDT-I.
- **Web-A:** This dataset is a real-world dataset whose images are downloaded from an e-commerce platform website, which is one of the biggest online shopping platforms around the world. These images are mainly product images, some of them have the same logos on different products or have the different product information such as prices on the same products (e.g., Figure 2(a)).
- **Web-B:** It is also a real-world dataset whose images are downloaded from another e-commerce platform website. It has the similar characteristic with Web-A.

B. Study of Idelta Encoder

In this section, we evaluate performance of Idelta Encoder, and choose the most widely used delta encoding approach, Xdelta [15], as the baseline. Note that *Xdelta+FSE* refers to use Xdelta to produce delta data and then use FSE [50] to further compress the delta data; and *simple Idelta* refers to Idelta without the corner case optimization (introduced in Section IV-B). All these four approaches are working on the same similar pairs of decoded JPEG images generated by the similarity detector. Figure 9 shows the comparison among these four different delta encoders.

Compression Ratio. According to Figure 9(a), Xdelta has the lowest compression ratios on all datasets, since it does not further compress the non-redundant data. *Xdelta+FSE* applies FSE, which is an individual compression, on deduplicated data to improve the compression ratio, but results show that it brings an insignificant improvement. *Xdelta+FSE* achieves a low compression ratio (less than 1), which means the benefits from deduplication and further individual compression in this approach is smaller than that from directly using JPEG encoding, and makes deduplication meaningless.

In contrast, Idelta-based encoders achieve 1.3×–1.6× higher compression ratios, and their compression ratios are all higher than 1.0, which makes an actual improvement on storage reduction. It is because *Idelta* is more friendly and effective to JPEG data. Besides, Figure 9(a) also demonstrates *simple Idelta* has the same compression ratios as *Idelta*, and the corner case optimization does not decline the compression effect.

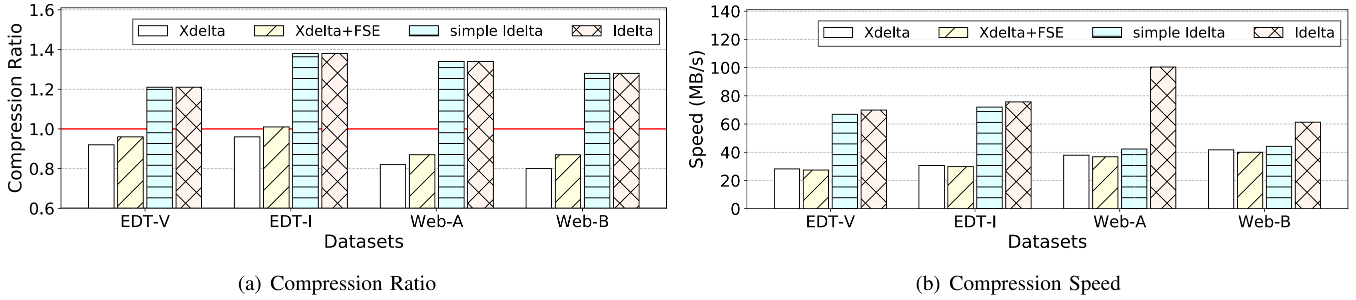
Compression Speed. Figure 9(b) shows the delta compression speeds of the four approaches. Xdelta-based ones have lower compression speeds, since their smaller processing unit increases the computation overhead when searching for redundancy. By comparison, *simple Idelta* uses block as processing unit and achieves 1.1×–2.4× higher speed. Further, *Idelta* achieves 1.5×–2.6× higher speed by reducing useless block comparisons in the corner case. Especially on Web-A and Web-B datasets, there is a speed gap between *simple Idelta* and *Idelta*, and it is because there are many "adjacent identical blocks" (mentioned in Section IV-B) on these datasets.

Summary. Results demonstrate the efficiency of Idelta. Compared with Xdelta, Idelta achieves 1.3×–1.6× higher compression ratios and 1.5×–2.6× higher compression speeds due to using block as processing unit.

