The Dilemma between Deduplication and Locality: Can Both be Achieved?

Xiangyu Zou[1], Jingsong Yuan[1], Philip Shilane[2], Wen Xia[1,3], Haijun Zhang[1], and Xuan Wang[1]

[1]Harbin Institute of Technology, Shenzhen; [2]Dell Technologies
[3]Wuhan National Laboratory for Optoelectronics

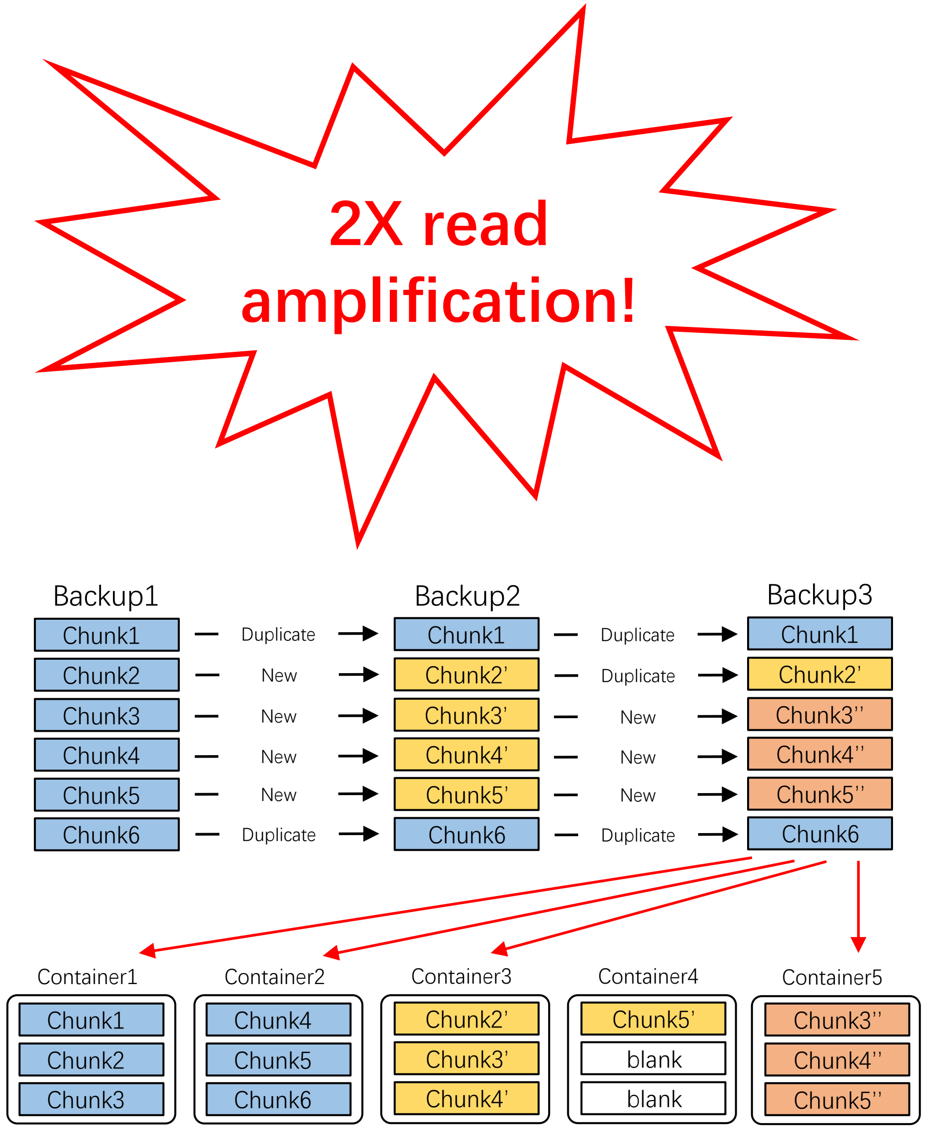
USENIX FAST ’21

Background

- Data Deduplication
  - Several steps
  - Container-based I/O
- Backup storage
  - Achieving about 10X-30X deduplication ratio
  - Workload: successive snapshots of the primary data
- Deduplication & Locality
  - Common chunks are shared
  - Locality of backups is broken

Background

- Fragmentation problem
  - Read amplification
  - Random access
  - Garbage collection

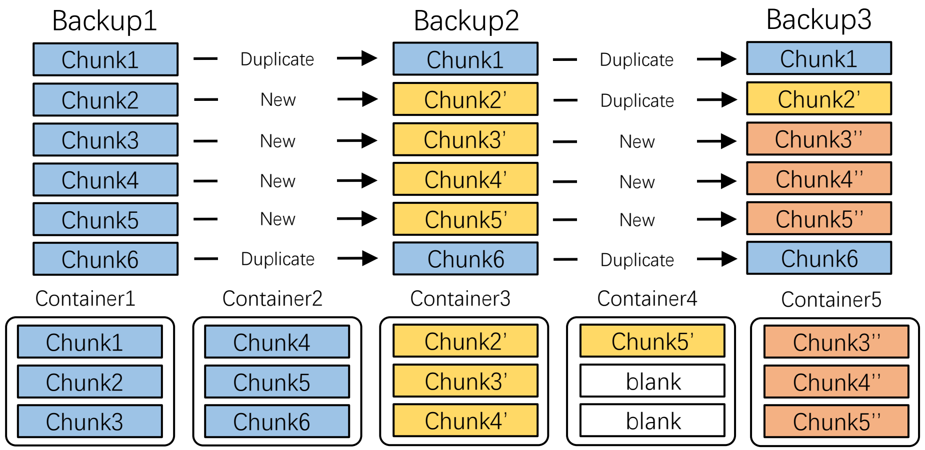


Related Work

- Rewrite techniques
  - Rewrite some duplicates according to their ‘fragmentation degree’ to maintain a level of data locality
  - CBR(Kaczmarczyk@SYSTOR’12)
  - Capping(Lillibridfe@FAST’13)
  - HAR(Fu@ATC’14)
- Fragment problem is alleviated with huge cost
  - Read amplification remains 2X – 4X
  - Deduplication ratio is sacrificed (10%~40%)

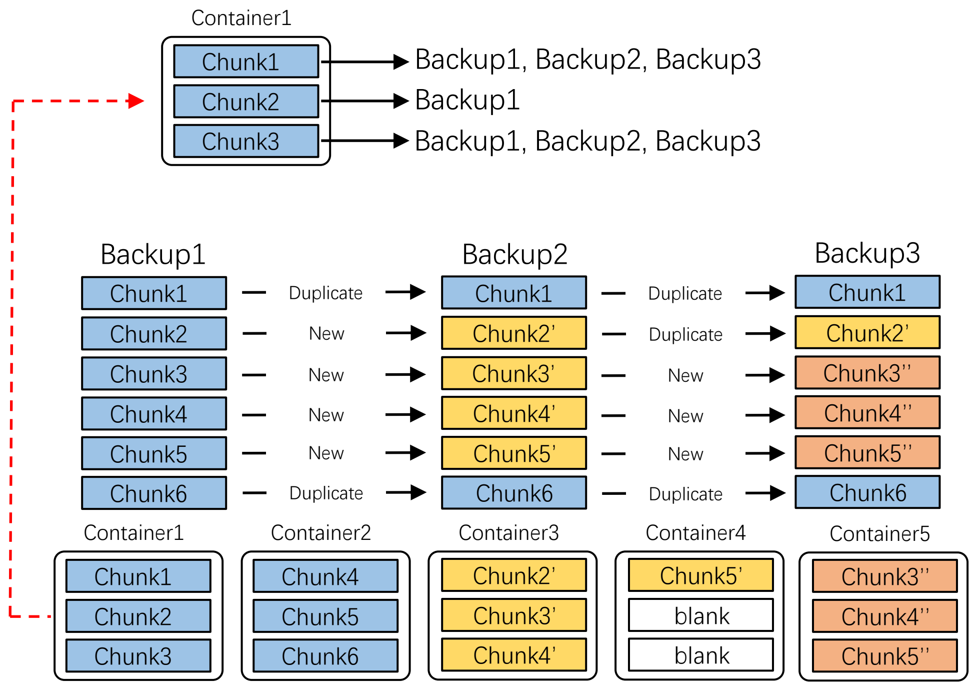
Observation & Motivation

- How read amplification is generated?
  - Container-based I/O
  - Layout of chunks is the key issue
  - Chunks in containers have different lifecycles



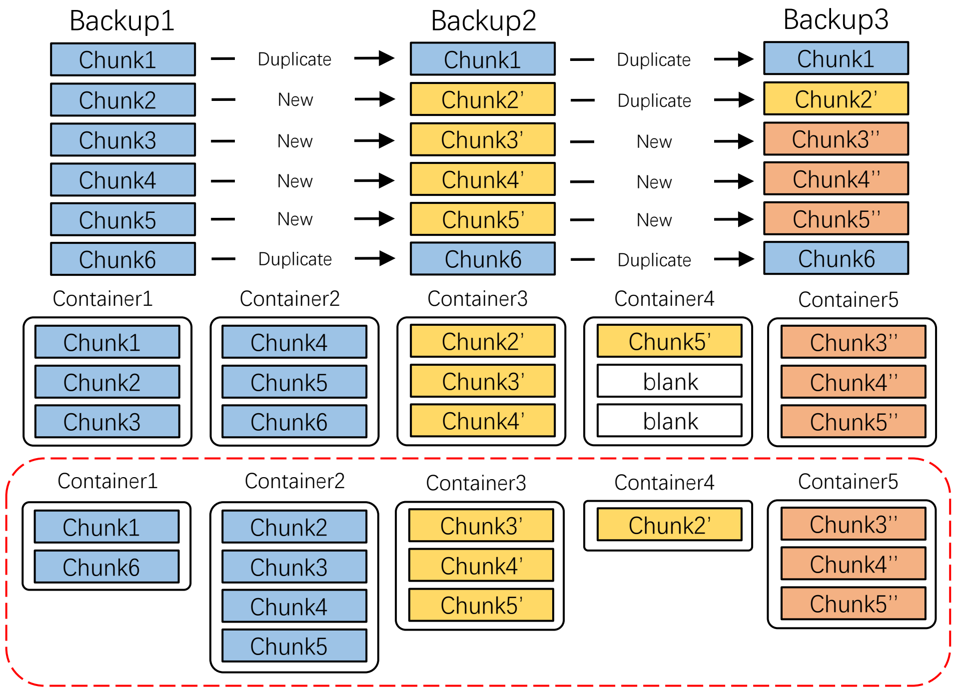
Observation & Motivation

- How read amplification is generated?
  - Container-based I/O
  - Layout of chunks is the key issue
  - Chunks in containers have different lifecycles



