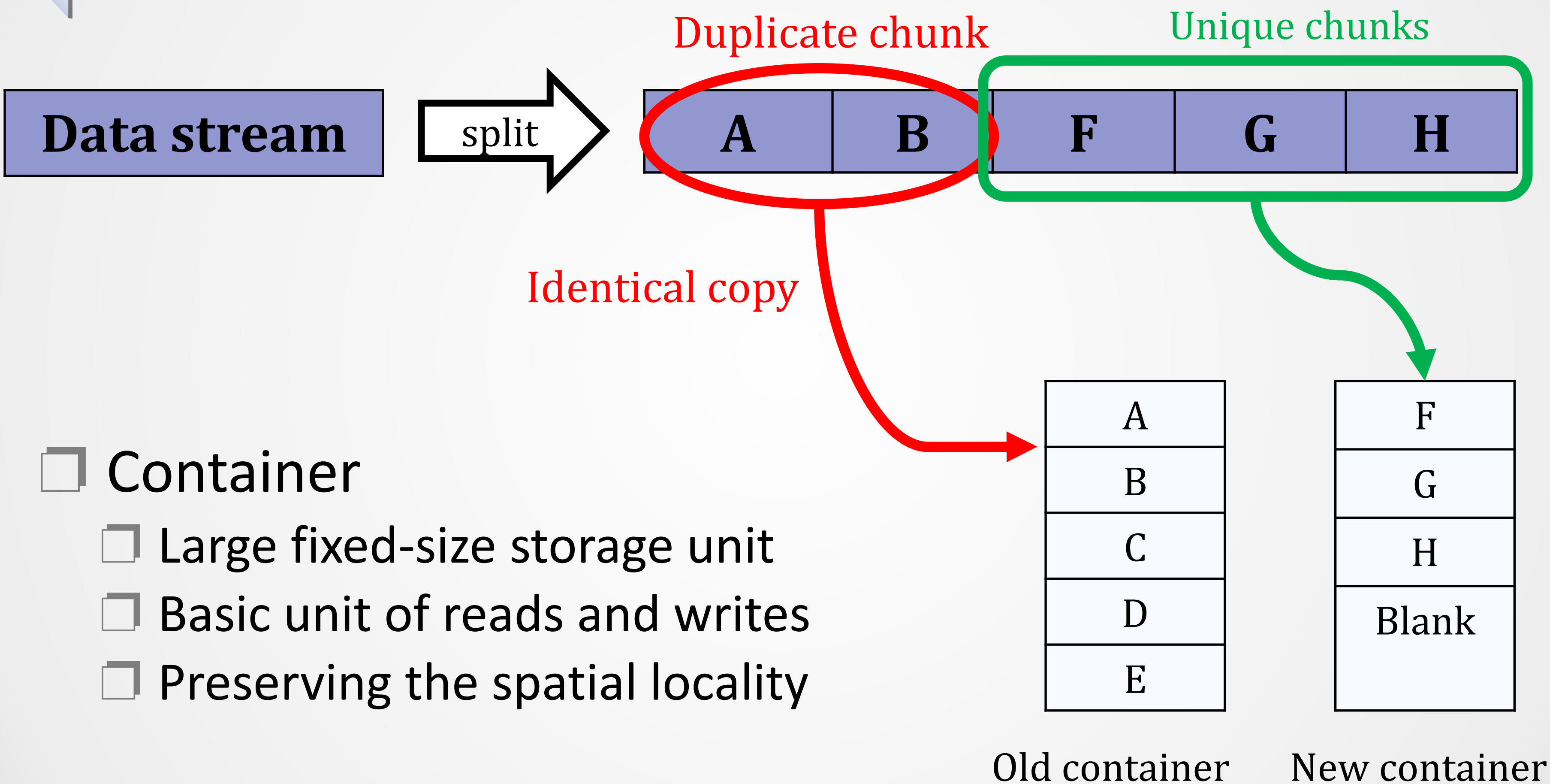
A Cost-efficient Rewriting Scheme to Improve Restore Performance in Deduplication Systems

Jie Wu, Yu Hua, **Pengfei Zuo**, Yuanyuan Sun

Huazhong University of Science and Technology, China

Data Deduplication



- Container
  - Large fixed-size storage unit
  - Basic unit of reads and writes
  - Preserving the spatial locality

Chunk Fragmentation

- Logically consecutive chunks are scattered in different containers

- Fragmentation degrades restore performance
  - Consecutive disk accesses → random ones
    - Penalty of disk seeks
  - Unreferenced chunks in retrieved containers
    - consume limited disk bandwidth

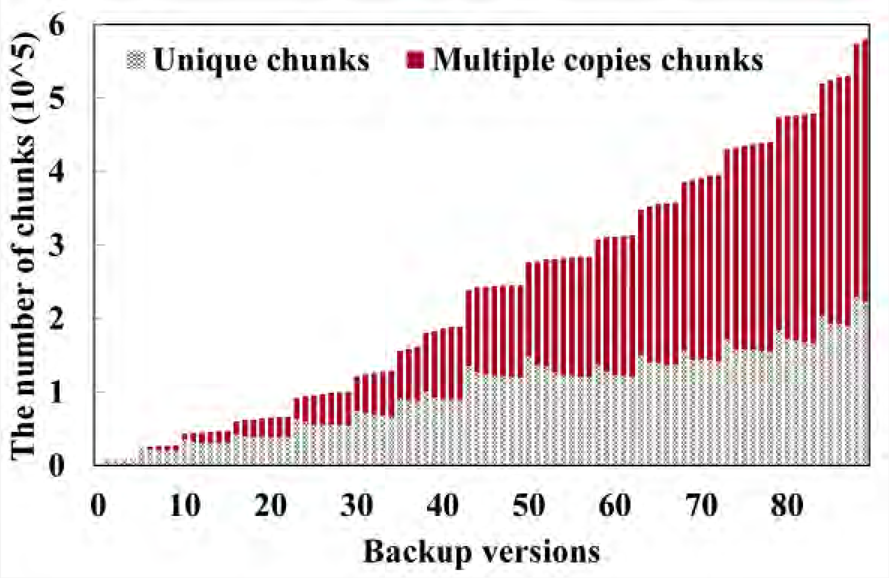
- Infrequent restore: very important, main concern

