Design Tradeoffs for Data Deduplication Performance in Backup Workloads

Min Fu†, Dan Feng†, Yu Hua†, Xubin He‡, Zuoning Chen*, Wen Xia†, Yucheng Zhang†, Yujuan Tan§

†Huazhong University of Science and Technology
‡Virginia Commonwealth University
*National Engineering Research Center for Parallel Computer
§Chongqing University

Feb. 19, 2015
In big data era,
  ▶ we had 4.4 ZB of data in 2013, and expectedly to grow by 10-fold in 2020 [IDC’2014];
  ▶ data redundancy widely exists in real-world applications.

Data deduplication is a scalable compression technology
  ▶ non-overlapping chunking
  ▶ no byte-by-byte comparison (fingerprinting)

A significantly lower computation overhead than traditional compression technologies
  ▶ faster due to coarse-grained compression
  ▶ higher compression ratio since it looks for duplicate chunks in a larger scope. (The entire system VS. a limited compression window)
An Overview of a Typical Deduplication System

(1) Lookup in the fingerprint cache

Hash engine

Chunk engine

Fingerprint Cache

(2) If yes

Find a match?

(3) If not

(7) Duplicate chunk
- Write the fingerprint to recipe store

Key-Value Store (fingerprint index)

Recipe Store

Container Store

(5) If yes

Find a match?

(6) If not

Unique chunk
- Write the chunk to container store
- Insert the fingerprint to key-value store
- Write the fingerprint to recipe store
Motivations

- Challenges to design an efficient deduplication system
  - Chunking
  - Indexing
  - Defragmenting
  - Restoring
  - Garbage collecting
  - ...

- We have a huge number of papers, solutions, and design choices
  - Which one is better?
  - Better in what?
    - backup performance, restore performance, deduplication ratio, or memory footprint?
  - How about their interplays?
1 The Parameter Space

2 The DeFrame Framework

3 Evaluation

4 Conclusions
The Parameter Space

- In this paper, we mainly explore
  - fingerprint indexing,
  - rewriting algorithm (in-line defragmenting),
  - restore algorithm (cache replacement),
  - and their interplays.

<table>
<thead>
<tr>
<th>Parameter Space</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Indexing</strong></td>
<td></td>
</tr>
<tr>
<td>Key-value mapping</td>
<td>Mapping fingerprints to their prefetching units</td>
</tr>
<tr>
<td>Fingerprint cache</td>
<td>In-memory fingerprint cache</td>
</tr>
<tr>
<td>Sampling</td>
<td>Selecting representative fingerprints</td>
</tr>
<tr>
<td>Segmenting</td>
<td>Splitting the unit of logical locality</td>
</tr>
<tr>
<td>Segment selection</td>
<td>Selecting segments to be prefetched</td>
</tr>
<tr>
<td>Segment prefetching</td>
<td>Exploiting segment-level locality</td>
</tr>
<tr>
<td><strong>Defragmenting</strong> (rewriting)</td>
<td>Reducing fragmentation</td>
</tr>
<tr>
<td><strong>Restoring</strong></td>
<td>Designing restore cache/algorithm</td>
</tr>
</tbody>
</table>

Table: The major parameters we discuss.
Indexing Bottleneck

- A deduplication system requires a huge key-value store to identify duplicates.
- An in-memory key-value store is not cost-efficient:
  - Amazon.com: a 1 TB HDD costs $60, and an 8 GB DRAM costs $80
  - Suppose 4KB-sized chunks and 32-byte-sized key-value pair
  - Indexing 1 TB unique data requires an 8 GB-sized key-value store.
- An HDD-based key-value store easily becomes the performance bottleneck, compared to the fast CDC chunking (400 MB/s and 102,400 chunks per sec under commercial CPUs).
- Modern fingerprint indexes exploit workload characteristics (locality) of backup systems to prefetch and cache fingerprints.
- Hence, a fingerprint index consists of two major components:
  - Key-value store
  - Fingerprint prefetching/caching module.
The Fingerprint Index Taxonomy

Classification according to the use of key-value store
- Exact Deduplication (ED): fully indexing stored fingerprints
- Near-exact Deduplication (ND): partially indexing stored fingerprints

Classification according to the prefetching policy
- Logical Locality (LL): the chunk sequence before deduplication
- Physical Locality (PL): the physical layout of stored chunks

Figure: Categories of existing work.
Exact vs. Near-exact Deduplication

- Exact Deduplication (ED) indexes all stored fingerprints
  - a huge key-value store on disks
  - fingerprint prefetching/caching to improve backup throughput

