Fragmentation in in-line deduplication backup systems


2. Improving Restore Speed for Backup Systems that Use Inline Chunk-Based Deduplication, Mark Lillibridge and Kave Eshghi and Deepavali Bhagwat. FAST 2013


Speaker: Oren Kishon, Advanced Topics in Storage Systems, Graduate Seminar
Talk outline

- The problem
- The new ideas for solutions
- Experimental results
- Summary / discussion / questions
Talk outline

- The problem
- The new ideas for solutions
- Experimental results
- Summary / discussion / questions
The problem: Fragmentation

In-line deduplication: handle blocks as they come, without post-processing:

- If block already exist: only update dedup data
- If new: “throw” at the end of the current data set.
- At restore time, the data is already very fragmented, and so the restore is slow.
The problem: Fragmentation

Where chunk fragmentation comes from

Day 1: A B C D E F D G

Store:
The problem: Fragmentation
The problem: Fragmentation
The problem: Fragmentation

Where chunk fragmentation comes from

Day 1: A B C D E F D G

Day 2: A C D E H F D G I

Store: A B C D E F G
The problem: Fragmentation

Where chunk fragmentation comes from

Day 1: ABCDEFDG

Day 2: ACDEHFDGI

Store: ABCDEFGHI
The problem: Fragmentation

Where chunk fragmentation comes from

Day 1:    A B C D E F D G

Day 2:    A C D E H F D G I

Day 3:    J A C D H K F D G I L

Store:    A B C D E F G H I
The problem: Fragmentation

Where chunk fragmentation comes from

Day 1:  A B C D E F D G

Day 2:  A C D E H F D G I

Day 3:  J A C D H K F D G I L

Store:  A B C D E F G H I J K L
The problem: Fragmentation
The problem: Fragmentation
Talk outline

- The problem
- The new ideas for solutions
- Experimental results
- Summary / discussion / questions
The two new ideas

- Trade off deduplication for lower fragmentation.
  In other words: write more, but read faster.

- Caching: smarter algorithm than LRU.
The two new ideas

- Trade off deduplication for lower fragmentation.
  In other words: write more, but read faster.

- Caching: smarter algorithm than LRU.
Trade off deduplication

- 9LivesData: “Context based Rewriting” (CBR)
- HP: “Capping”
Trade off deduplication

- 9LivesData: “Context based Rewriting” (CBR)
- HP: “Capping”
Context base rewriting (CBR)

Each dup:
- Stream context (5MB)
- Disk context (2MB)

Figure 4. Disk and stream contexts of a block
Context base rewriting (CBR)

- Large intersection: no need to rewrite.
- Small intersection: Rewriting will speed-up restore time.
Context base rewriting (CBR)

Reaching a decision

- Metric used - “Rewrite Utility” (for context): disk blocks not in stream / total disk blocks

- Two thresholds:
  1. Global: rewrite-utility > 70%
  2. Adjusting: rewrite-utility > 95% of other scores so far.

- 5% limit: if 5% actually rewritten, stop rewriting.
Trade off deduplication

- 9LivesData: “Context based Rewriting” (CBR)
- HP: “Capping”
Capping

- Input: Break into 20 MB segments.

- Disk: Read/seek chunks. Chunk container: 4MB.

- Metric used: “Fragmentation” = \#containers read (at restore) / MB actually restored

- Deduplicate each segment against at most T containers. At cost of not deduplicating some chunks...

- Fragmentation is limited to (T+5)/20 MB
Capping

Capping illustrated

Segment:

Store:

T = 3

Container 5: 5
Container 7: 3
Container 12: 3
Container 9: 1
Container 10: 1
Capping illustrated

Segment:

Store:

T = 3
Container 5: 5
Container 7: 3
Container 12: 3

Result: 6 containers -> 4 containers used
2 extra chunk copies
## Trade off deduplication

The major difference

<table>
<thead>
<tr>
<th>CBR</th>
<th>Capping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Focus on limiting:</td>
<td>Rewriting (lose of deduplication)</td>
</tr>
<tr>
<td>Don't limit:</td>
<td>Fragmentation</td>
</tr>
</tbody>
</table>
The two new ideas

- Trade off deduplication for lower fragmentation.
  In other words: write more, but read faster.

- Caching: smarter algorithm than LRU.
Better caching: Forward assembly area

An insight: LRU and other cache algorithms solve a problem of non-deterministic requests, while deduplicated data restoration is totally deterministic.
Better caching

The forward assembly area method

Store:

RAM:

Container buffer (4 MB)  Forward assembly area (100 MB)

Recipe buffer (~2 MB)
Better caching

The forward assembly area method

Store:

[Diagram showing store with sections]

RAM:

[Diagram showing RAM with sections]

- Container buffer (4 MB)
- Forward assembly area (100 MB)
- Recipe buffer (~2 MB)
Better caching
Better caching
Better caching
Better caching
Better caching

**The forward assembly area method**

- **Store:**

- **RAM:**
Better caching

The forward assembly area method

Store:

RAM:
Better caching

The forward assembly area method

Store:

RAM:
Better caching

The forward assembly area method

Store:

RAM:
Better caching:
Forward assembly area

Wrap up:
- Knowledge of the future helps us decide what to keep / throw from cache (no need to make assumptions).
- Caching in sizes of chunks (rather than containers) saves space and later copying.
Talk outline

- The problem
- The new ideas for solutions
- Experimental results
- Summary / discussion / questions
Experimental results

- CBR: up to X2 restore time, 1-5% dedup loss
- Capping: 2-6X, 8% dedup loss
- Forward assembly area: 2-4X
Experimental results

- CBR: up to X2 restore time, 1-5% dedup loss
- Capping: 2-6X, 8% dedup loss
- Forward assembly area: 2-4X

But:
They had different metrics of “restore speed”
And:
CBR was tested on some sets of 14 updates.
Capping/Assembly on sets of hundreds...
Talk outline

- The problem
- The new ideas for solutions
- Experimental results
- Summary / discussion / questions
In-line deduplication has a downfall of fragmentation causing slow restore speeds.

The restore speed can be greatly increased by losing only little deduplication, and smarter caching.

Questions? Comments?