Accelerating Restore and Garbage Collection in Deduplication-based Backup Systems via Exploiting Historical Information

Min Fu[†], Dan Feng[†], Yu Hua[†], Xubin He[‡], Zuoning Chen[*], Wen Xia[†], Fangting Huang[†], Qing Liu[†]

[†]Huazhong University of Science and Technology
[‡]Virginia Commonwealth University,
[*]National Engineering Center for Parallel Computer

June 19, 2014

Background

What's data deduplication?

Data deduplication is a scalable compression technique used in large-scale backup systems.

Traditional compression compresses a piece of data (e.g., a small file) at byte granularity.

Data deduplication compresses the entire storage system at chunk granularity.

Background

What's data deduplication?

Data deduplication is a scalable compression technique used in large-scale backup systems.

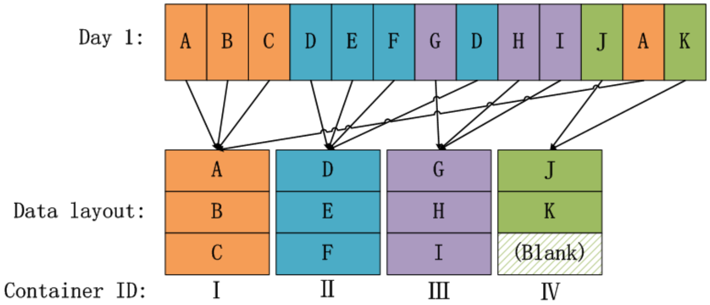
Traditional compression compresses a piece of data (e.g., a small file) at byte granularity.

Data deduplication compresses the entire storage system at chunk granularity.

The fragmentation problem caused by data deduplication:

1. **Slow restore** (a 21X decrease!)
   — *data we need is dispersed physically.*
2. **Slow and cumbersome garbage collection**
   — *data we do NOT need is dispersed physically.*

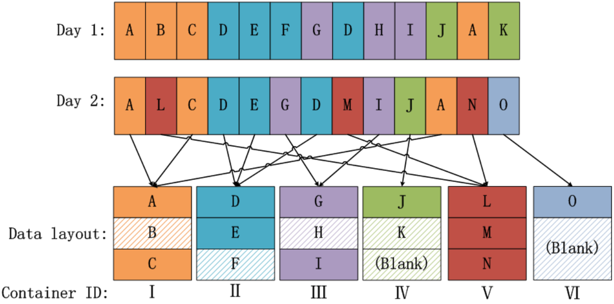
How the fragmentation arises



The first backup:

- We have 13 chunks, most of which are **UNIQUE**.
- Restoring this backup with 3-container-sized LRU cache requires **5** container reads.
- Restoring this backup with an unlimited cache requires **4** container reads.

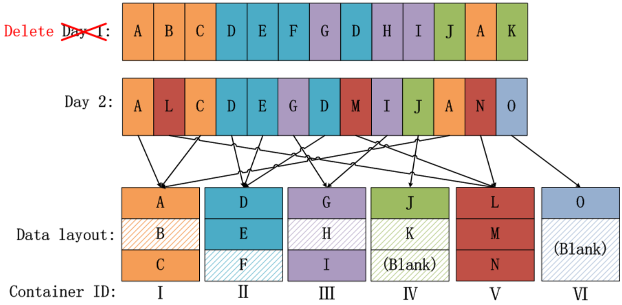
How the fragmentation arises



The second backup:

- We also have 13 chunks, 9 of which are **DUPLICATE**.
- Restoring this backup with 3-container-sized LRU cache requires **9** container reads.
- Restoring this backup with an unlimited cache requires **6** container reads.

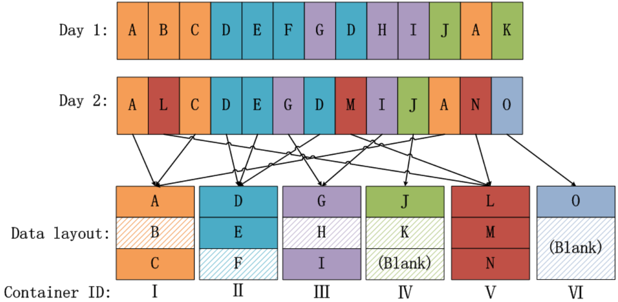
How the fragmentation arises



If we delete the first backup:

- **4** chunks (**B, F, H, and K**) become invalid.
- We can **NOT** reclaim their space without additional mechanisms.
- **Container merge** operation: migrate valid chunks into new containers. The most time-consuming phase in garbage collection.