C. Study of Similarity Detector

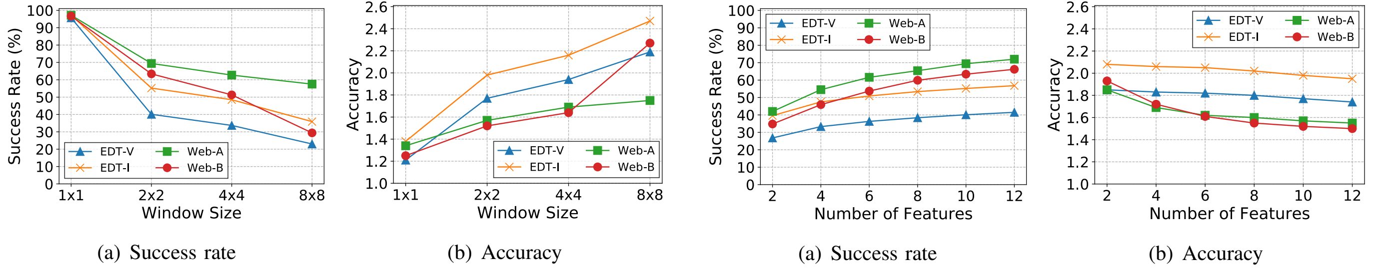
In this subsection, we will evaluate performance of the similarity detector. Configuration of some critical parameters in our FM-based similarity detector will be explored to make a better balance between compression ratio and system throughput. We will also introduce some other approaches as baselines, all of which use Idelta as the delta encoder.

Parameter Configuration: Window Size for FM-based detector. We evaluate four different window sizes in imDedup, which are 1×1, 2×2, 4×4, and 8×8 agents. Figures 10(a) and 10(b) show the the success rate and accuracy on 1×1, 2×2, 4×4, and 8×8 window sizes. On all four datasets,



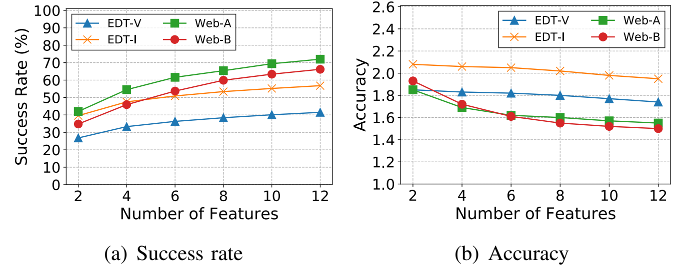
(a) Compression Ratio (b) Compression Speed

Fig. 9. Comparison among different delta encoders for eliminating fine-grained redundancy between images.



(a) Success rate (b) Accuracy

Fig. 10. Impact of window size on success rate and accuracy.



(a) Success rate (b) Accuracy

Fig. 11. Impact of feature number on success rate and accuracy.

the success rates decrease with the increase of window size, but accuracy increases accordingly. This is because a smaller window covers less data, and it is easier to find a similar match, which leads to a higher success rate. Besides, though a bigger window size will bring more difficulties on finding similar candidates, it also filters matches with low similarity degree and thus achieves a higher accuracy. If we use a large window size, which is equal to the image size, our approach will degenerate into file-level exact deduplication.

Figure 12 shows window sizes' impacts on the whole imDedup system. With the increase of windows size, the whole system throughput is also increasing since there will be lower success rates and fewer similar candidates are detected & compressed. When window size is bigger than 2×2, compression ratio declines with the increase of window size, because using a bigger window size may not detect and eliminate redundancy with lower similarity degree in imDedup. For example, in Figure 15(a), if detector generates features using the depicted window, this pair of images will not be regarded as similar ones when using a big window size, and redundancy between them will not be eliminated.

At the same time, results show using 1×1 window produces an inferior compression ratio than using 2×2 window. That is because using a too-small window size brings too many similar candidates, leading to misjudgments sometimes. For example, in Figure 15(b), the first two images are misjudged as the most similar ones when using smaller windows.

Finally, we learn that using a bigger window size in FM-based detector always leads to a higher system throughput, and a 2×2 window always has a higher compression ratio.