Existing Containers Selection Solutions

- Rewriting Schemes: Capping (FAST'13), NED (ICA3PP'14)
  - Trade off deduplication for reducing chunk fragmentation
  - Improve restore performance
- Select some containers to de-duplicate
  - Deduplicate chunks to identical copies in selected containers
  - Rewrite duplicate chunks belonging to other unselected containers into new containers
- How to select?
  - Capping: selects top T containers ranked by the reference ratio
  - NED: selects containers with the reference ratio over a threshold

Observation: Redundancy among Containers

- Rewrite: multiple identical copies stored in different containers
  - Causing redundant chunks in selected containers



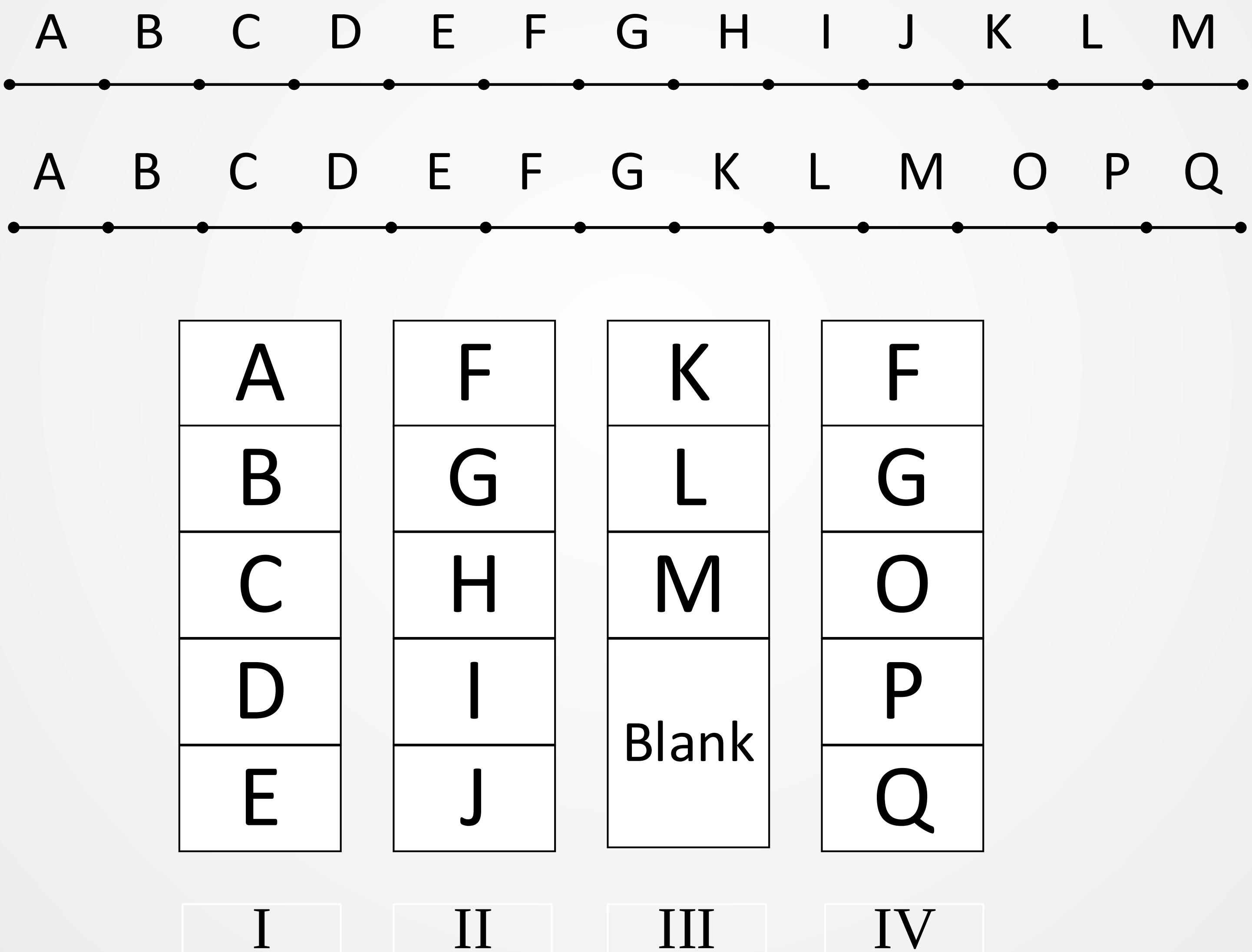
- Existing rewriting schemes are suboptimal due to overlooking the redundancy among containers