Our observations and motivations



The fragmentation taxonomy

Sparse container: a container with a **utilization** smaller than **utilization threshold** (e.g., 50%), such as Container IV for backup 2.

Out-of-order container: its chunks are intermittently referenced by a backup, such as Container V for backup 2.

Our observations and motivations

The negative impacts and potential solutions:

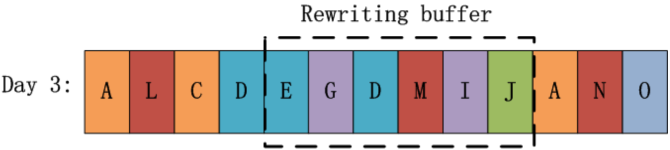
Sparse containers directly amplify read operations, hence hurt both restore and garbage collection.

- *Solution: rewriting referenced chunks in them to new compact containers, i.e., rewriting algorithm.*

Out-of-order containers hurt restore if the restore cache is small.

- *Solution: Increasing the cache size, or developing more intelligent replacement algorithm than LRU.*

Our observations and motivations

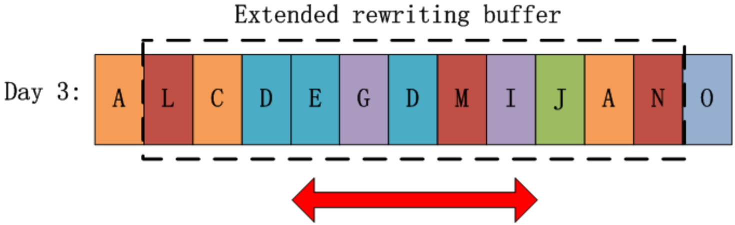


How existing rewriting algorithms work?

Deduplication is delayed to identify fragmented duplicate chunks.

- They use a **rewriting buffer** and identify duplicate but fragmented chunks in the buffer.
  - The chunk M is supposed to be in a sparse container, since it has no physical neighbor in the buffer.

Our observations and motivations



Existing rewriting algorithms:

- If we extend the rewriting buffer, more physical neighbors of *M* would be found. *M is in an out-of-order container rather than a sparse container!*
  - *NOT scalable since memory is limited.*

Our observations and motivations

The problems of existing rewriting algorithms:

They **CANNOT** accurately differentiate sparse containers from out-of-order containers due to the limited size of the rewriting buffer. As a result, they

- lose too much storage efficiency, and
- gain limited restore speed.

Our observations and motivations

The problems of existing rewriting algorithms:

They **CANNOT** accurately differentiate sparse containers from out-of-order containers due to the limited size of the rewriting buffer. As a result, they

- lose too much storage efficiency, and
- gain limited restore speed.

The challenge:

Due to the existence of out-of-order containers, accurately identifying sparse containers requires the complete knowledge of the on-going backup.

- How can we obtain such knowledge on the fly?

Our observations and motivations

Rationale

Due to the incremental nature of backup, consecutive backups share similar characteristics, including fragmentation.

Our key observations:

1. The number of total sparse containers continuously grows.
2. The number of total sparse containers increases smoothly.
   - Only a limited number of emerging sparse containers in each backup.
3. A backup inherits most of the sparse containers of last backup.

Design and implementation

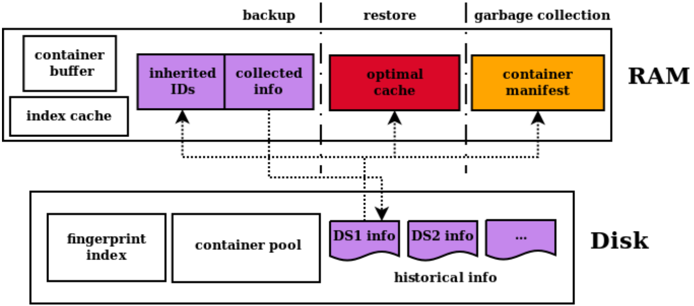


Figure : The system architecture. Colored modules are our contributions.

Three contributions:

- History-Aware Rewriting Algorithm (HAR) in backup (tackling sparse containers);
- Belady's optimal replacement algorithm (OPT) in restore (tackling out-of-order containers);
- Container-Marker Algorithm (CMA) in garbage collection (a new reference management).