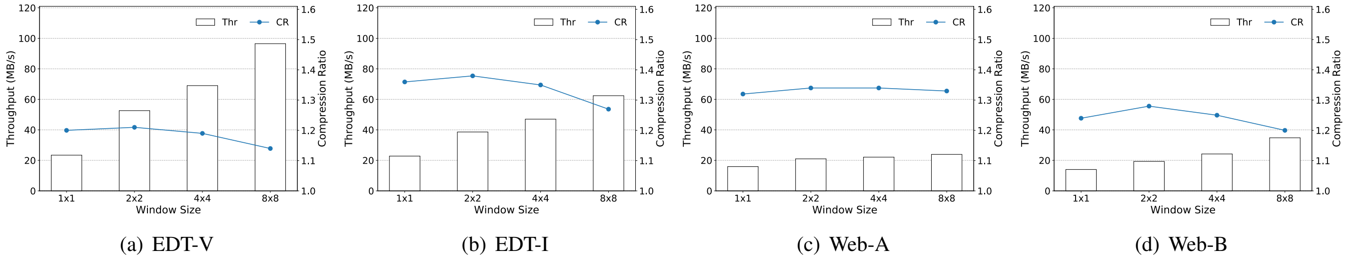
Parameter Configuration: Feature Number for FM-based detector. Here we explore how many features generated for each image is reasonable, and configure a 2×2 window size according to the above studies on window size. Figures 11(a) and 11(b) show the success rate and accuracy of imDedup using different feature numbers for similarity detection. With the growth of feature number, success rate is also increasing. This is because using more features per image makes the detector easier to find a feature match. Therefore, using more features can detect more image pairs with lower similarity degrees, and also leads to worse accuracy.

Figure 13 shows the compression ratios and system throughputs of imDedup using different feature numbers. With the increase of feature number, imDedup's compression ratio is also increasing due to that imDedup detects more matches and thus can eliminate more redundancy. However, using more features bring more computation overhead, since similarity detector needs to generate more features and delta encoder needs to process more similar pairs. Therefore, with the increase of feature number, the system throughput is decreasing.

Finally, we learn that a bigger feature number leads to a higher compression ratio, but a lower deduplication throughput, as shown in Figure 13. In imDedup we set feature number to 10, since the changes of compression ratio tend to be flat when feature number is bigger than 10.

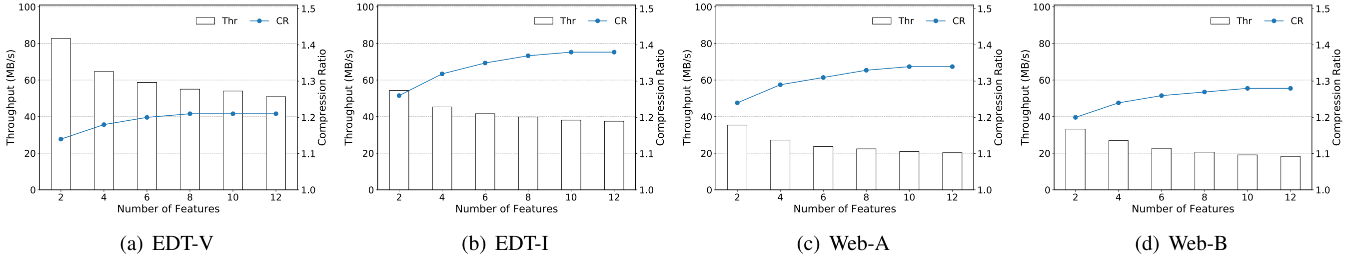
Comparison with Other Approaches. Here we compare FM-based approach with *byte-wise* and *2-d Rabin*. Note that *byte-wise* refers to using traditional byte-wise detection scheme with 64 bytes length one-dimensional window calculating Rabin hash [48] to generate features; *2-d Rabin* refers to using a two-dimensional window calculating Rabin hash in block unit (i.e., similar to Figure 7) to generate features.

Figure 14 illustrates the experimental results. Figure 14(a) suggests that all these approaches achieve a very close compression ratio on datasets, which demonstrates that they have similar efficiency on finding similar candidates. On the other hand, Figure 14(b) shows that *2-d Rabin* achieves a 1.4×–1.5× higher throughput, since it has a bigger processing unit and