Example with Capping

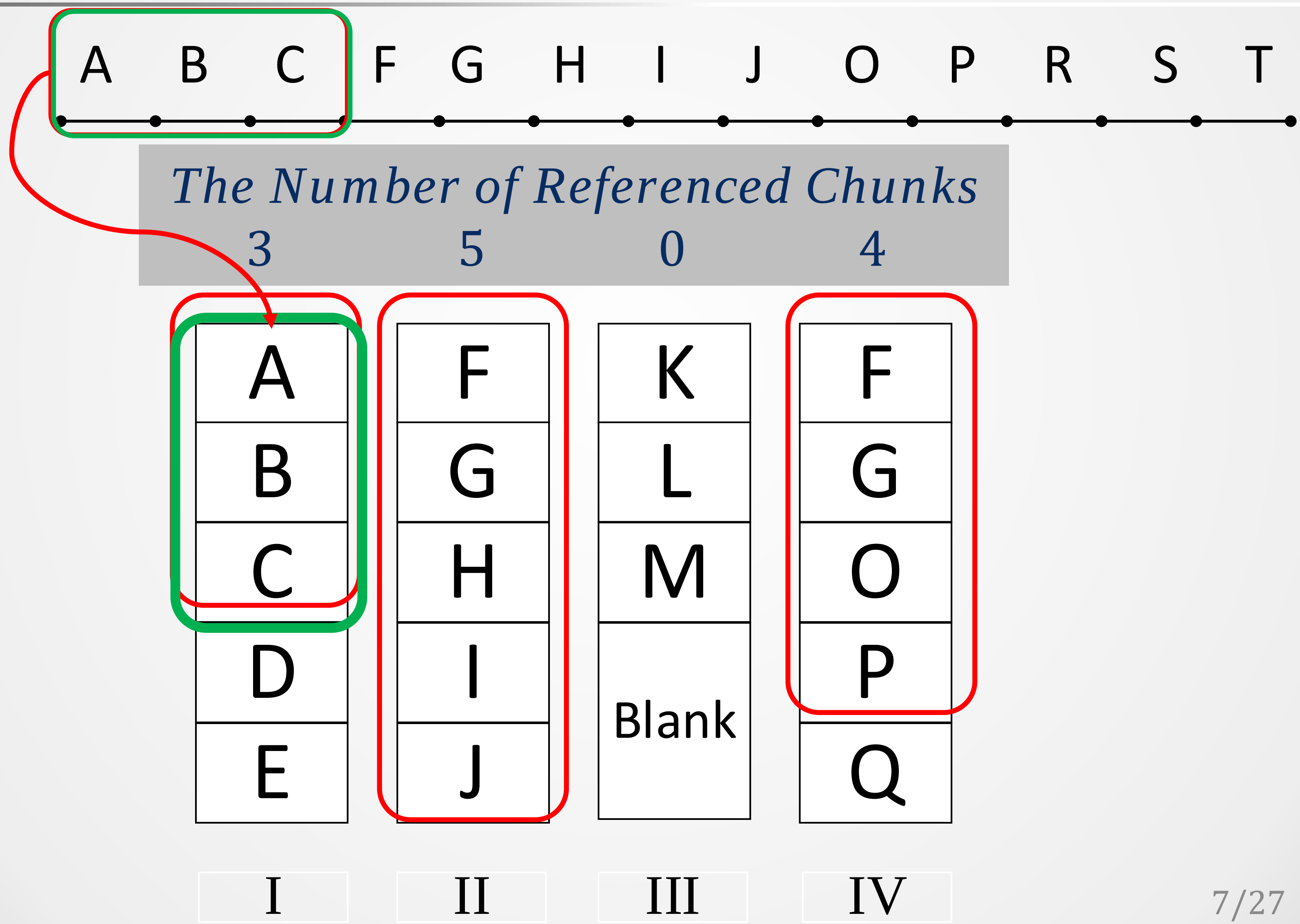
- Backup 3 consecutive data streams each with 13 chunks

- Capping
  - select top (T = 2) containers ranked by the number of referenced chunks
  - The container size: 5 chunks