Design and implementation

History-Aware Rewriting

History-Aware Rewriting (HAR)

HAR records sparse containers during the backup, and rewrite referenced chunks in them during next backup.

- The emerging sparse containers of a backup become the inherited sparse containers of the next backup.

Design and implementation

History-Aware Rewriting

History-Aware Rewriting (HAR)

HAR records sparse containers during the backup, and rewrite referenced chunks in them during next backup.

- The emerging sparse containers of a backup become the inherited sparse containers of the next backup.

Advantages:

- NOT rewriting emerging sparse containers does NOT hurt restore performance due to the limited number of emerging sparse containers (observation 2);
- Rewriting inherited sparse containers does NOT hurt backup performance due to the limited number of inherited sparse containers (observation 2);
- Identify sparse containers accurately due to Observation 3.

Design and implementation

Optimal cache

The problem of out-of-order containers

Out-of-order containers hurt restore performance if the restore cache is small.

Design and implementation

Optimal cache

The problem of out-of-order containers

Out-of-order containers hurt restore performance if the restore cache is small.

Our observations:

We restore a backup stream according to the fingerprint sequence preserved in the recipe. As a result,

- We exactly know the future access pattern of containers during the restore.
  - more intelligent cache replacement algorithms than LRU are possible.

Design and implementation

Optimal cache

The problem of out-of-order containers

Out-of-order containers hurt restore performance if the restore cache is small.

Our observations:

We restore a backup stream according to the fingerprint sequence preserved in the recipe. As a result,

- We exactly know the future access pattern of containers during the restore.
  - more intelligent cache replacement algorithms than LRU are possible.

Belady's optimal replacement algorithm:

When the restore cache is full, the container that will not be accessed for the longest time in the future is evicted.

Design and implementation

Container-Marker Algorithm

Two-phased garbage collection:

1. **Chunk reference management**: find invalid chunks, and calculate the utilizations of containers to identify which containers are worth being merged (i.e., sparse containers).
   - *its overhead is proportional to the number of chunks.*
2. **Container merge**: migrate valid chunks in sparse containers to new containers.
   - *it competes with regular backup and urgent restore for I/O bandwidth.*

Design and implementation

Container-Marker Algorithm

Two-phased garbage collection:

1. **Chunk reference management**: find invalid chunks, and calculate the utilizations of containers to identify which containers are worth being merged (i.e., sparse containers).
   - *its overhead is proportional to the number of chunks.*
2. **Container merge**: migrate valid chunks in sparse containers to new containers.
   - *it competes with regular backup and urgent restore for I/O bandwidth.*

Our observation:

After the latest backup referring to the sparse container is deleted, we can directly reclaim the container rather than merging it.

Simplified reference management is possible!

Container-Marker Algorithm (CMA)

- Maintains a container manifest for each dataset.
  - The manifest records IDs of all containers related to the dataset.
  - Each container ID is paired with a backup time that indicates the most recent backup referring to the container.
- Suppose we delete all backups before time T.
  - All containers with a backup time smaller than T can be reclaimed.
- The overhead is proportional to the number of containers rather than chunks.

Evaluation

Evaluation Methodology

We implement the *baseline* (no rewriting) and two existing rewriting algorithms (CBR @ SYSTOR'12 and CAP @ FAST'13) for comparisons.

- Deduplication ratio: the size of the non-deduplicated data divided by that of the deduplicated data.
- Speed factor (@ FAST'13): a metric to measure restore performance. It's defined as the size of restored data (MB) per container read.
- The number of valid containers (the actual storage cost after GC).

Evaluation

Table : Characteristics of datasets.

dataset name	VMDK	Linux	Synthetic
total size	1.44TB	104GB	4.5TB
# of versions	102	258	400
deduplication ratio	25.44	45.24	37.26
avg. chunk size	10.33KB	5.29KB	12.44KB
sparse	medium	severe	severe
out-of-order	severe	medium	medium

Evaluation

Table : Default settings.

fingerprint index	in-memory
container size	4MB
utilization threshold	50%
caching scheme	OPT
backup retention time	20 days
container merge	N/A

Evaluation

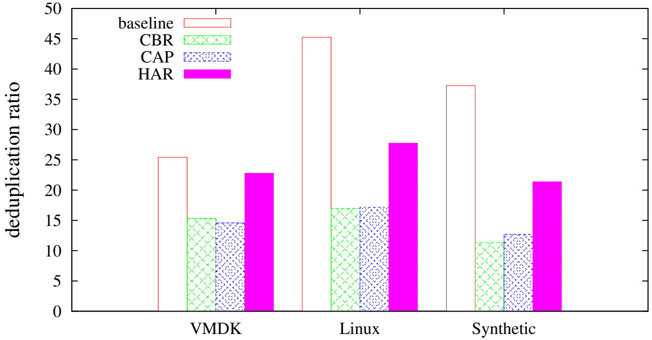


Figure : The comparisons between HAR and other rewriting algorithms in terms of deduplication ratio.