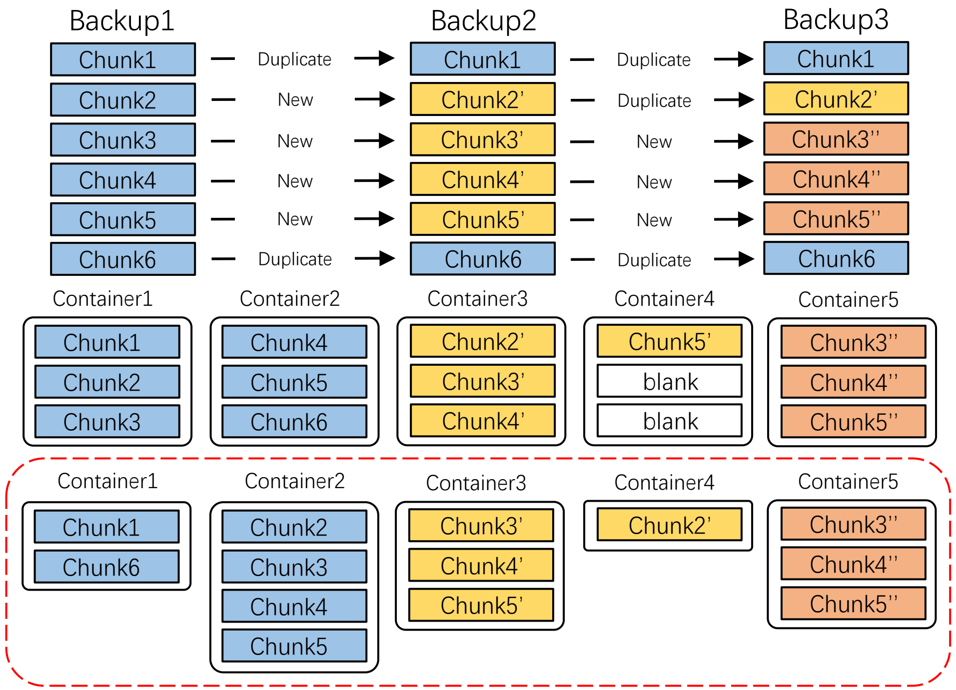
Observation & Motivation

- How read amplification is generated?
  - Container-based I/O
  - Layout of chunks is the key issue
  - Chunks in containers have different lifecycles
- How to avoid read amplification?
  - Chunks in containers have the same lifecycle
  - Classifying chunks with their lifecycle into categories
  - Each category maps to a container



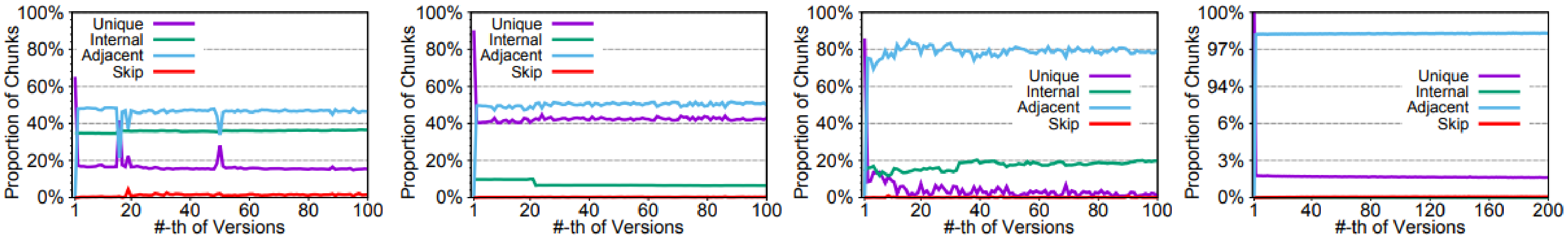
Observation & Motivation

- How read amplification is generated?
  - Container-based I/O
  - Layout of chunks is the key issue
  - Chunks in containers have different lifecycles
- How to avoid read amplification?
  - Chunks in containers have the same lifecycle
  - Classifying chunks with their lifecycle into categories
  - Each category maps to a container
- But…
  - The amount of categories will be very large!!
  - Up to $2^n - 1$, unacceptable (n is the number of backups)



Observation & Motivation

- How to reduce the number of categories.
- We denote four kinds of chunks in a backup version B_i as follows:
  - **Internal duplicate chunks**: exist identical chunks in Bi.
  - **Adjacent duplicate chunks**: exist identical chunks in Bi-1.
  - **Skip duplicate chunks**: exist identical chunks in Bi-2, Bi-3, or ….
  - **Unique chunks**: no identical chunks.
- Avoiding deduplicating Skip chunks slightly impacts deduplication ratio.
- The number of categories will be reduced to $n(n+1)/2$.



(a) WEB Dataset (b) CHM Dataset (c) VMS Dataset (d) SYN Dataset

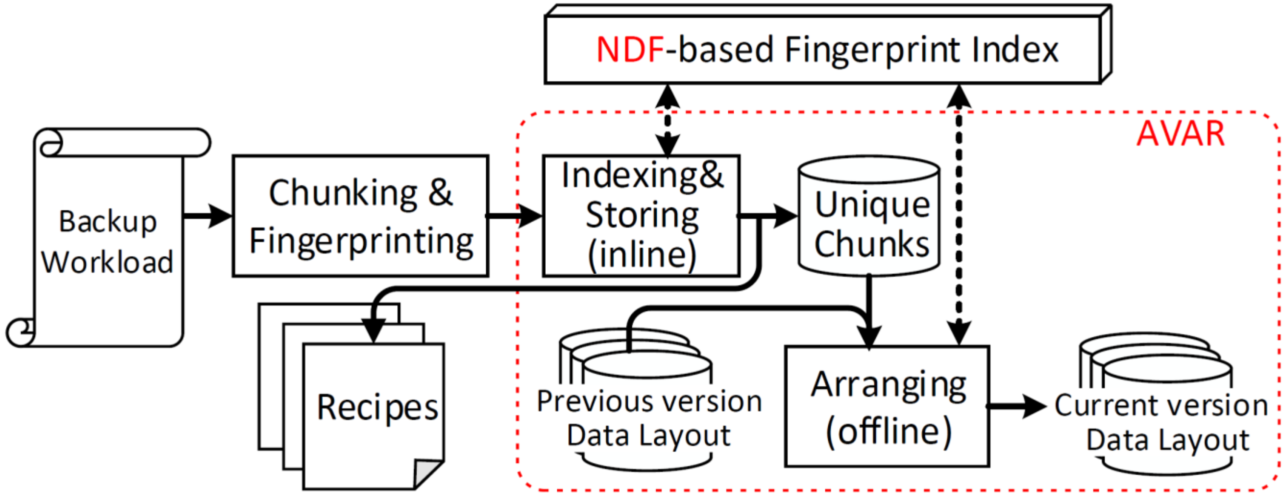
Figure 2: Distribution of four kinds of chunks on four backup datasets. *Skip* duplicate chunks are the least common.

Remaining Challenges

- We create a feasible chunk layout of deduplicated data with no read amplification.
- How to acquire and keep this kind of chunk layout?
  - Reorganizing all chunks after each backup written is costly.
  - Mathematical-induction-like approach is considered.

Our approach

- Our approach implements iterative evolution of our classification-based chunk layout.
- Two techniques
  - Neighbor-Duplicate-Focus indexing
  - Across-Version-Aware Reorganization
- No read amplification in restoring
- Completely eliminates garbage collection (mark-sweep)



Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj

Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj

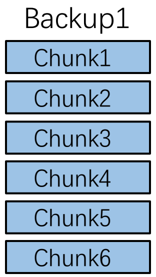
Start storing Backup1

Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj

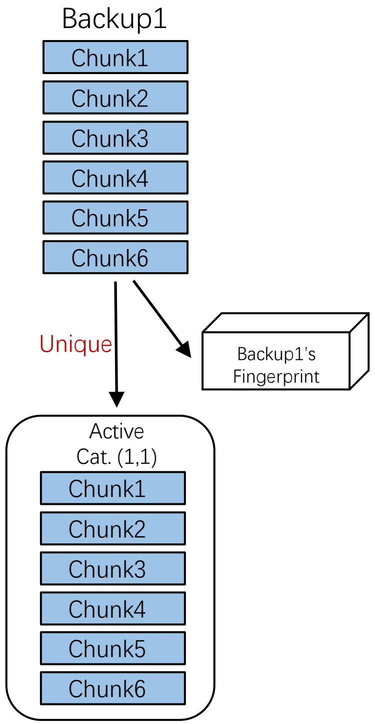
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



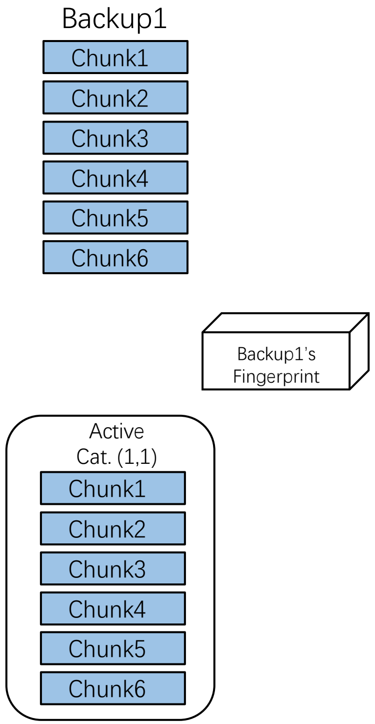
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



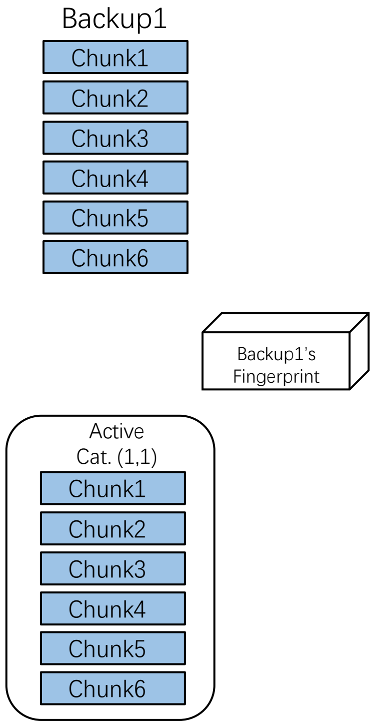
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



Iterative Evolution

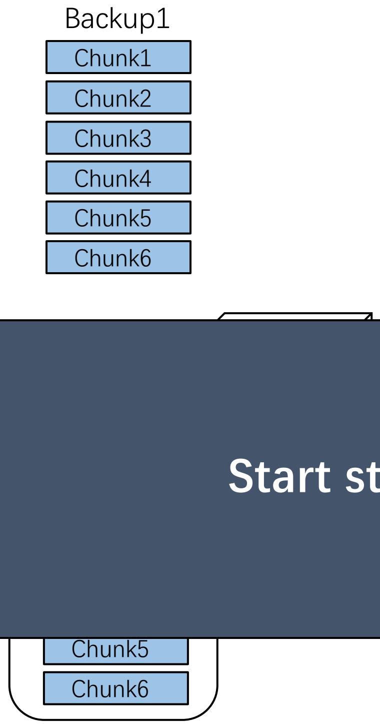
Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



Natural classification-based layout, do not require arranging.