Back up the First Two Data Streams



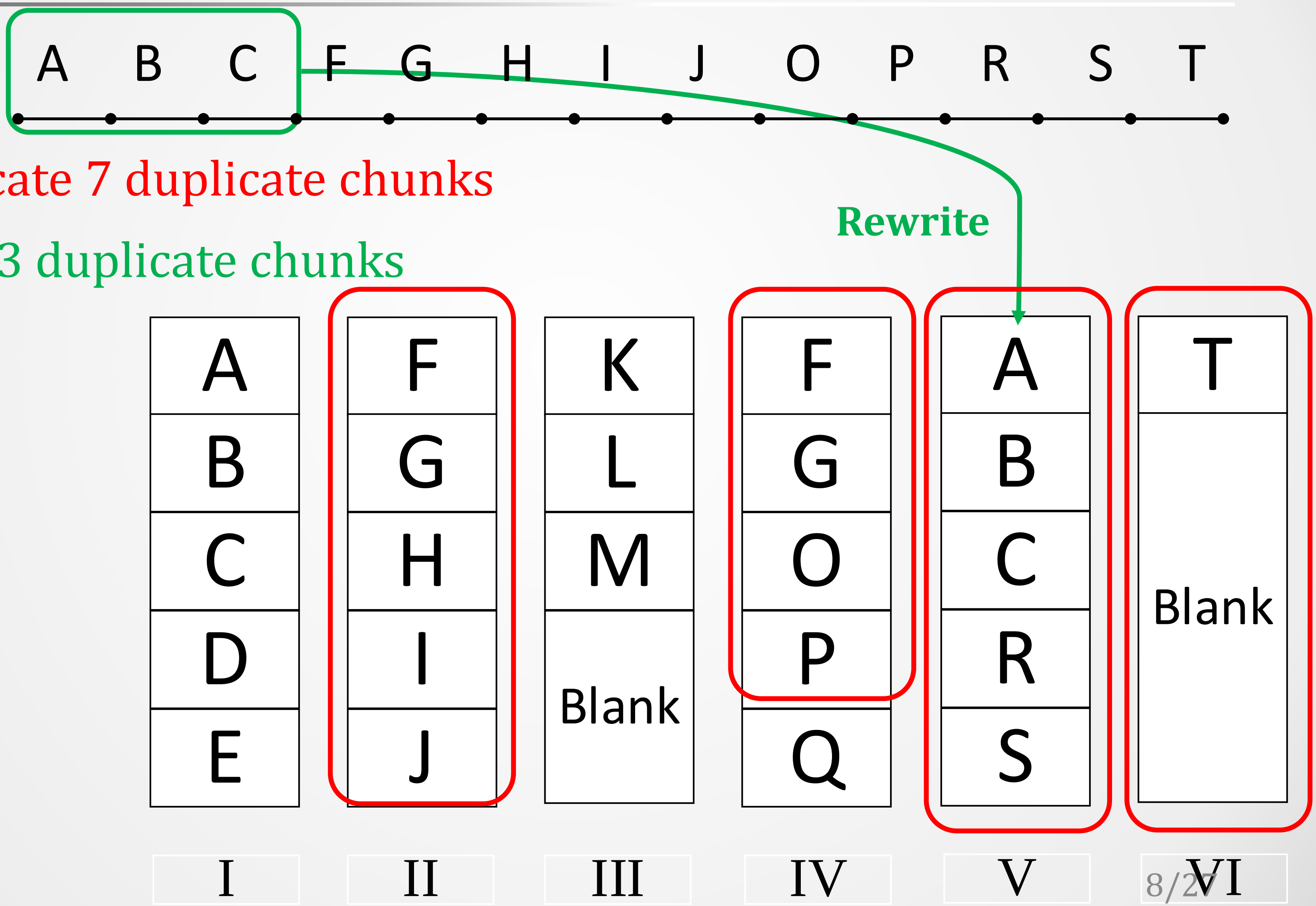
Back up the Third Data Stream



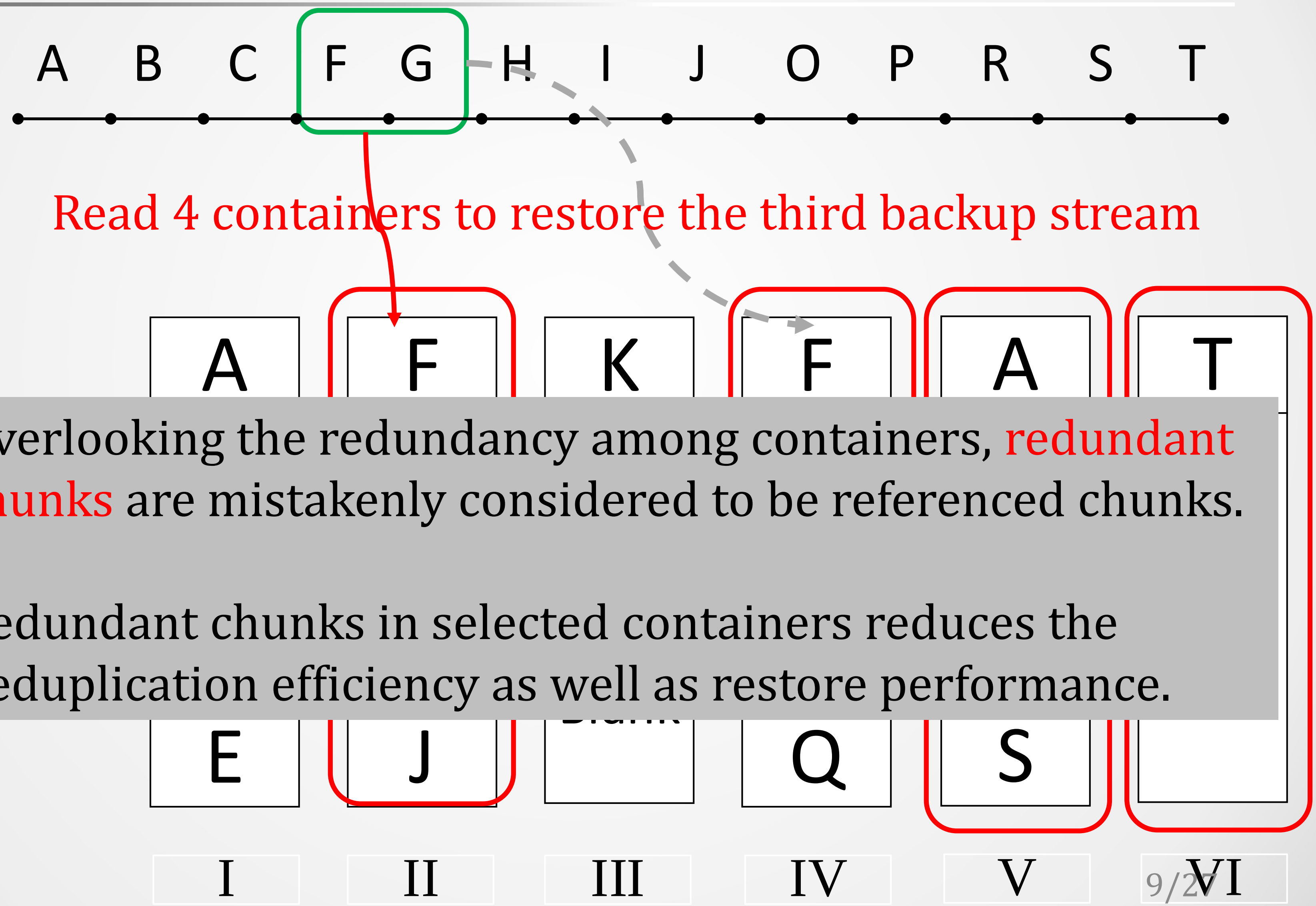
Back up the Third Data Stream

Deduplicate 7 duplicate chunks

Rewrite 3 duplicate chunks



Restore the Third Data Stream



- Overlooking the redundancy among containers, redundant chunks are mistakenly considered to be referenced chunks.
- Redundant chunks in selected containers reduces the deduplication efficiency as well as restore performance.

Motivation

- Review the observation
  - Redundancy among containers
  - Decrease the deduplicaton efficiency as well as restore performance

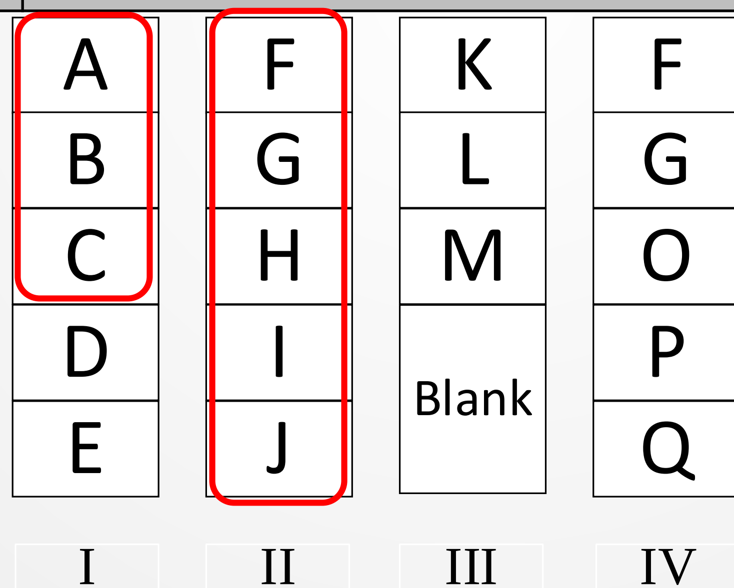
- Motivation
  - Consider the redundancy among containers when selecting containers
  - Select a fixed-size subset of containers with more distinct referenced chunks for deduplication