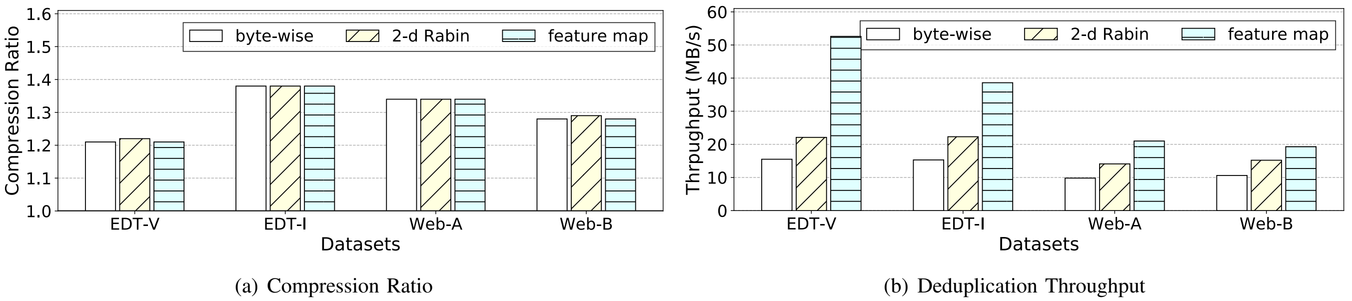
(a) EDT-V (b) EDT-I (c) Web-A (d) Web-B

Fig. 12. imDedup performances with different size of windows (Deduplication Throughput and Compression Ratio).



(a) EDT-V (b) EDT-I (c) Web-A (d) Web-B

Fig. 13. imDedup performances with different number of features (Deduplication Throughput and Compression Ratio).

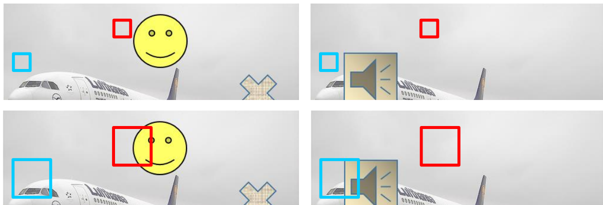


(a) Compression Ratio (b) Deduplication Throughput

Fig. 14. The comparison among different feature generation methods.

reduces computation overhead on generating features. Further, FM-based detector achieves a 1.8×–3.4× higher throughput, since it has a more efficient hash calculation approach and thus reduces computation overheads and memory accesses.

Summary. Experiments prove the efficiency of FM-based detector. It achieves 1.8×–3.4× higher throughputs and comparably high compression ratios.



(a) The bigger window may miss images with low similarity degrees.

(b) The smaller window may lead to detection misjudgment.

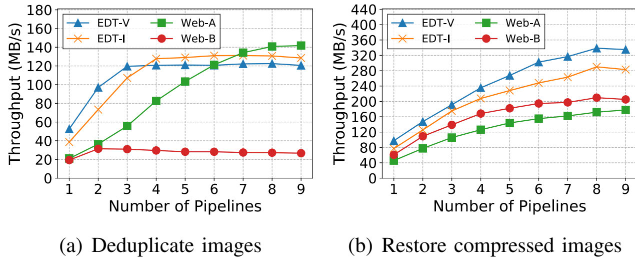
Fig. 15. Examples of how window size impacts on detecting similar images.

D. Comprehensive Experiment

In this subsection, we evaluate the concurrency of imDedup and compare imDedup with exact image deduplication.

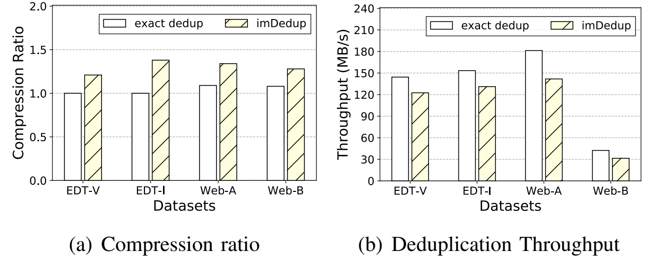
Concurrency. To speed up imDedup further, we apply concurrent pipelines to process a series of images at the same time. As shown in Figure 16, both when deduplicating and restoring (i.e., decompressing), throughput almost linearly increases with the growth of pipeline number before it meets the throughput peak, which suggests imDedup is salable in throughput. The bottleneck of imDedup's concurrency is I/O speed. The throughput no longer increases when it reaches the speed of reading raw/compressed images when deduplicating/restoring. Note that all pipelines' similarity detectors share a global feature index to maximize the compression ratio when deduplicating. And when restoring, the system doesn't write restored images into external storage since in most cases restored images will be used only in memory.

When deduplicating, image size has a significant impact on the I/O speed, and a smaller file size always leads to a worse I/O performance. Thus, datasets with smaller average sizes will meet the throughput peak sooner. The average image size of each dataset can be seen in Table II, and imDedup achieves the lowest deduplication throughput on Web-B since it has the smallest average image size. When restoring, the success rate of similarity detection has a significant impact on restore



(a) Deduplicate images (b) Restore compressed images

Fig. 16. The concurrency of imDedup.