- Near-exact Deduplication (ND) indexes only sampled (representative) fingerprints
  - a small key-value store in DRAM
  - fingerprint prefetching/caching to improve deduplication ratio

- ND trades deduplication ratio for higher backup/restore performance and lower memory footprint
  - Does a lower memory footprint indicate a lower financial cost?
  - To avoid an increase of the storage cost, ND needs to achieve 97% of the deduplication ratio of ED.
Exploiting Physical Locality

- The key-value store maps a fingerprint to its physical location, i.e., a container.
- Weakness: the prefetching efficiency decreases over time due to the fragmentation problem.
  - Older containers have many useless fingerprints for new backups.
- For Near-exact Deduplication, how to select (sample) representative fingerprints in each container?
  - Selects the fingerprints that $\mod R = 0$ in a container, or
  - Selects the first fingerprint every $R$ fingerprints in a container.
Exploiting Logical Locality

The key-value store maps a fingerprint to its logical location, i.e., a segment in a recipe.

The segment serves as the prefetching unit

Advantage: no fragmentation problem

Weakness: extremely high update overhead to the key-value store
  ▶ Even duplicate fingerprints have new logical locations (in new recipes and new segments).
  ▶ Optimization: only update sampled duplicate fingerprints.

How to segmenting and sampling?
Design Choices for Exploiting Logical Locality

<table>
<thead>
<tr>
<th></th>
<th>BLC</th>
<th>Extreme Binning</th>
<th>Sparse Indexing</th>
<th>SiLo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact deduplication</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Segmenting method</td>
<td>FSS</td>
<td>FDS</td>
<td>CDS</td>
<td>FSS &amp; FDS</td>
</tr>
<tr>
<td>Sampling method</td>
<td>N/A</td>
<td>Minimum</td>
<td>Random</td>
<td>Minimum</td>
</tr>
<tr>
<td>Segment selection</td>
<td>Base</td>
<td>Top-all</td>
<td>Top-k</td>
<td>Top-1</td>
</tr>
<tr>
<td>Segment prefetching</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Key-value mapping relationship</td>
<td>1:1</td>
<td>1:1</td>
<td>Varied</td>
<td>1:1</td>
</tr>
</tbody>
</table>

Table: Existing work exploiting logical locality.

- **Segmenting:** Fixed-Sized Segmenting (FSS), File-Defined Segmenting (FDS), and Content-Defined Segmenting (CDS)
- **Sampling:** Uniform, Random, and Minimum.
- **Segment selection:** Base, Top-k, and Mix.
- **Segment prefetching:** exploiting segment-level locality.
- **Key-value mapping:** each representative fingerprint can refer to a varied number of logical locations.
The rewriting algorithm is an emerging dimension to reduce fragmentation.

What is the fragmentation?
- The deviation between the logical locality and physical locality.

The fragmentation hurts the restore (read) performance, and the backup performance of the fingerprint index exploiting physical locality.
Existing Rewriting Algorithms

- Buffer-based algorithm
  - CFL-based Selective Deduplication [Nam’2012]
  - Context-Based Rewriting [Kaczmarczyk’2012]
  - Capping [Lillibrige’2013]

- History-aware algorithm
  - History-Aware Rewriting [Fu’2014]

How about their interplays with the state-of-the-art fingerprint indexes?
  - How does the rewriting algorithm improve the fingerprint index exploiting physical locality?
  - How do the different prefetching schemes affect the efficiency of the rewriting algorithm?
The Restore Algorithm

- While the rewriting algorithm determines the chunk layout, the restore algorithm improves restore performance under limited memory.
  - How to write, and then how to read.
- Existing restore algorithms:
  - traditional LRU cache
  - Belady’s optimal replacement cache
  - rolling forward assembly area [Lillibridge’2013]
- How about their interplays with the rewriting algorithm?
The DeFrame Architecture

- **Fingerprint index**: duplicate identification
  - key-value store
  - fingerprint prefetching/caching
- **Container store**: container management (physical locality)
- **Recipe store**: recipe management (logical locality)
**The Backup Pipeline**

- **Dedup phase** identifies duplicate/unique chunks
- **Rewrite phase** identifies fragmented duplicate chunks
- **Filter phase** determines whether write a chunk
- Advantage: we can implement a new rewriting algorithm without the need to modify the fingerprint index, and vice versa.
The Restore Pipeline

- **Read recipe phase** reads the recipe and output fingerprints
- **Restore algorithm phase** receives fingerprints and fetch chunks from the container store
- **Reconstruct file phase** receives the chunks and reconstruct files
Garbage Collection