Same Example for Our Scheme

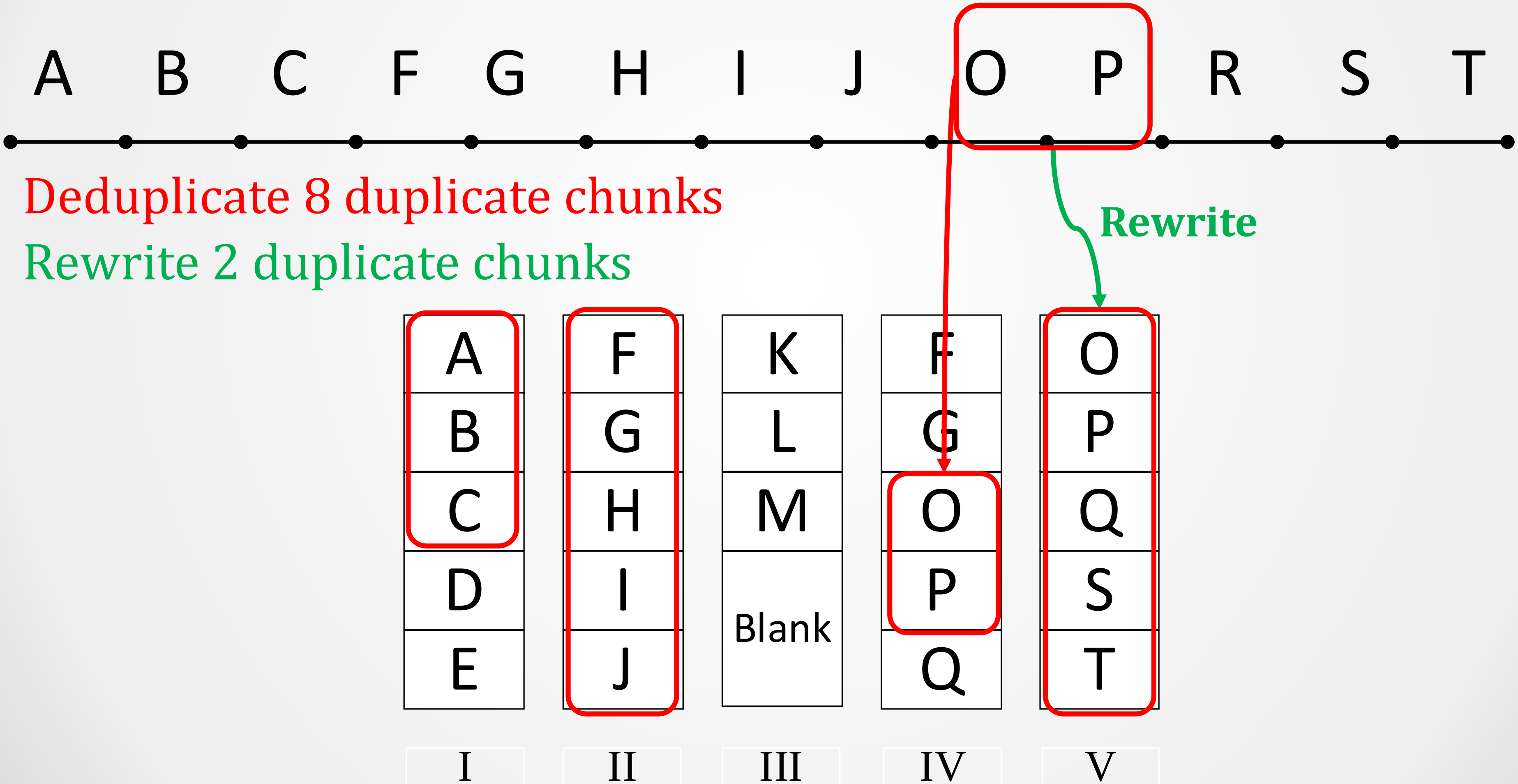
- Backup the same 3 data streams
- Selecting two containers for deduplication
- First and Second backup are ignored

Selecting a fixed-size subset of containers with more distinct referenced chunks for deduplication

Container ID	*Distinct Referenced Chunks*	*Chunks Amount*
I, II	A B C F G H I J	8
I, III	A B C	3
I, IV	A B C F G O P	7
II, III	F G H I J	5
II, IV	F G H I J O P	7
III, IV	F G O P	4