Conclusion (1)

HAR rewrites less data than CBR and CAP.

Evaluation

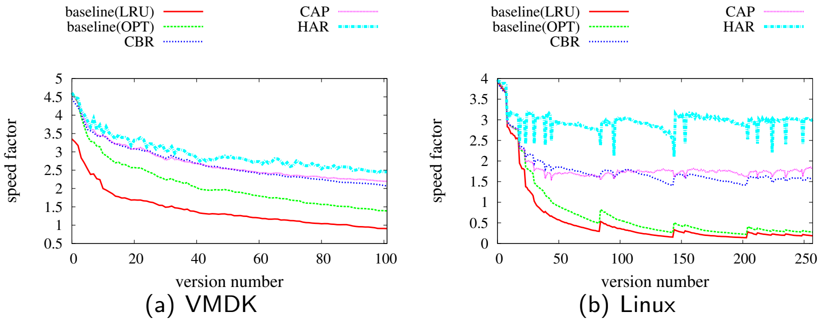


Figure : The comparisons of rewriting algorithms in terms of restore performance. The cache is 512- and 32-container-sized in VMDK and Linux respectively.

Conclusion (2)

HAR achieves better restore performane, while rewrites less data than CBR and CAP.

Evaluation

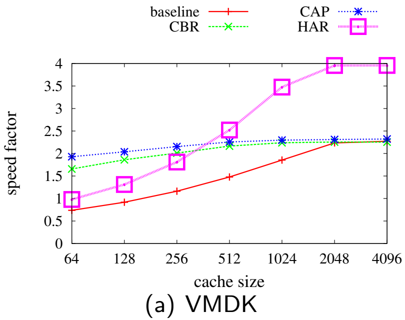
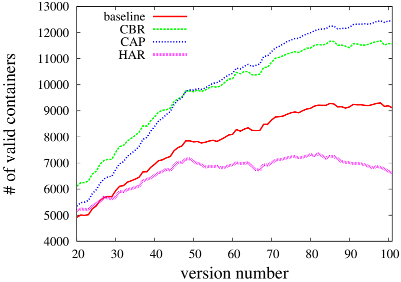


Figure : The comparisons of rewriting algorithms under various cache size. Speed factor is the average value of last 20 backups. The cache size is in terms of # of containers.

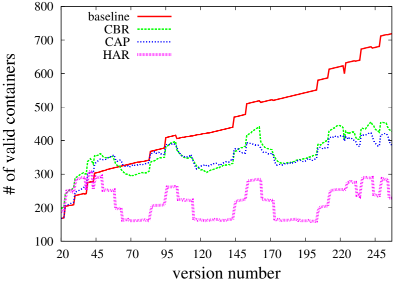
Conclusion (3)

HAR works significantly better when the restore cache is large.

Evaluation



(a) VMDK



(b) Linux

Figure : The comparisons of rewriting algorithms in terms of the storage cost after garbage collection.

Conclusion (4)

After GC, HAR has lowest storage cost since it reduces sparse containers.

More in the paper

- A hybrid rewriting scheme:
  - HAR+CBR;
  - HAR+CAP.
- More experimental results:
  - For Synthetic dataset.
  - Metadata overhead of garbage collection.
  - Varying the utilization threshold in HAR.
- Related work.

Summary

- The fragmentation taxonomy:

  Sparse containers hurt both restore and garbage collection.

  Out-of-order containers hurt restore if the cache is small.
- History-Aware Rewriting: rewrites less data but gains more restore speed than existing work.
  - *Solve the sparse container problem.*
- Optimal cache: reduces the cache size we require.
  - *Alleviate the out-of-order container problem.*
- Container-Marker Algorithm: simpler and lower metadata overhead.
  - *A new reference management.*