Iterative Evolution

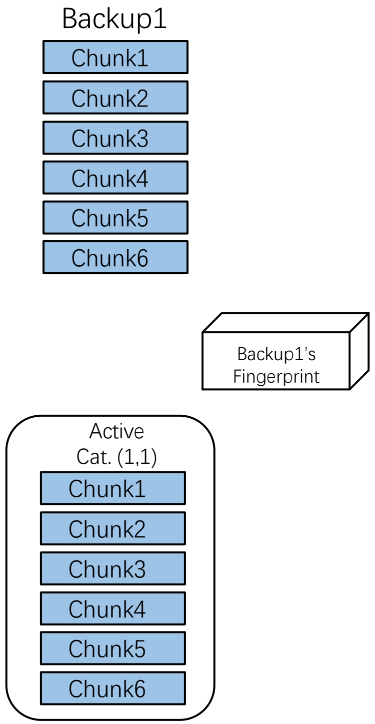
Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



Start storing Backup2

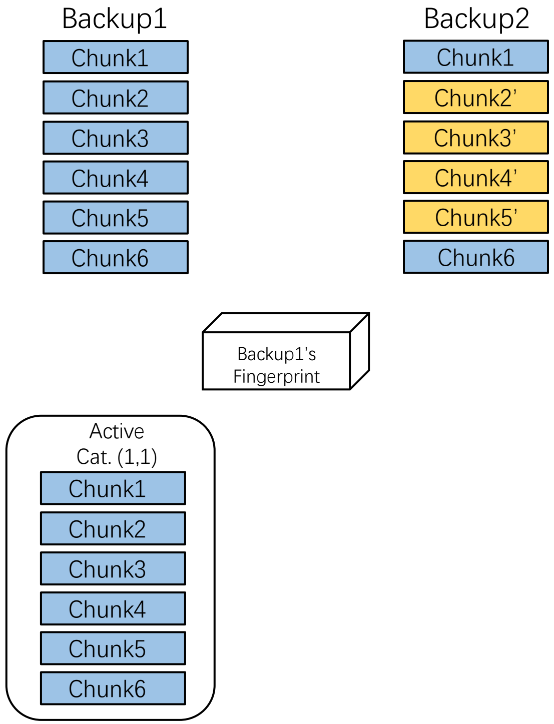
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



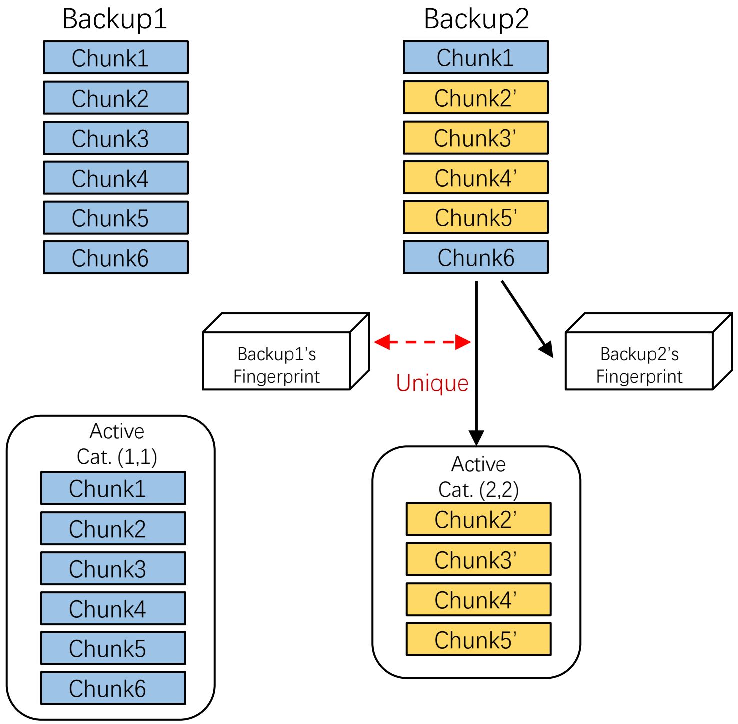
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



Iterative Evolution

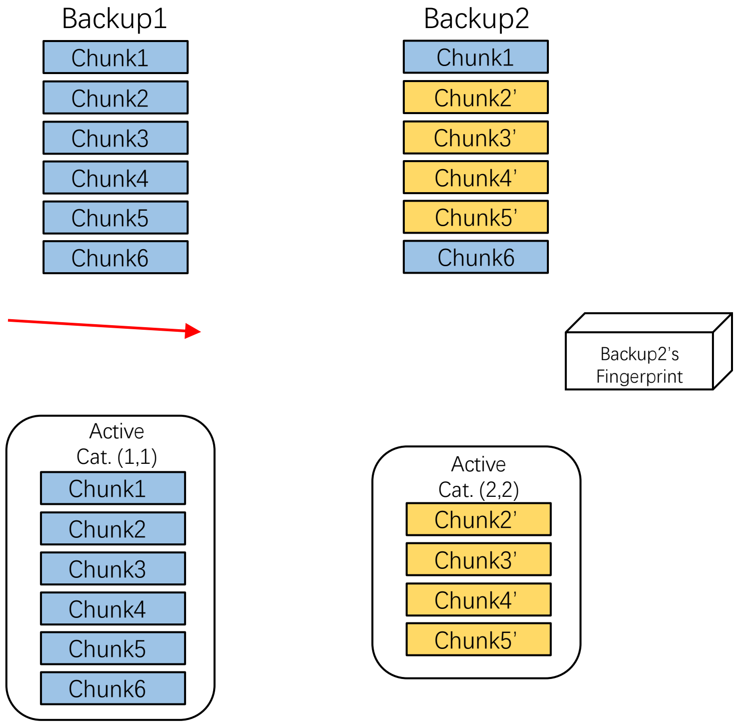
Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



Iterative Evolution

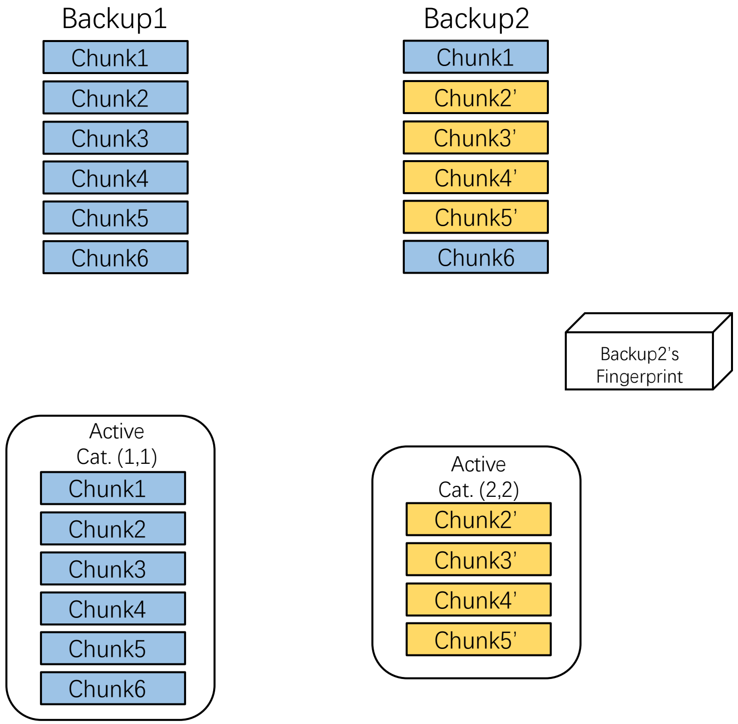
Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj

Backup1's Fingerprint is already useless, release it.



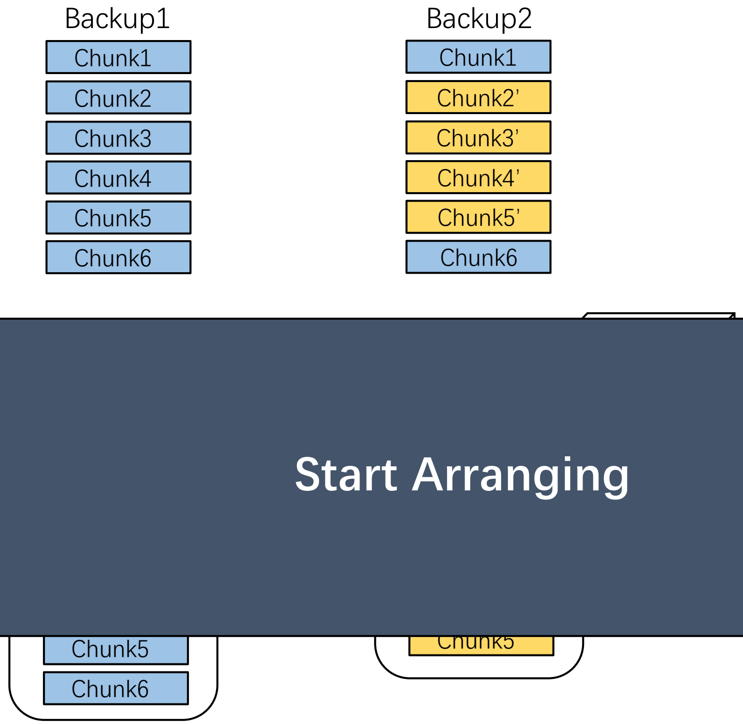
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



Iterative Evolution

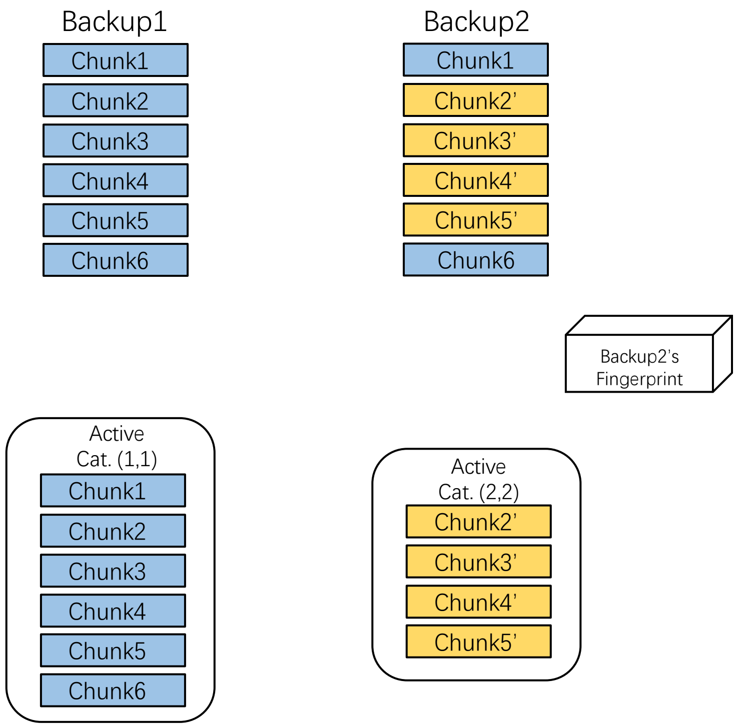
Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



Start Arranging

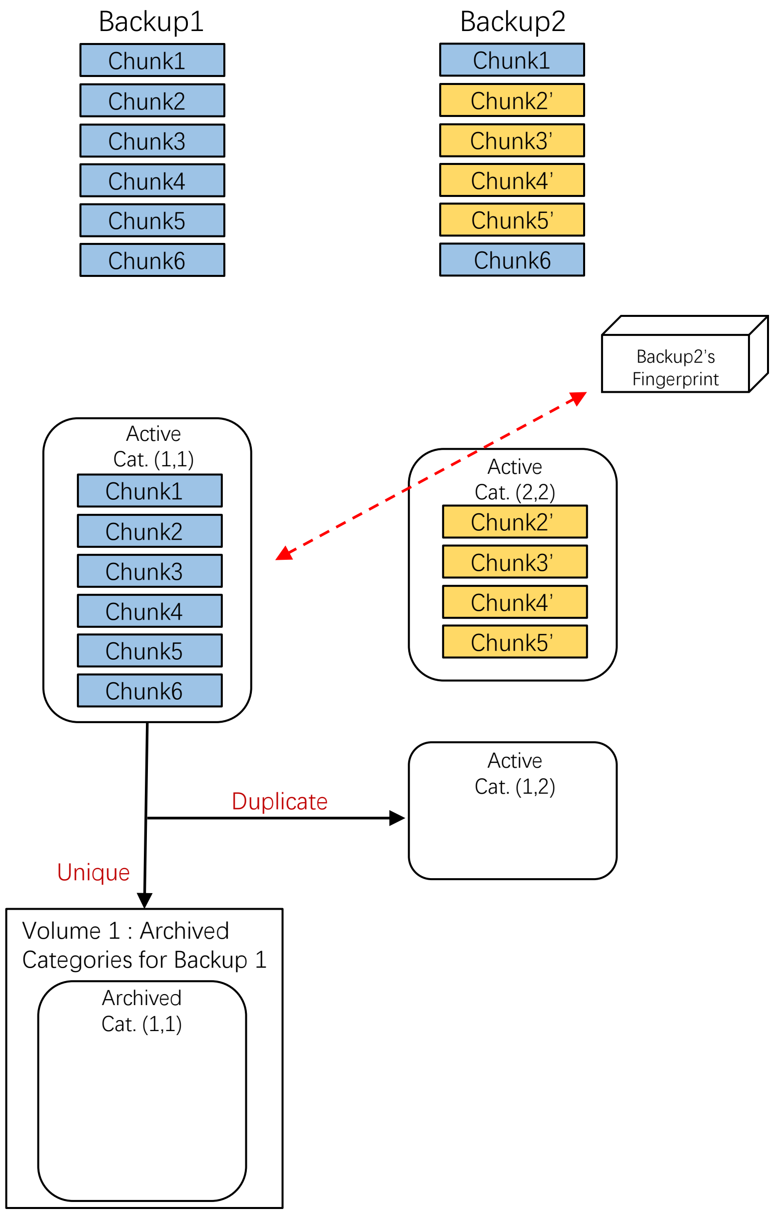
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