Same Example for Our Scheme: Backup

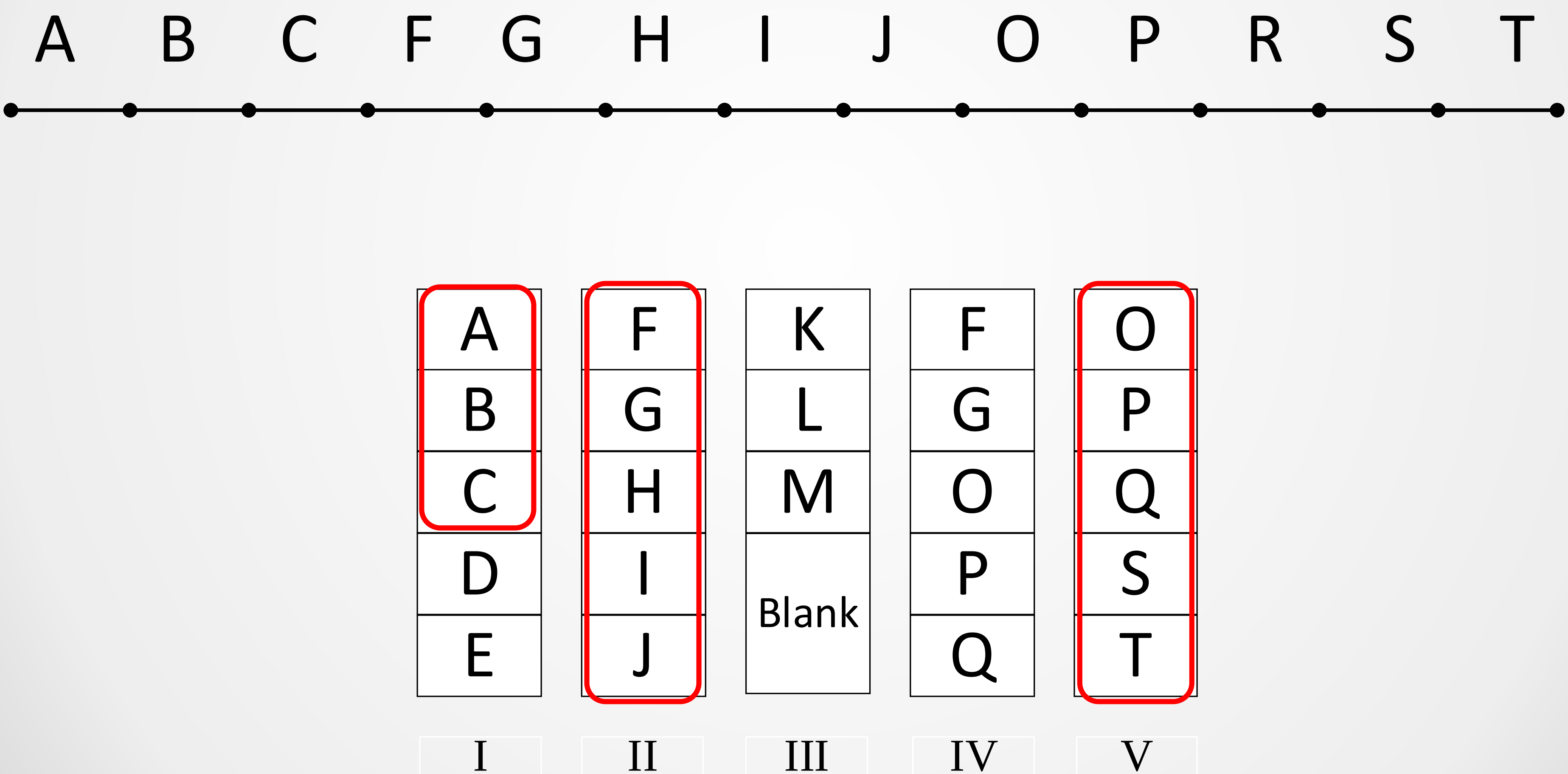


Deduplicate 8 duplicate chunks

Rewrite 2 duplicate chunks

Same Example for Our Scheme: Restore

Read 3 containers to restore the third backup stream



Comparisons

Selecting a fixed-size subset of containers with more distinct referenced chunks for deduplication

	Capping	*Our Scheme*
Deduplicate	7 chunks	8 chunks
Rewrite	3 chunks	2 chunks
Reads for Restore	4 containers	3 containers

- Selecting containers with more distinct referenced chunks
  - De-duplicate more chunks
  - Rewrite less chunks
  - Read less containers in restore
- Better trade-off: achieving higher deduplication ratio and also improving restore performance.

SMR: A Submodular Maximization Rewriting Scheme

- Select a subset of containers with more distinct referenced chunks
  - Reduce the disk accesses for unreferenced and redundant chunks

- The number of containers in the subset is limited
  - Limit the number of containers read from disks

Formulate the Subset Selection Problem

- Given a set of old containers to be selected $V = (C_1, C_2, \ldots, C_{|V|})$
- A budget T, the limited number of selected containers:
- Find a container subset S ($S \subseteq V$, $|S| \leq T$), offering the largest number of distinct referenced chunks for the backup
- The subset selection can be performed by computing:

$$S^* \in \operatorname*{argmax}_{S \subseteq V} F(S) \qquad s.t. \; |S| \leq T.$$

The Scoring Function: F(S)

- A scoring function $F : 2^V \rightarrow \mathbb{R}$: the amount of distinct referenced chunks in a subset

$$F(S) = \left| \bigcup_{C_i \in S} w(C_i) \right| .$$

- $w(C_i)$: all referenced chunks in container C_i

- $F(S)$ is a monotone submodular function

How to Compute S*?

$$S^* \in \underset{S \subseteq V}{\operatorname{argmax}} F(S) \quad s.t. |S| \leq T.$$

- Computing S* is intractable
  - Selecting T containers from N containers: $\binom{T}{N}$ possible cases
- Naive scheme to compute S*
  - Emulate all possible container subsets
  - Rank these subsets by the scores and select one with the highest score
  - Time and compuation inefficiency
- Our Scheme
  - F(S) is monotone submodular function (MSF)
  - Greedy algorithm is time-efficient for computing the maxization for MSF
  - Constant-factor mathematical quality guarantee

Evaluation Datasets

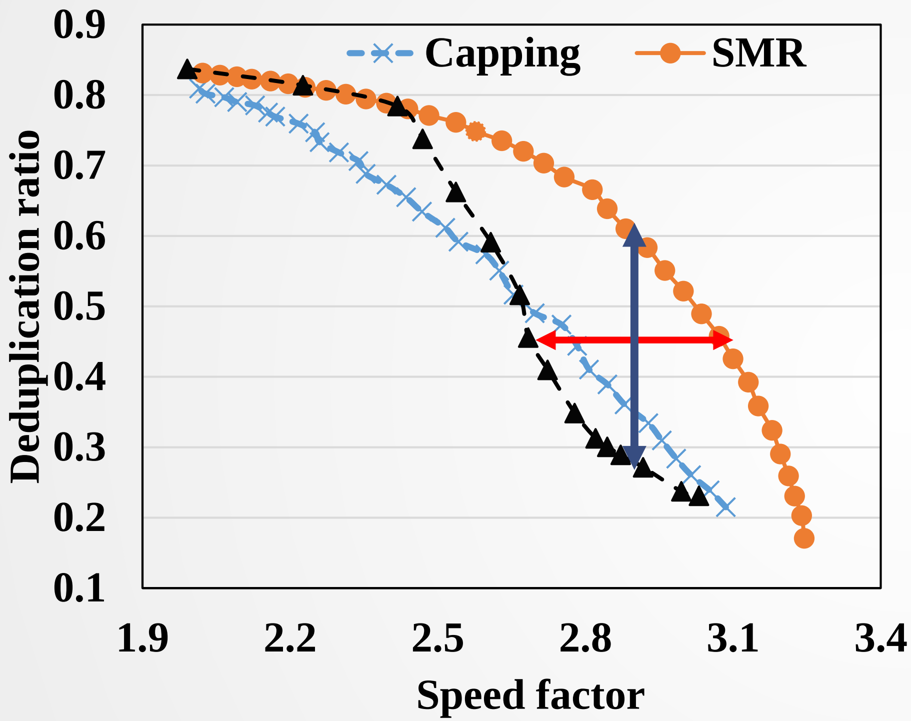
Datasets	GCC	Linux
Total size	56GB	97GB
# of versions	89	96
Version numbers	2.95 to 6.1.0	4.0 to 4.7