(a) Compression ratio (b) Deduplication Throughput

Fig. 17. Comparison between exact image deduplication and imDedup.

throughput. A larger success rate means more images will be compressed, and then imDedup has more work to do in restore, which leads to a lower restore throughput. As shown in Figures 10(a) and 11(a), Web-A always has the largest success rate and therefore has the lowest restore throughput.

Figure 16(a) shows that during deduplication, imDedup achieves up to a 140 MB/s throughput and only takes 1.58 ms for a single image on average. Figure 16(b) shows that when restoring, imDedup achieves up to a 335 MB/s throughput and only takes 0.34 ms on average to restore a single image.

Comparison with Exact Deduplication. To comprehensively reveal the efficiency of our proposed imDedup, we implement a prototype of exact deduplication [16] for comparison. We choose exact deduplication as the representation of existing image deduplication instead of visual deduplication since we target compressing images in a lossless way.

The results in Figure 17 suggest that imDedup achieves a 19%–38% improvement in compression ratio with only a 14%–26% reduction in throughput. On the two simulated datasets, exact deduplication achieves a compression ratio of 1.0, which means there are no totally duplicate images. However, imDedup achieves compression ratios higher than 1.0 on them, which demonstrates imDedup successfully finds and eliminates fine-grained redundancy among images. The results that imDedup achieves higher compression ratios on two real-world datasets also demonstrate that there does be fine-grained redundancy among real-world images.

Summary. Experiments demonstrate that imDedup successfully detects and eliminates fine-grained redundancy between similar images while introducing more computation overheads for the system throughput. But the extra computation doesn't bother because imDedup is scalable in system throughput.

E. Discussion

In this subsection, we discuss the usage scenarios of imDedup according to its design and the above evaluation results.

Usage scenarios and limitations. On the one hand, the proposed imDedup is designed for image datasets containing lots of fine-grained redundancy, which can achieve a much higher compression ratio than classic schemes and thus save more storage space. On the other hand, the main limitation of imDedup is that it will take up excessive I/O and computation resources during deduplicating. Therefore, imDedup is more suitable for cold data in cloud storage to reduce the heavy storage cost of images.

Scalability in large-scale workload. For a large-scale image workload, a much higher deduplication throughput is required. So we can separate the whole workload into several parts and process them in a parallel way. For example, a large-scale cluster may have many nodes, and we can let each node run imDedup separately. This kind of implementation requires the separation of the workload promising most similar images are always allocated to the same parts or the same node (e.g., according to users or keywords). As another solution, we can introduce a distributed feature index for matching similar images and dispatch similar images to the same nodes (some related works on the distributed index include [55], [56]). This kind of implementation no longer requires separation of the workload but produces more heavy network traffic in the cluster to transmit images to the node of their similar ones.

VI. Conclusion and Future Work

imDedup is a lossless fine-grained SBD scheme for images. It detects and eliminates redundancy among images in a fine-grained and lossless way. imDedup fully utilizes image data's characteristics and is friendly to the original image format, which achieves both higher compression ratio and deduplication throughput. Experiments suggest that (i) there does be fine-grained redundancy among real-world images, which can not be eliminated by existing lossless image deduplication schemes, but imDedup can detect and eliminate fine-grained redundancy among images, bringing 19%–38% improvements in compression ratio; (ii) imDedup is more efficient in deduplicating image data than state-of-the-art SBD schemes, achieving 1.3×–1.6× higher compression ratios and 1.8×–3.4× higher throughputs, which suggests our proposed Idelta encoder and FM-based detector are more efficient.

In the future, we will verify the efficiency of the imDedup framework on other image formats (e.g., WebP, PNG, etc.). We also plan to explore the practicability of eliminating fine-grained redundancy among images in a lossy way.

Acknowledgment

This work was partly supported by NSFC (No. 61972441), Shenzhen Science and Technology Program (No. JCYJ20200109113427092 and GXWD20201230155427003-20200821172511002) and Guangdong Basic and Applied Basic Research Foundation (No. 2021A1515012634). Bo Tang was supported by the Guangdong Provincial Key Laboratory (No. 2020B121201001). Wen Xia (xiawen@hit.edu.cn) is the corresponding author.

REFERENCES

[1] H. Wang, X. Yi, P. Huang, B. Cheng, and K. Zhou, "Efficient ssd caching by avoiding unnecessary writes using machine learning," in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 1–10.