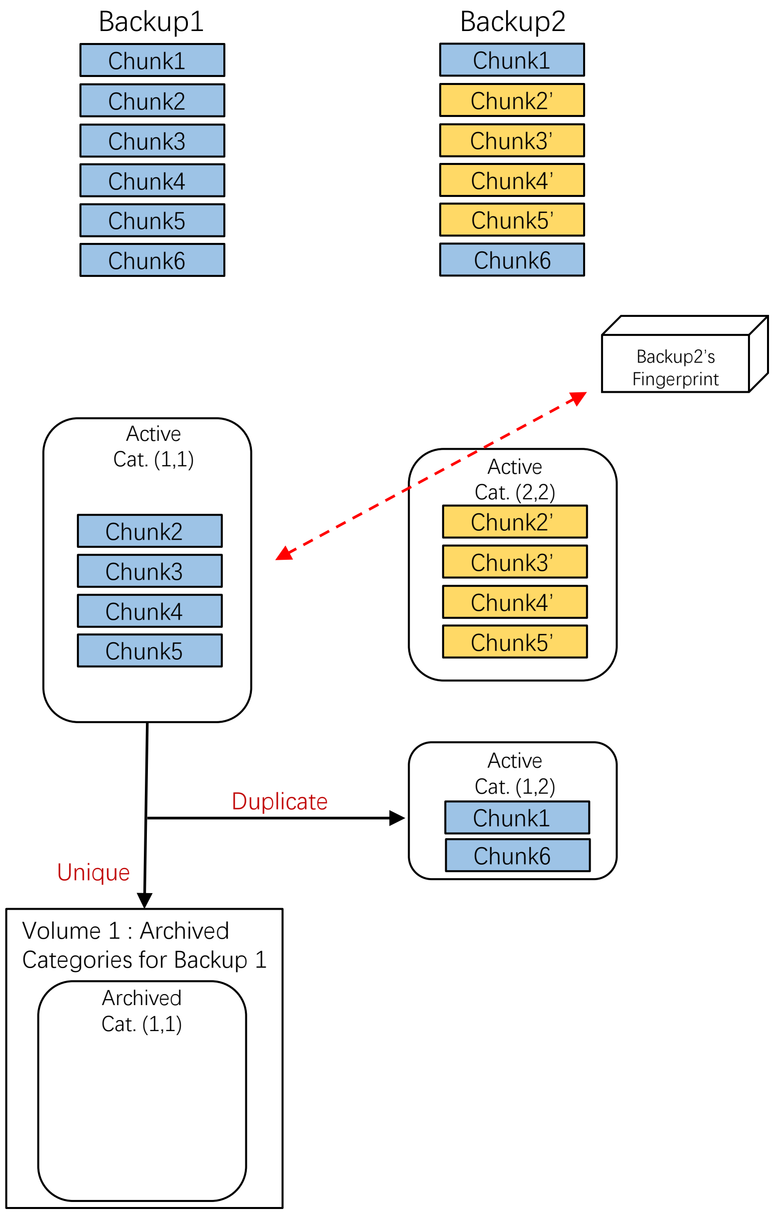
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



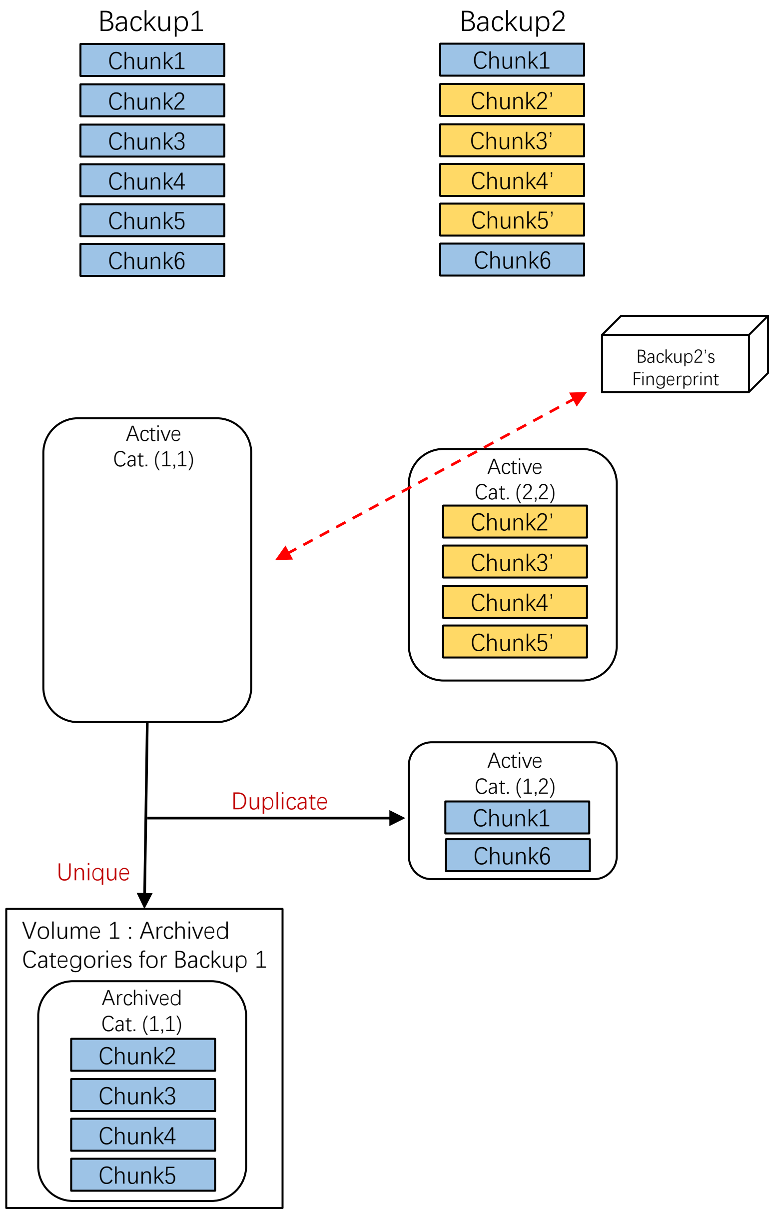
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



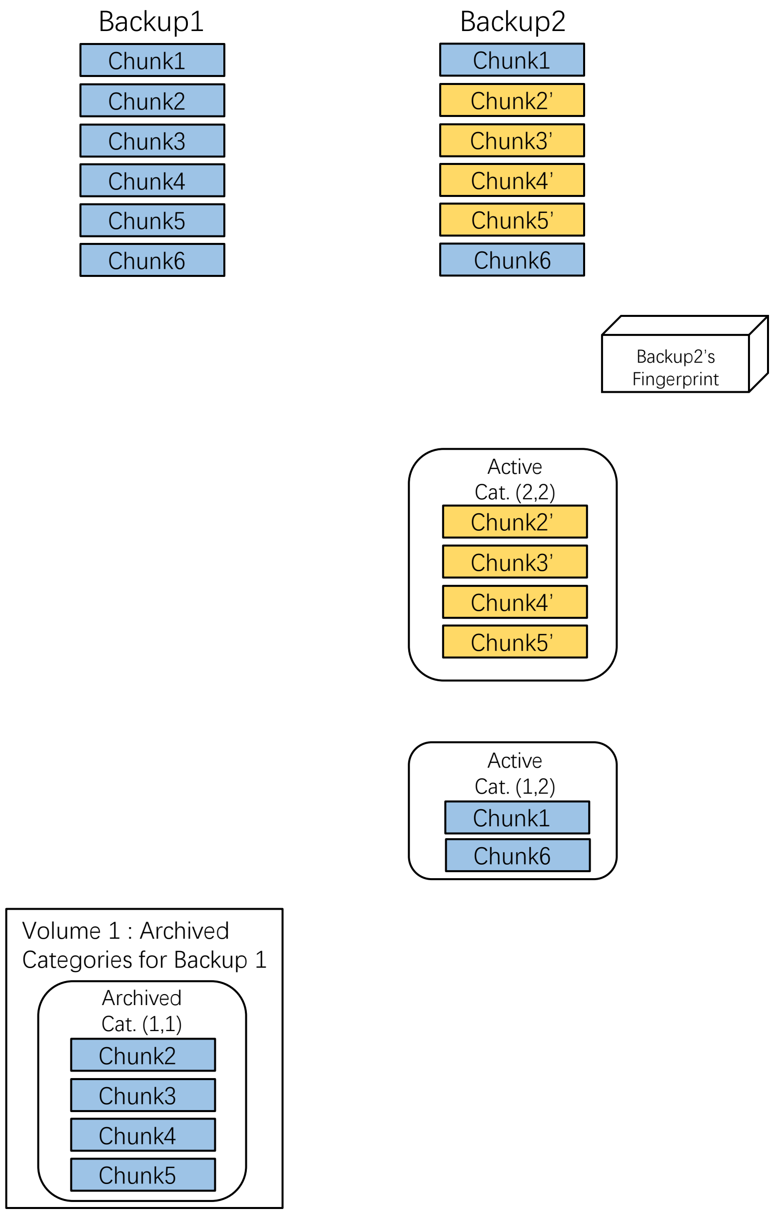
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