- GCC: source code of the GNU Complier Collection
- Linux: unpacked linux kernel sources

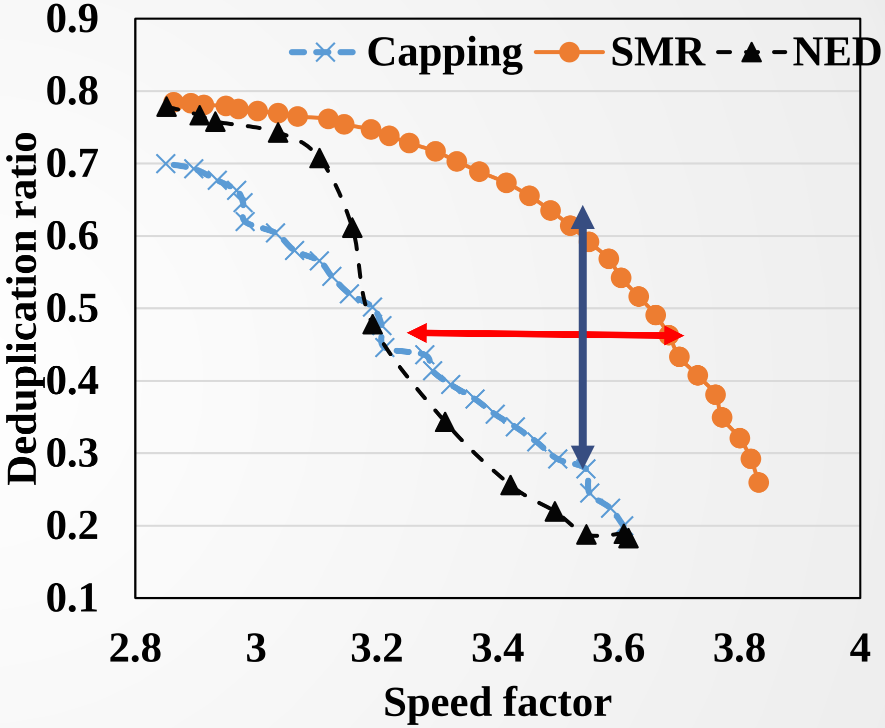
Evaluations Metrics

- Speed factor: 1 divided by mean container read per MB of data restored (restore performance)
- Deduplication ratio: the ratio of total size of the removed duplicate chunks to that of all backed up chunks (deduplication performance)
- Deduplication throughput: the amount of backed up data per second (deduplication performance)