- Users can set a retention time for the backups.
- All expired backups will be deleted automatically by DeFrame.
- How to reclaim the invalid chunks becomes a major challenge.
  - We develop History-Aware Rewriting algorithm to aggregate valid chunks into fewer containers
  - We develop Container-Marker Algorithm to reclaim invalid containers.
- More details can be found in our ATC’14 paper.
Experimental Setups

<table>
<thead>
<tr>
<th>Dataset name</th>
<th>Kernel</th>
<th>VMDK</th>
<th>RDB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total size</strong></td>
<td>104 GB</td>
<td>1.89 TB</td>
<td>1.12 TB</td>
</tr>
<tr>
<td><strong># of versions</strong></td>
<td>258</td>
<td>127</td>
<td>212</td>
</tr>
<tr>
<td><strong>Deduplication ratio</strong></td>
<td>45.28</td>
<td>27.36</td>
<td>39.1</td>
</tr>
<tr>
<td><strong>Avg. chunk size</strong></td>
<td>5.29 KB</td>
<td>5.25 KB</td>
<td>4.5 KB</td>
</tr>
<tr>
<td><strong>Self-reference</strong></td>
<td>&lt; 1%</td>
<td>15-20%</td>
<td>0</td>
</tr>
<tr>
<td><strong>Fragmentation</strong></td>
<td>Severe</td>
<td>Moderate</td>
<td>Severe</td>
</tr>
</tbody>
</table>

**Table:** The characteristics of our datasets.

- **Kernel:** downloaded from kernel.org
- **VMDK:** 127 consecutive snapshots of a virtual machine disk image
- **RDB:** 212 consecutive snapshots of a Redis database
## Metrics and Our Goal

### Quantitative Metrics

- **Deduplication ratio**: the original backup data size divided by the size of stored data.
- **Memory footprint**: the runtime DRAM consumption.
- **Storage cost**: the total cost of HDDs and DRAM for stored chunks and the fingerprint index.
- **Lookup/update request per GB**: the number of lookup/update requests to the key-value store to deduplicate 1 GB of data.
- **Restore speed**: 1 divided by mean containers read per MB of restored data.

- It is practically impossible to find a solution that performs the best in all metrics.
- We aim to find a solution with the following properties:
  - sustained, high backup performance as the top priority.
  - reasonable tradeoffs in the remaining metrics.
EDPL vs. EDLL

**Figure:** Comparisons between EDPL and EDLL in terms of lookup and update overheads. $R = 256$ indicates a sampling ratio of 256:1. Results come from RDB.

- EDPL suffers from the ever-increasing lookup overhead.
- For EDLL, the sampling optimization is efficient.
NDPL vs. NDLL

Figure: Comparing NDPL and NDLL under different cache sizes. The Y-axis shows the relative deduplication ratio to exact deduplication.

- NDLL performs better in Kernel and RDB, but worse in VMDK than NDPL.
The Interplays Between Fingerprint Index and Rewriting Algorithm

Figure: (a) How does HAR improve EDPL in terms of lookup overhead in Kernel? (b) How does fingerprint index affect HAR? The Y-axis shows the relative deduplication ratio to that of exact deduplication without rewriting.

- Exact Deduplication exploiting Physical Locality (EDPL) has the best interplays with the rewriting algorithm (HAR).
The Interplays Between Rewriting and Restore Algorithm

![Graph showing restore speed vs backup version for different cache algorithms.](image)

**Figure:** EDPL is used as the fingerprint index

- When HAR is used, the optimal cache is better; otherwise, the rolling forward assembly area is better.
Conclusions

- We propose a taxonomy to understand the parameter space of data deduplication.
- We design and implement a framework to evaluate the parameter space.
- We present our experimental results, and draw the following recommendations.

<table>
<thead>
<tr>
<th>Subspace</th>
<th>Recommended parameter settings</th>
<th>Advantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDLL</td>
<td>content-defined segmenting</td>
<td>lowest storage cost</td>
</tr>
<tr>
<td></td>
<td>random sampling</td>
<td>sustained backup performance</td>
</tr>
<tr>
<td>NDPL</td>
<td>uniform sampling</td>
<td>lowest memory footprint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>simplest logical frame</td>
</tr>
<tr>
<td>NDLL</td>
<td>content-defined segmenting</td>
<td>lowest memory footprint</td>
</tr>
<tr>
<td></td>
<td>similarity detection &amp; segment</td>
<td>high deduplication ratio</td>
</tr>
<tr>
<td></td>
<td>prefetching</td>
<td></td>
</tr>
<tr>
<td>EDPL</td>
<td>an efficient rewriting algorithm</td>
<td>sustained high restore performance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>good interplays with the rewriting algorithm</td>
</tr>
</tbody>
</table>

**Table:** How to choose a reasonable solution according to required tradeoff.
Thank You!

Q & A

DeFrame is released at www.github.com/fomy/destor
Exploiting Similarity for NDLL

Figure: This figure shows the workflow of the Top-k similarity detection.