[2] R. F. Haines, *The effects of video compression on acceptability of images for monitoring life sciences experiments*. National Aeronautics and Space Administration, Office of Management, 1992, vol. 3239.

[3] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001, pp. 174–187.

[4] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage." in *FAST*, vol. 2, 2002, pp. 89–101.

[5] W. Xia, H. Jiang, D. Feng, and et al., "A comprehensive study of the past, present, and future of data deduplication," *Proc. of the IEEE*, vol. 104, no. 9, pp. 1681–1710, 2016.

[6] A. Wildani, E. L. Miller, and O. Rodeh, "Hands: A heuristically arranged non-backup in-line deduplication system," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 2013, pp. 446–457.

[7] L. L. You, K. T. Pollack, and D. D. Long, "Deep store: An archival storage system architecture," in *21st International Conference on Data Engineering (ICDE'05)*. IEEE, 2005, pp. 804–815.

[8] W. Xia, H. Jiang, D. Feng, and Y. Hua, "Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput." in *USENIX annual technical conference*, 2011, pp. 26–30.

[9] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system." in *Fast*, vol. 8, 2008, pp. 269–282.

[10] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality." in *Fast*, vol. 9, 2009, pp. 111–123.

[11] A. T. Clements, I. Ahmad, M. Vilayannur, J. Li *et al.*, "Decentralized deduplication in san cluster file systems." in *USENIX annual technical conference*, vol. 9, 2009, pp. 101–114.

[12] L. Xu, A. Pavlo, S. Sengupta, J. Li, and G. R. Ganger, in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015, pp. 222–235.

[13] P. Shilane, M. Huang, G. Wallace, and W. Hsu, "Wan-optimized replication of backup datasets using stream-informed delta compression," *ACM Transactions on Storage (ToS)*, vol. 8, no. 4, pp. 1–26, 2012.

[14] D. Bhagwat, K. Eshghi, D. D. Long, and M. Lillibridge, "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," in *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*. IEEE, 2009, pp. 1–9.

[15] J. MacDonald, "File system support for delta compression," 2000.

[16] Z. Lei, Z. Li, Y. Lei, Y. Bi, L. Hu, and W. Shen, "An improved image file storage method using data deduplication," in *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2014, pp. 638–643.

[17] J. Liu, N. Asokan, and B. Pinkas, "Secure deduplication of encrypted data without additional independent servers," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 874–885.

[18] F. Rashid, A. Miri, and I. Woungang, "Secure image deduplication through image compression," *Journal of Information Security and Applications*, vol. 27, pp. 54–64, 2016.

[19] M. Chen, S. Wang, and L. Tian, "A high-precision duplicate image deduplication approach." *J. Comput.*, vol. 8, no. 11, pp. 2768–2775, 2013.

[20] X. Li, J. Li, and F. Huang, "A secure cloud storage system supporting privacy-preserving fuzzy deduplication," *Soft Computing*, vol. 20, no. 4, pp. 1437–1448, 2016.

[21] M. Chen, Y. Wang, X. Zou, S. Wang, and G. Wu, "A duplicate image deduplication approach via haar wavelet technology," in *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*, vol. 2. IEEE, 2012, pp. 624–628.

[22] C. Zauner, M. Steinebach, and E. Hermann, "Rihamark: perceptual image hash benchmarking," in *Media watermarking, security, and forensics III*, vol. 7880. International Society for Optics and Photonics, 2011, p. 78800X.

[23] H. Xie, Y. Deng, H. Feng, and L. Si, "Pxdedup: Deduplicating massive visually identical jpeg image data," *Big Data Research*, vol. 23, p. 100171, 2021.

[24] G. K. Wallace, "The jpeg still picture compression standard," *IEEE transactions on consumer electronics*, vol. 38, no. 1, pp. xviii–xxxiv, 1992.

[25] "libjpeg-turbo," https://libjpeg-turbo.org/.

[26] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[27] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, "Tadoc: Text analytics directly on compression," *The VLDB Journal*, vol. 30, pp. 163–188, 2021.

[28] F. Zhang, J. Zhai, X. Shen, D. Wang, Z. Chen, O. Mutlu, W. Chen, and X. Du, "Poclib: A high-performance framework for enabling near orthogonal processing on compression," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 2, pp. 459–475, 2022.

[29] S. Al-Kiswany, D. Subhraveti, P. Sarkar, and M. Ripeanu, "Vmflock: Virtual machine co-migration for the cloud," in *Proceedings of the 20th international symposium on High performance distributed computing*, 2011, pp. 159–170.