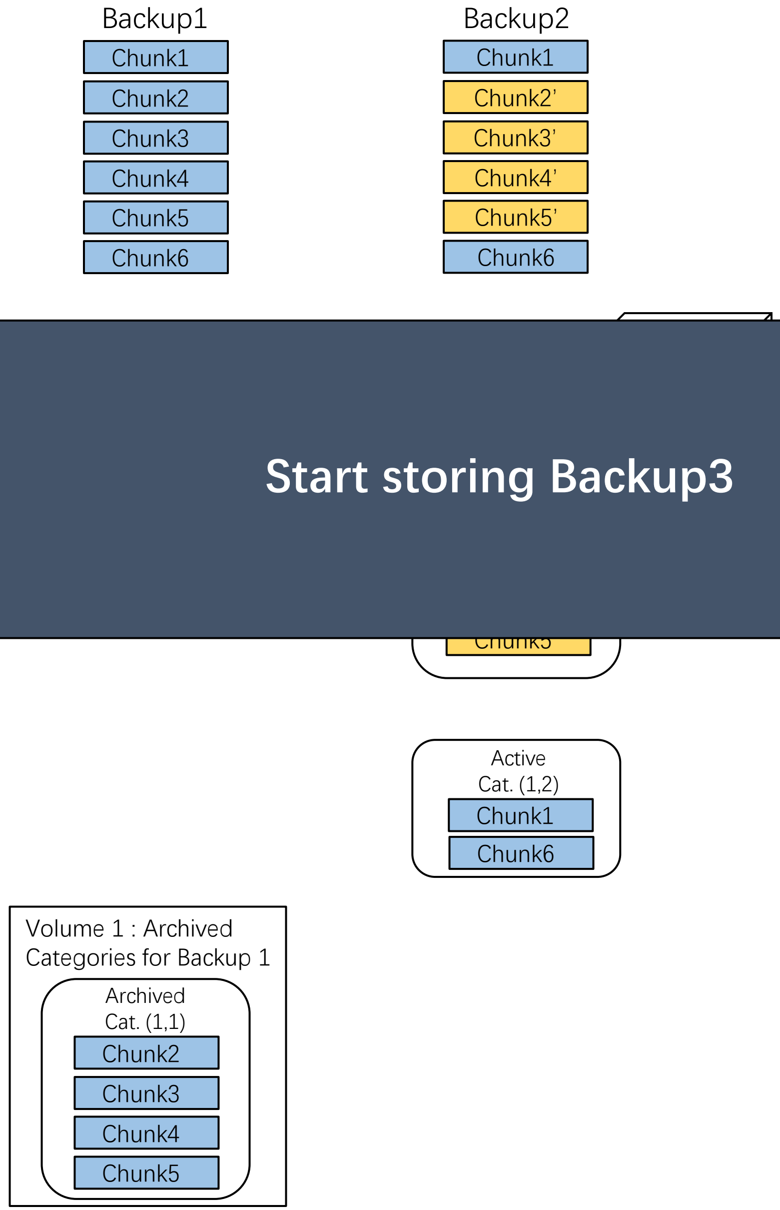
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



Iterative Evolution

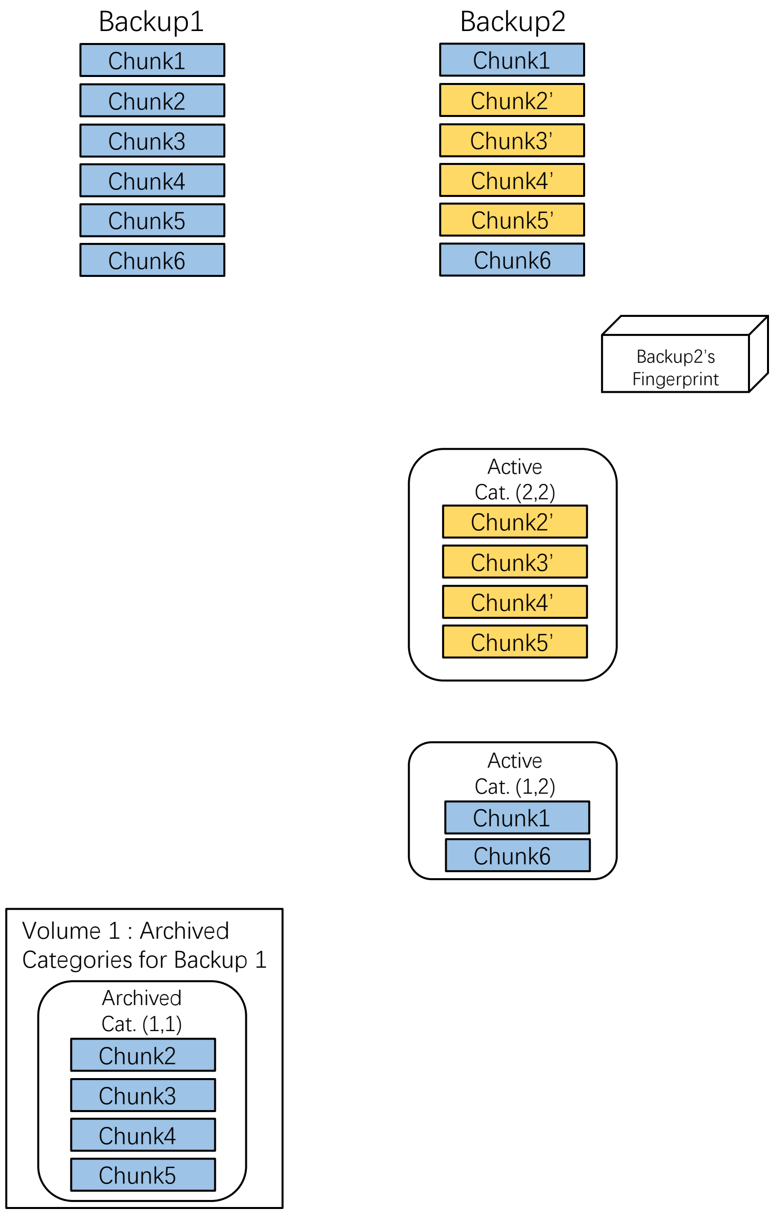
Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



Start storing Backup3

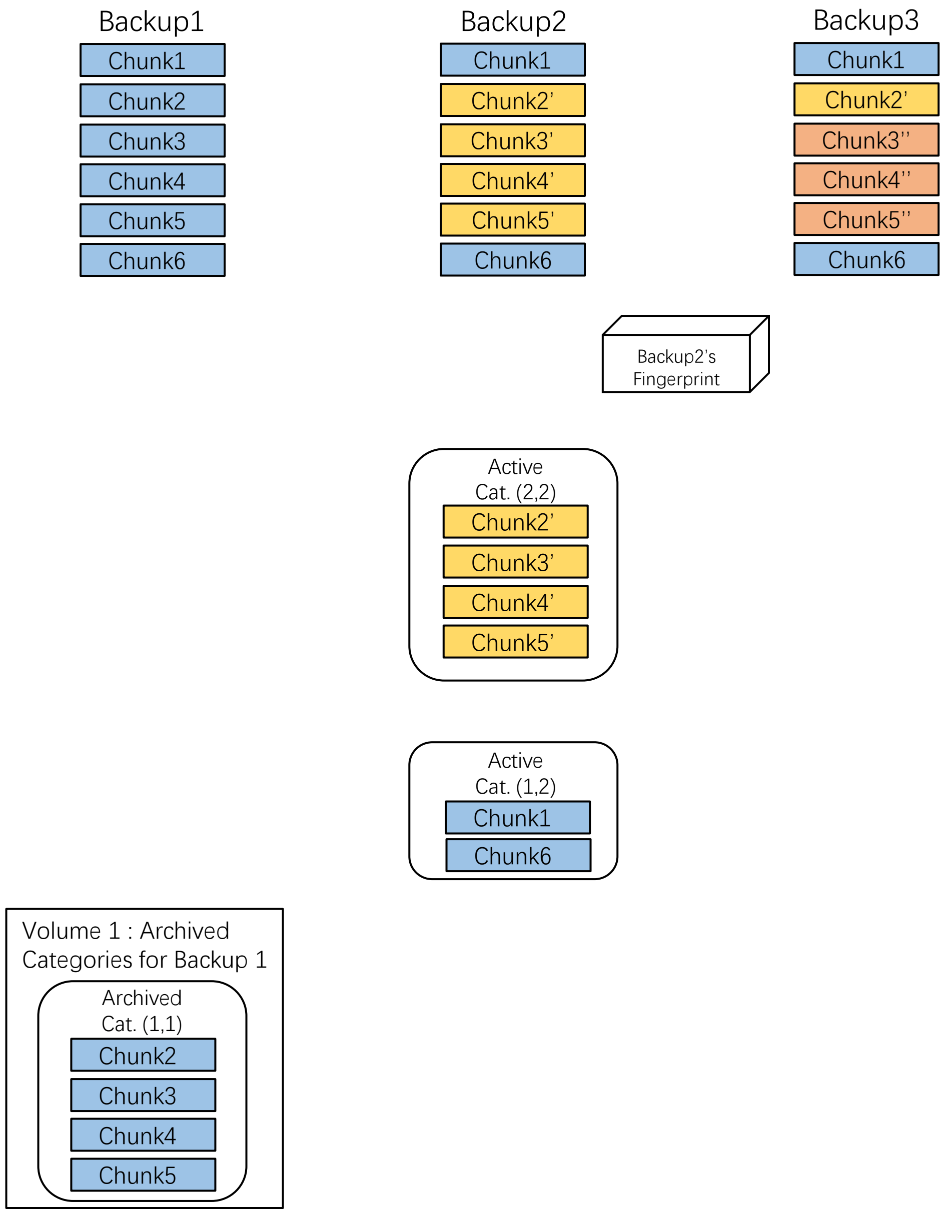
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



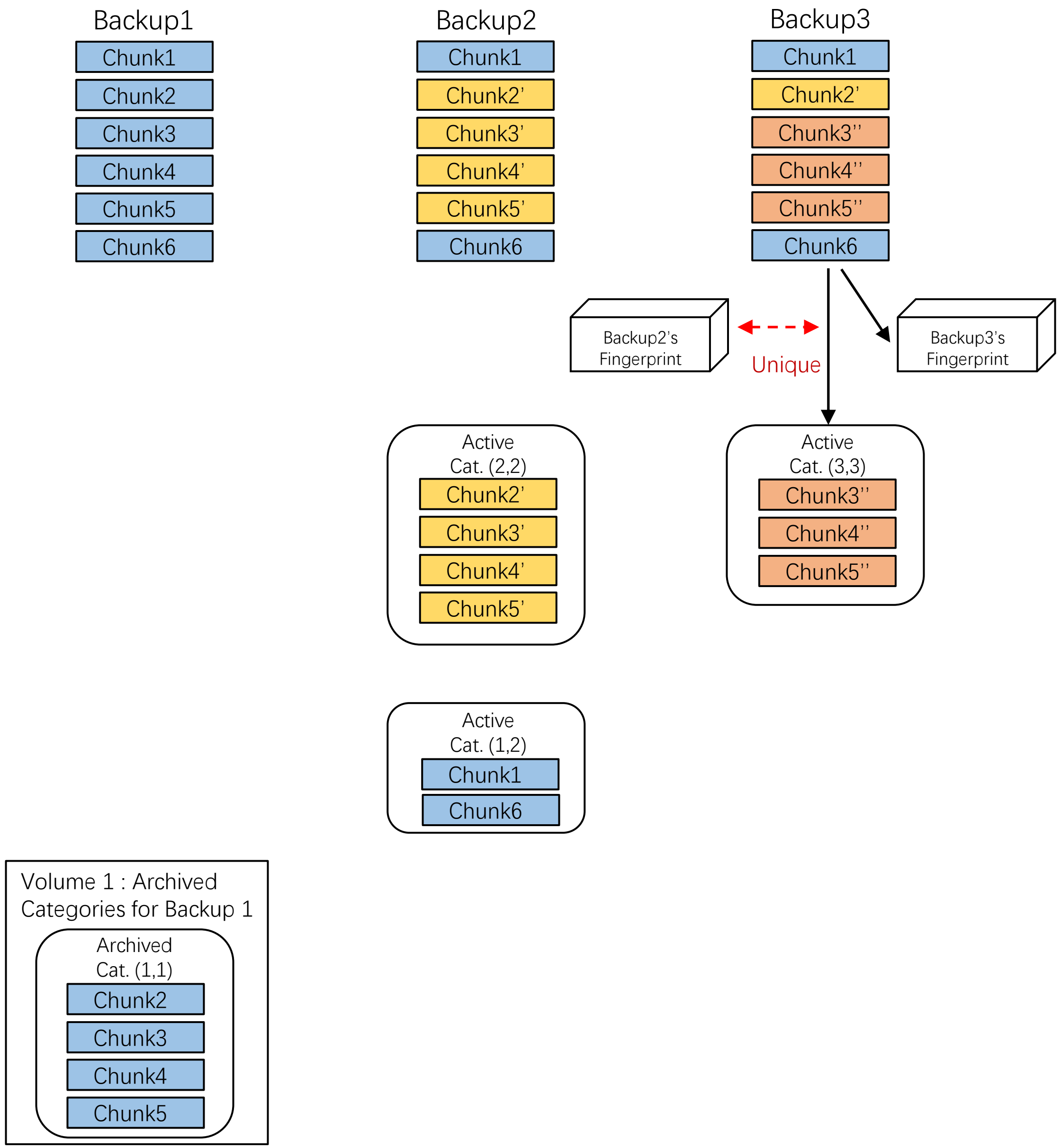
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