- **Observations**: NDLL works better than NDPL in datasets where self-references are rare, but worse in datasets where self-references are common.
  - Self-references are duplicates in a single file (backup).
- We could exploit similarity to improve deduplication ratio.
- **Advantage**: higher deduplication ratio than the Base procedure.
- **Weakness**: more complicated procedure and an additional buffer, compared to the Base procedure.
The efficiency of similarity detection

Figure: Impacts of the segment selection ($s$), segment prefetching ($p$), and mapping relationship ($v$) on deduplication ratio.

- On the X-axis, we have parameters in the format $(s, p, v)$.
  - $s$ could be Base and Top-$k$ ($k$ varies from 1 to 4).
  - $p$ varies from 1 to 4.
  - $v$ varies from 1 to 4.
- They finally achieve 90% of the deduplication ratio of exact deduplication.
The efficiency of similarity detection

**Figure:** Impacts of the segment selection (s), segment prefetching (p), and mapping relationship (v) on segments read.

- The segment prefetching is complementary with Top-k.
### Storage cost

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Fraction</th>
<th>EDPL/EDLL</th>
<th>NDPL-64</th>
<th>NDPL-128</th>
<th>NDPL-256</th>
<th>NDLL-64</th>
<th>NDLL-128</th>
<th>NDLL-256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>DRAM</td>
<td>1.33%</td>
<td>0.83%</td>
<td>0.49%</td>
<td>0.31%</td>
<td>0.66%</td>
<td>0.34%</td>
<td>0.16%</td>
</tr>
<tr>
<td></td>
<td>HDD</td>
<td>57.34%</td>
<td>65.01%</td>
<td>70.56%</td>
<td>77.58%</td>
<td>59.03%</td>
<td>59.83%</td>
<td>60.23%</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>58.67%</td>
<td>65.84%</td>
<td>71.04%</td>
<td>77.89%</td>
<td>59.69%</td>
<td>60.17%</td>
<td>60.39%</td>
</tr>
<tr>
<td>RDB</td>
<td>DRAM</td>
<td>1.40%</td>
<td>0.83%</td>
<td>0.48%</td>
<td>0.31%</td>
<td>0.70%</td>
<td>0.35%</td>
<td>0.17%</td>
</tr>
<tr>
<td></td>
<td>HDD</td>
<td>55.15%</td>
<td>61.25%</td>
<td>66.08%</td>
<td>73.58%</td>
<td>55.27%</td>
<td>55.34%</td>
<td>55.65%</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>56.55%</td>
<td>62.07%</td>
<td>66.56%</td>
<td>73.89%</td>
<td>55.97%</td>
<td>55.69%</td>
<td>55.82%</td>
</tr>
<tr>
<td>VMDK</td>
<td>DRAM</td>
<td>1.41%</td>
<td>0.82%</td>
<td>0.45%</td>
<td>0.27%</td>
<td>0.71%</td>
<td>0.35%</td>
<td>0.18%</td>
</tr>
<tr>
<td></td>
<td>HDD</td>
<td>54.86%</td>
<td>60.32%</td>
<td>63.16%</td>
<td>67.10%</td>
<td>59.79%</td>
<td>62.92%</td>
<td>71.24%</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>56.27%</td>
<td>61.14%</td>
<td>63.61%</td>
<td>67.36%</td>
<td>60.49%</td>
<td>63.27%</td>
<td>71.42%</td>
</tr>
</tbody>
</table>

**Table:** The storage costs relative to the baseline which indexes all fingerprints in DRAM. NDPL-128 is NDPL of a 128:1 uniform sampling ratio.

- Via exploiting locality, the storage cost reduces by about 40%.
- Near-exact deduplication reduces the memory footprint, however it generally increases the total storage cost.
Choosing Sampling Method in NDPL

Figure: Impacts of varying sampling method on NDPL. Points in each line are of different sampling ratios, which are 256, 128, 64, 32, 16, and 1 from left to right.

- The uniform sampling achieves significantly better deduplication ratio than the random sampling.