[30] Z. Wen, J. Luo, H. Chen, J. Meng, X. Li, and J. Li, "A verifiable data deduplication scheme in cloud computing," in *2014 International Conference on Intelligent Networking and Collaborative Systems*. IEEE, 2014, pp. 85–90.

[31] H. Gang, H. Yan, and L. Xu, "Secure image deduplication in cloud storage," in *Information and Communication Technology-EurAsia Conference*. Springer, 2015, pp. 243–251.

[32] J. Takeshita, R. Karl, and T. Jung, "Secure single-server nearly-identical image deduplication," in *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2020, pp. 1–6.

[33] D. Trendafilov, N. Memon, and T. Suel, "zdelta: An efficient delta compression tool," 2002.

[34] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Y. Zhou, "Ddelta: A deduplication-inspired fast delta compression approach," *Performance Evaluation*, vol. 79, pp. 258–272, 2014.

[35] H. Tan, Z. Zhang, X. Zou, Q. Liao, and W. Xia, "Exploring the potential of fast delta encoding: Marching to a higher compression ratio," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2020, pp. 198–208.

[36] Y. Zhang, W. Xia, D. Feng, H. Jiang, Y. Hua, and Q. Wang, "Finesse: Fine-grained feature locality based fast resemblance detection for post-deduplication delta compression," in *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, 2019, pp. 121–128.

[37] N. Jain, M. Dahlin, and R. Tewari, "Taper: Tiered approach for eliminating redundancy in replica synchronization." in *FAST*, vol. 5, 2005, pp. 21–21.

[38] H. Pucha, D. G. Andersen, and M. Kaminsky, "Exploiting similarity for multi-source downloads using file handprints." in *NSDI*, 2007, p. 2007.

[39] L. Xu, A. Pavlo, S. Sengupta, and G. R. Ganger, "Online deduplication for databases," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1355–1368.

[40] F. Zhou, L. Zhuang, B. Y. Zhao, L. Huang, A. D. Joseph, and J. Kubiatowicz, "Approximate object location and spam filtering on peer-to-peer systems," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2003, pp. 1–20.

[41] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein, "The design of a similarity based deduplication system," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, 2009, pp. 1–14.

[42] G. Forman, K. Eshghi, and S. Chiocchetti, "Finding similar files in large document repositories," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, 2005, pp. 394–400.

[43] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: Harnessing memory redundancy in virtual machines," vol. 53, no. 10. ACM New York, NY, USA, 2010, pp. 85–93.

[44] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*. IEEE, 1997, pp. 21–29.

[45] P. Kulkarni, F. Douglis, J. D. LaVoie, and J. M. Tracey, "Redundancy elimination within large collections of files." in *USENIX Annual Technical Conference, General Track*, 2004, pp. 59–72.

[46] A. Z. Broder, "Identifying and filtering near-duplicate documents," in *Annual Symposium on Combinatorial Pattern Matching*. Springer, 2000, pp. 1–10.

[47] P. Jaccard, "Etude de la distribution florale dans une portion des alpes et du jura," *Bulletin de la Societe Vaudoise des Sciences Naturelles*, vol. 37, pp. 547–579, 1901.

[48] M. O. Rabin, "Fingerprinting by random polynomials," *Technical report*, 1981.

[49] Google, "Webp," /http://code.google.com/speed/webp/.

[50] Y. Collet, "New generation entropy codecs : Finite state entropy and huff0," https://github.com/Cyan4973/FiniteStateEntropy.

[51] V. Bhaskaran and K. Konstantinides, *Image and video compression standards: algorithms and architectures, Second Edition*. Springer Science & Business Media, 1997.

[52] L. V. Agostini, I. S. Silva, and S. Bampi, "Pipelined fast 2d dct architecture for jpeg image compression," in *Symposium on Integrated Circuits and Systems Design*. IEEE, 2001, pp. 226–231.

[53] M. Everingham, S. A. Eslami, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes challenge: A retrospective," *International journal of computer vision*, vol. 111, no. 1, pp. 98–136, 2015.

[54] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[55] P. Zuo, J. Sun, L. Yang, S. Zhang, and Y. Hua, "One-sided rdma-conscious extendible hashing for disaggregated memory," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 15–29.

[56] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers accelerating index traversals for in-memory databases," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 468–479.