Deduplication Ratio vs. Speed Factor



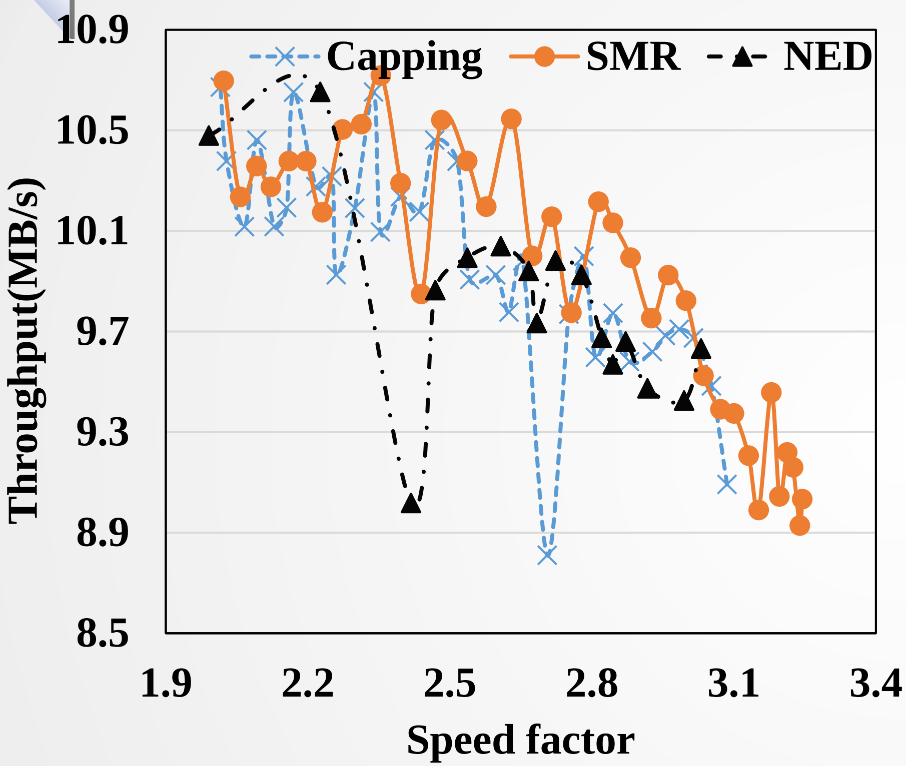
GCC dataset



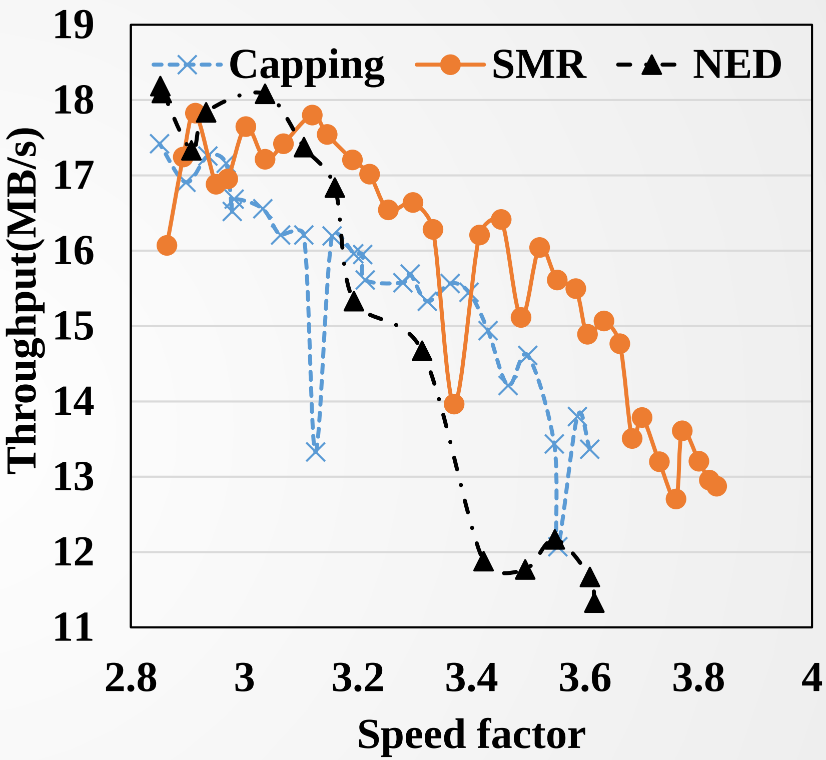
Linux dataset

SMR achieves better trade-off between restore performance and deduplication ratio

Deduplication Throughput vs. Speed Factor



GCC dataset

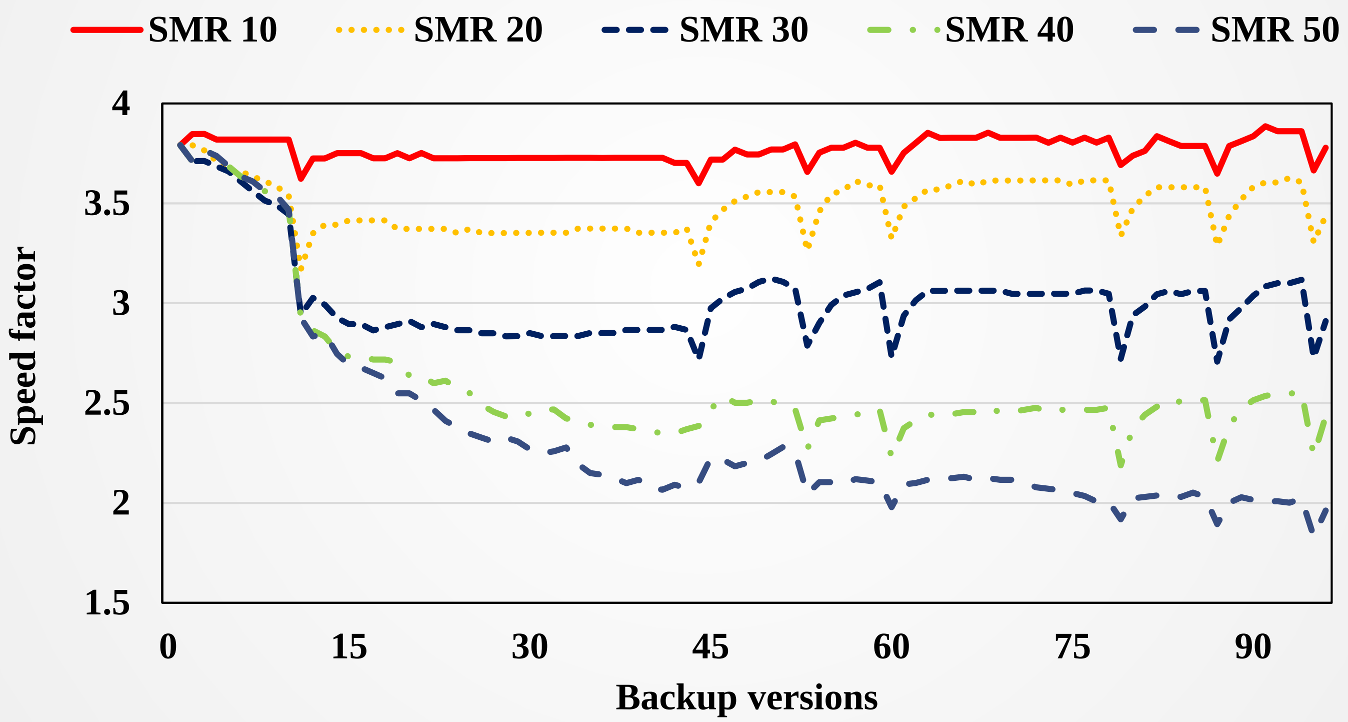


Linux dataset

In most cases, SMR achieves higher throughputs

The Effects of the Budget T

- SMR T: selecting T containers for each data segment



- Smaller T results in higher speed factor
- T is adjustable to meet the needs of different restore performance

Conclusion

- Fragmentation severely degrades restore performance
- Existing work addressing the problem is suboptimal due to overlooking redundancy among containers
- We propose a submodular maximization rewriting scheme SMR
  - Consider the redudancy among containers when selecting containers
  - Select more suitable containers by a submodular maximization model
- SMR outperforms the state-of-the-art work in both restore performance and deduplication ratio

Thanks & Questions

Open-source Code: https://github.com/courageJ/SMR
E-mail: wujie@hust.edu.cn

The Monotone Submodular Function

- Submodular
  - A set function F is submodular if for any set $A \subseteq B \subset V$, and $v \in V \setminus B$, we have that:
    $$F(A + \{v\}) - F(A) \geq F(B + \{v\}) - F(B)$$
  - Adding v to smaller set A brings more benefit than to larger set B
  - Referenced chunks in new container v
    - Distinct to A
    - Redundant to set B
    - Incremental number of distinct referenced chunks of A ≥ number of B
- Monotone
  - A set function F is monotone if for any set $A \subseteq B \subset V$, we have that:
    $$F(B) \geq F(A)$$
  - The number of distinct referenced chunks of B ≥ number of A.