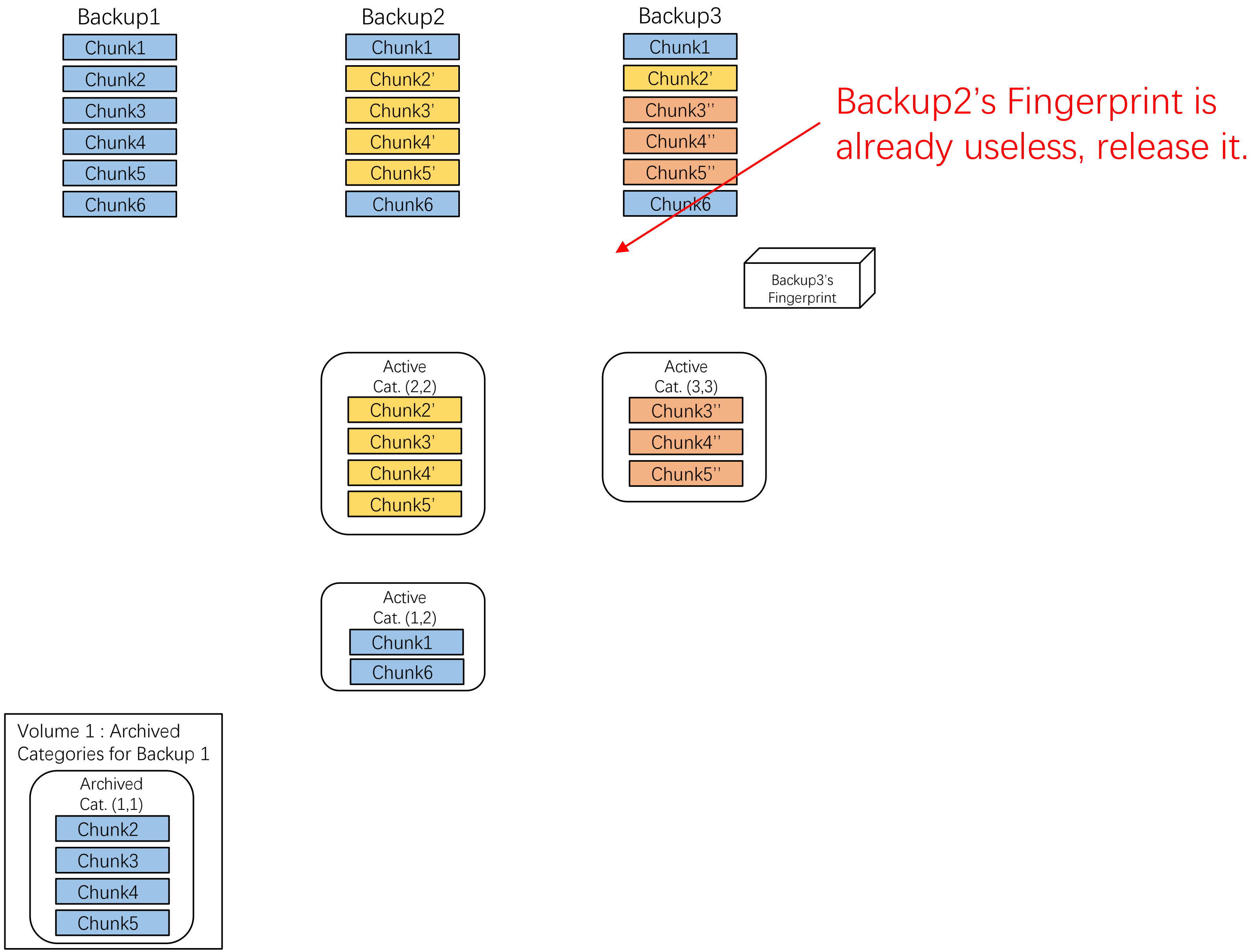
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



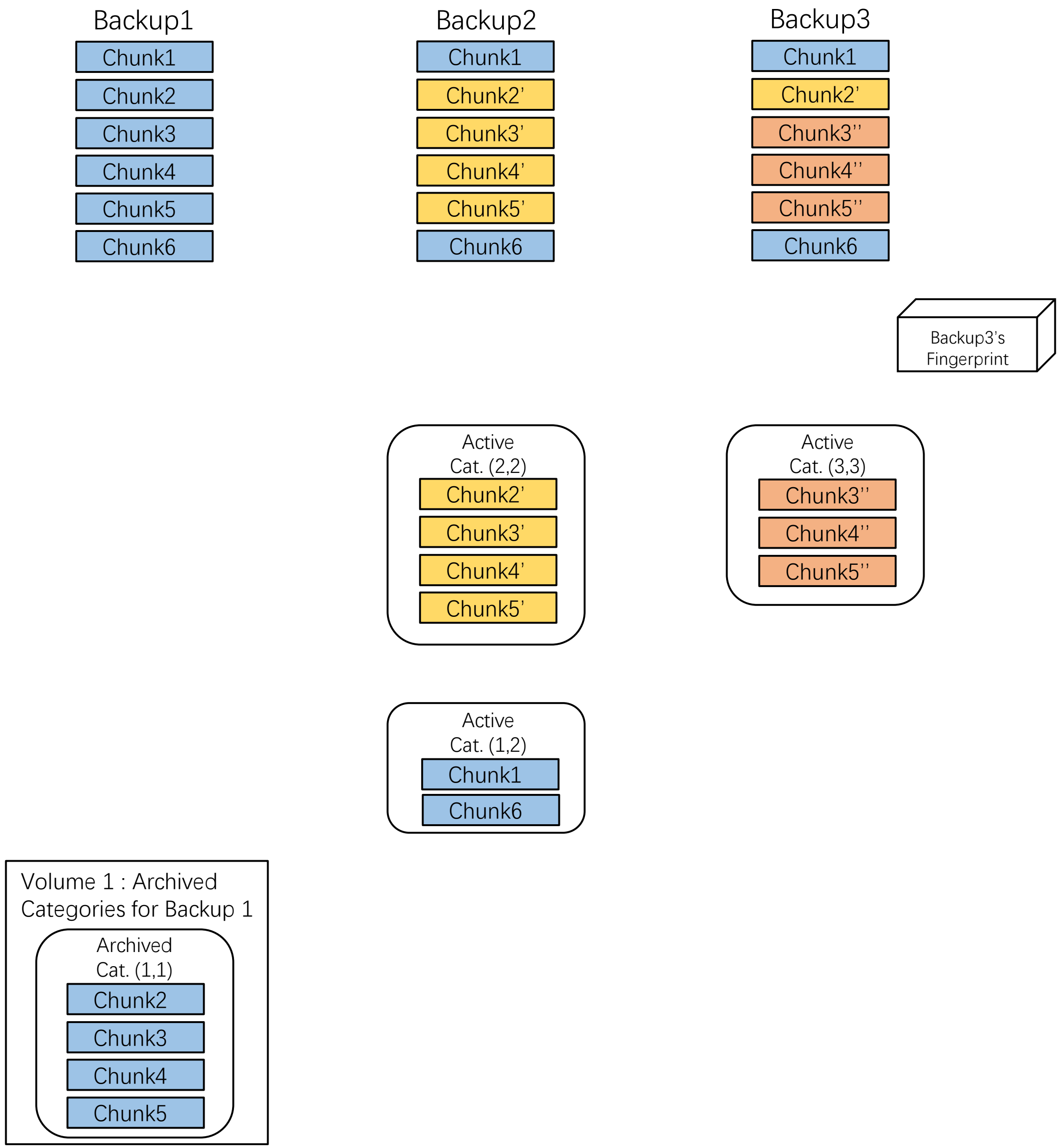
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



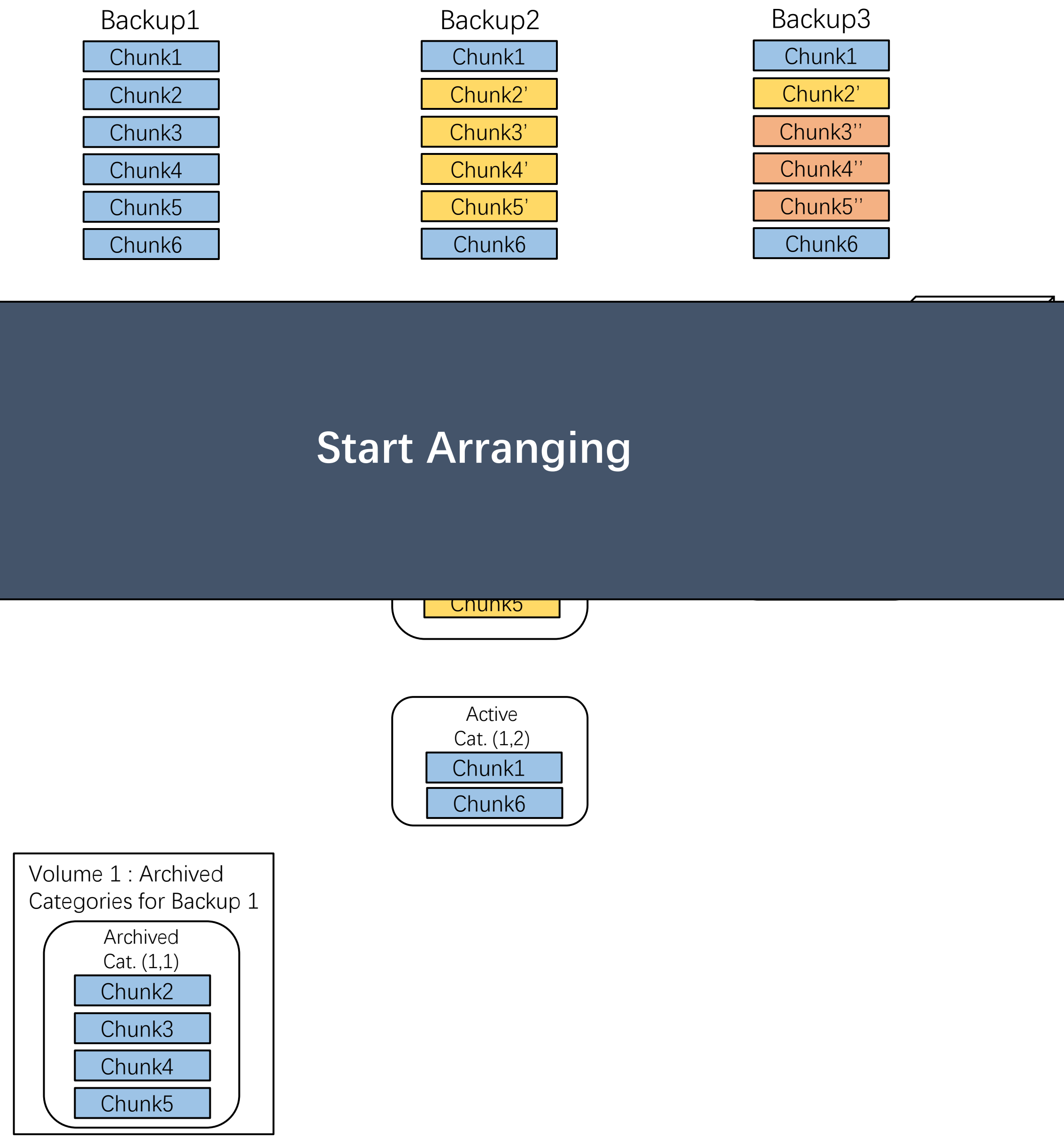
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



Iterative Evolution

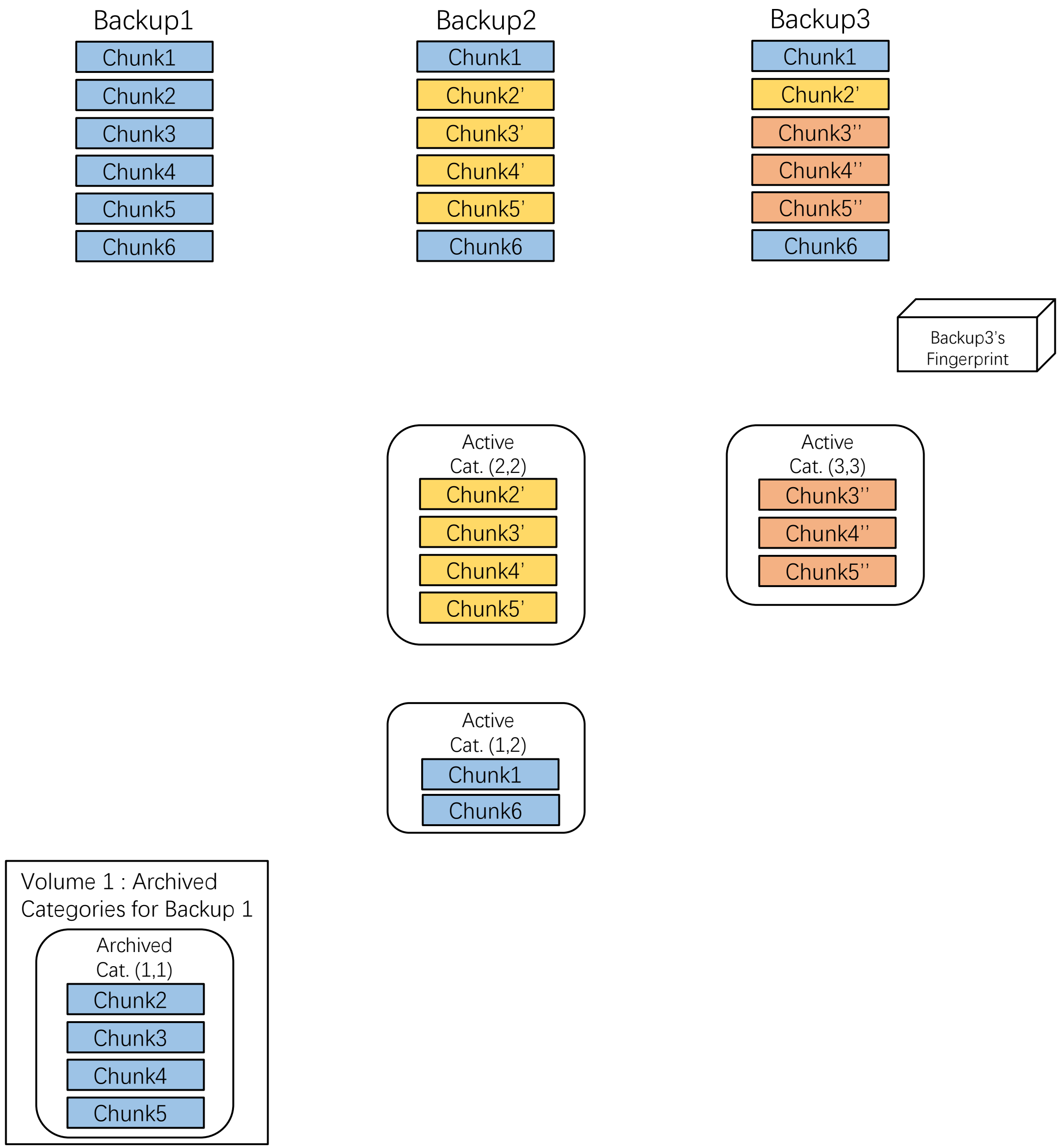
Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



Start Arranging

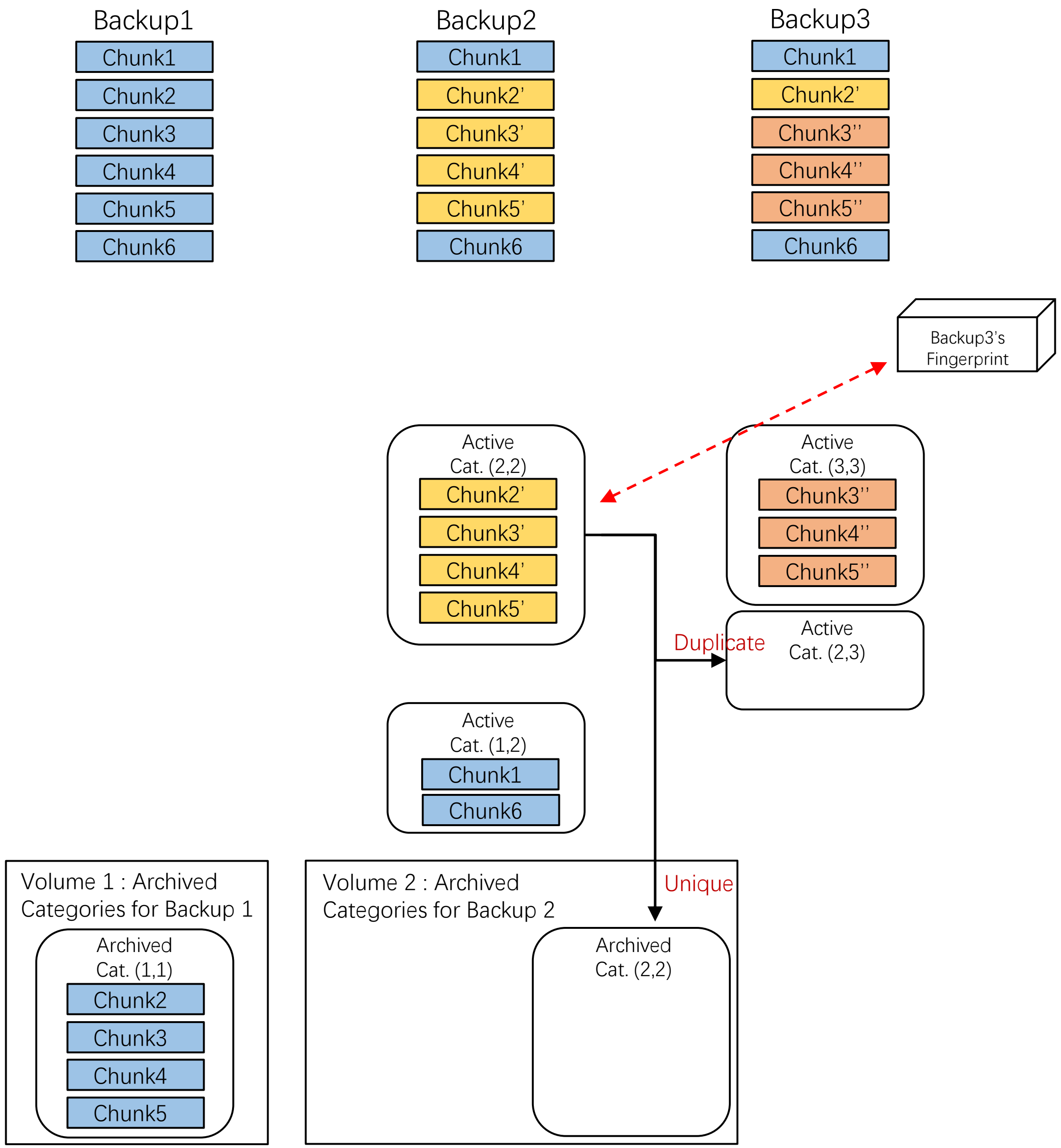
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



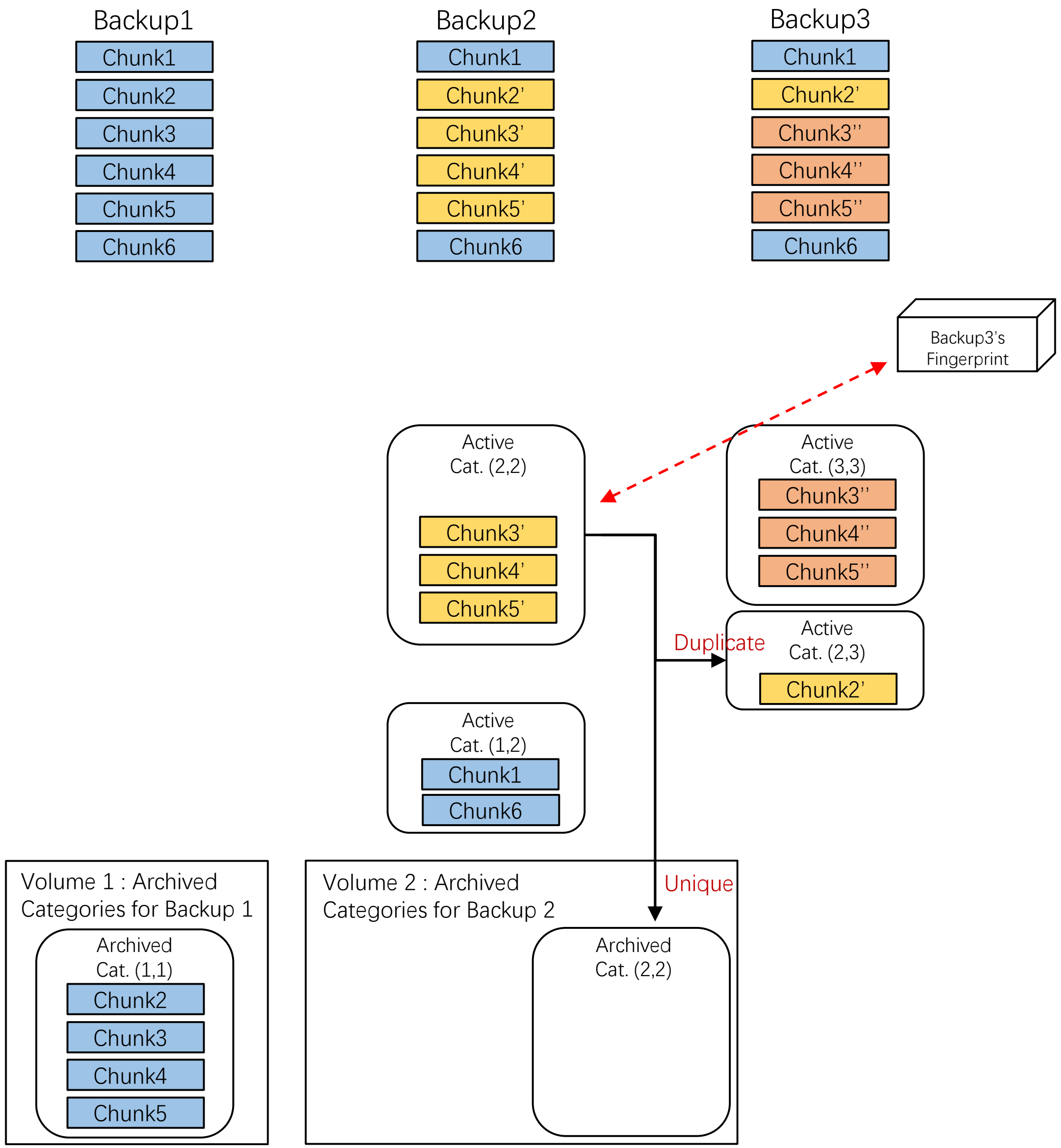
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



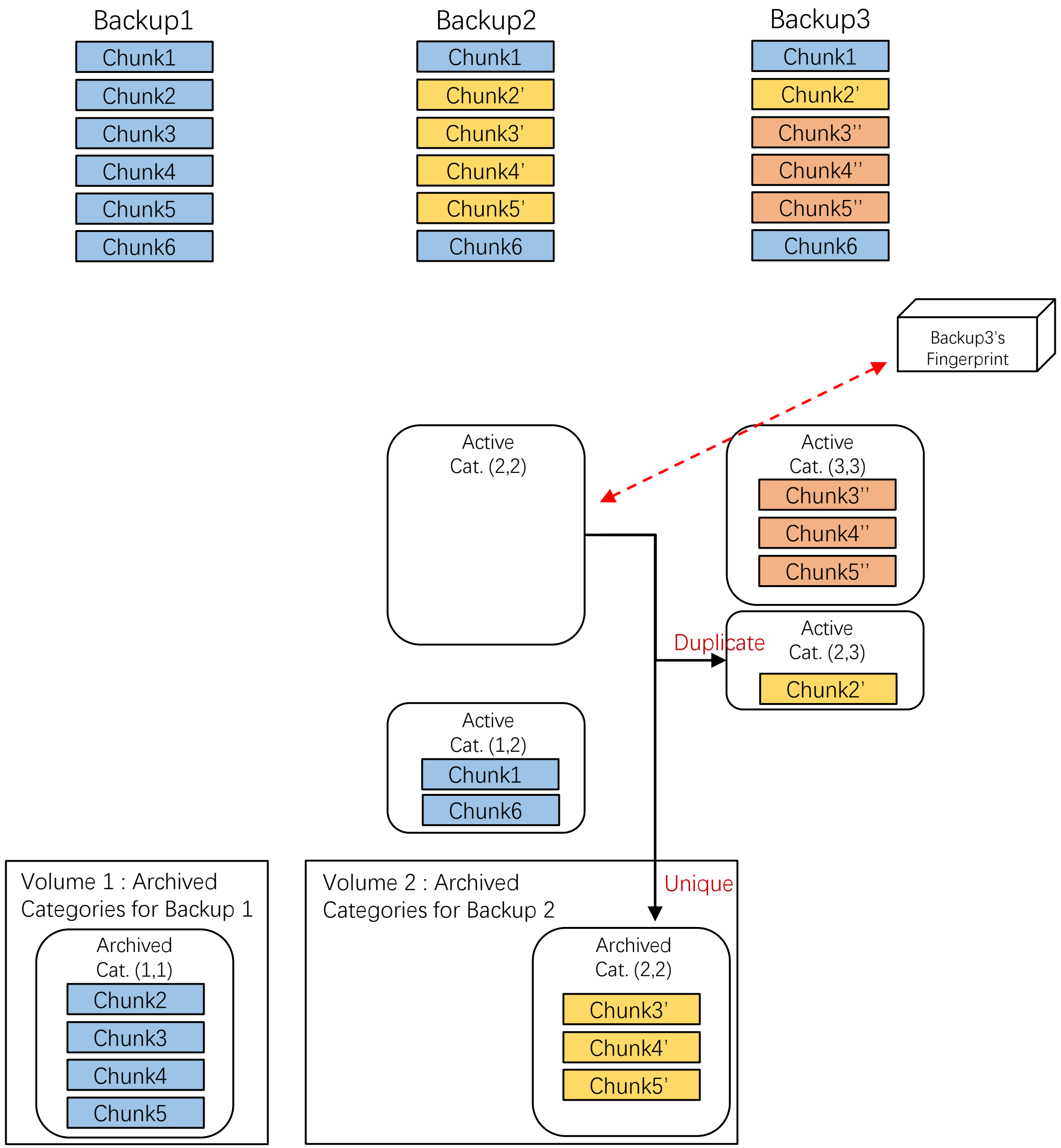
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



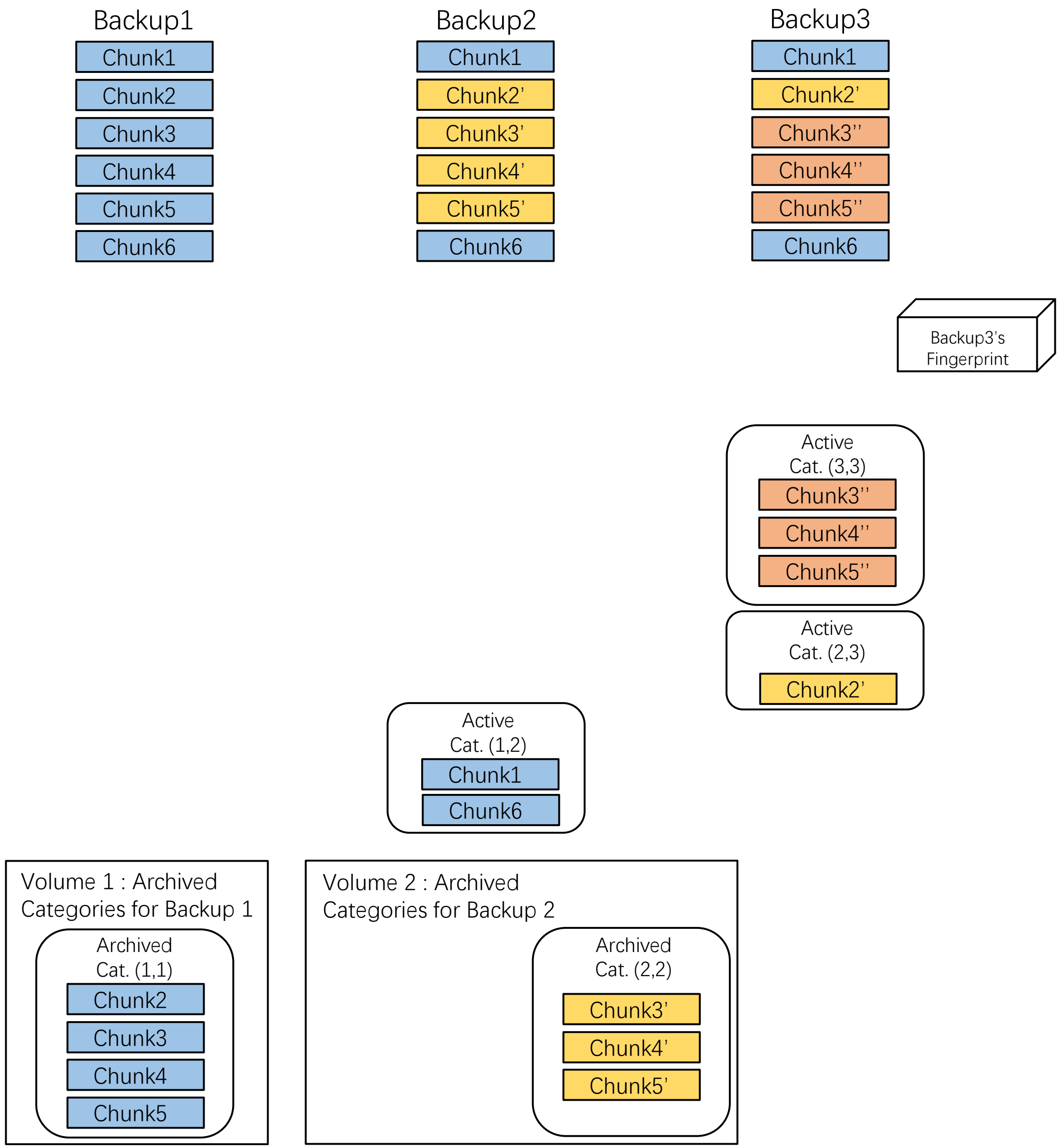
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



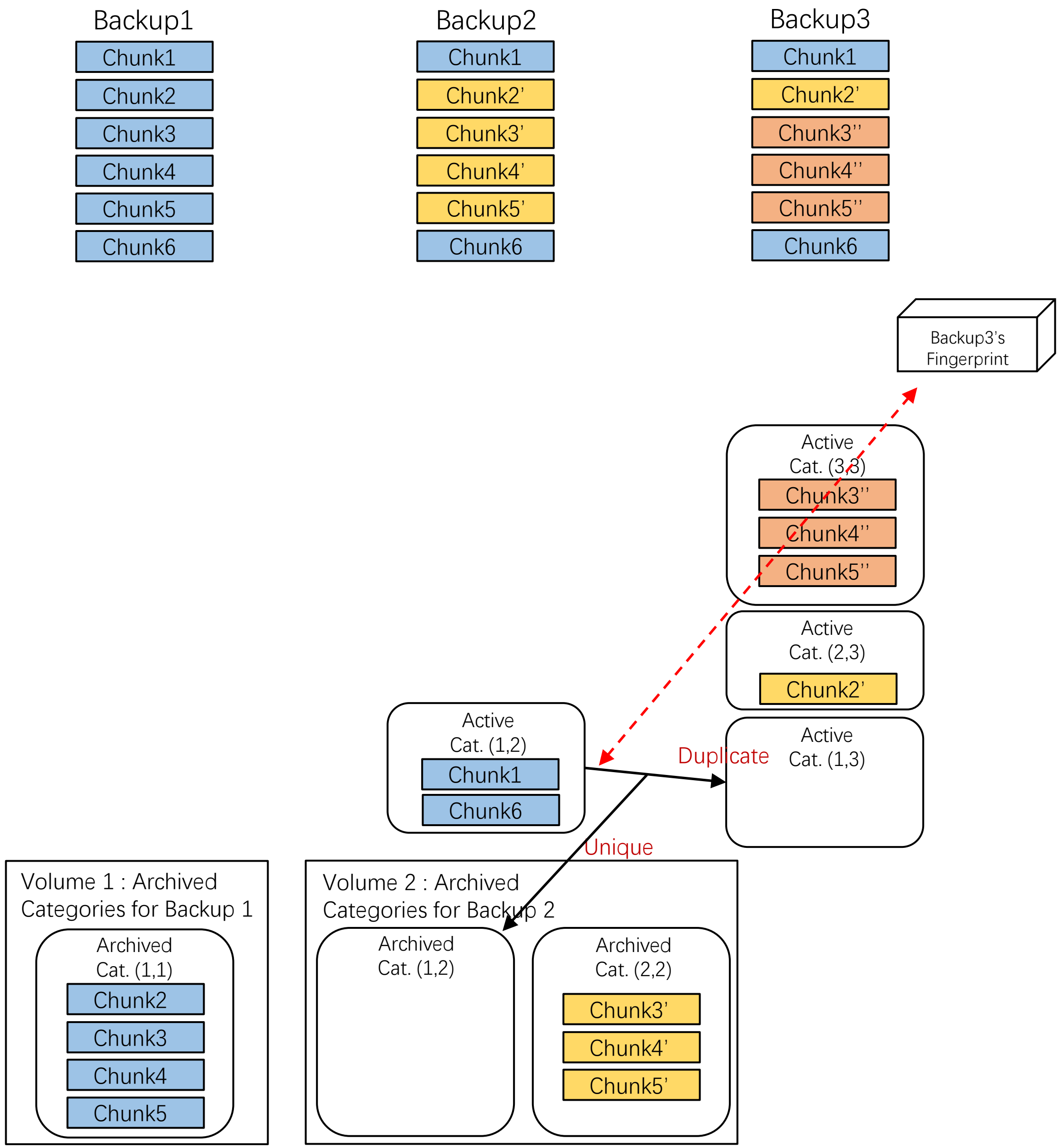
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



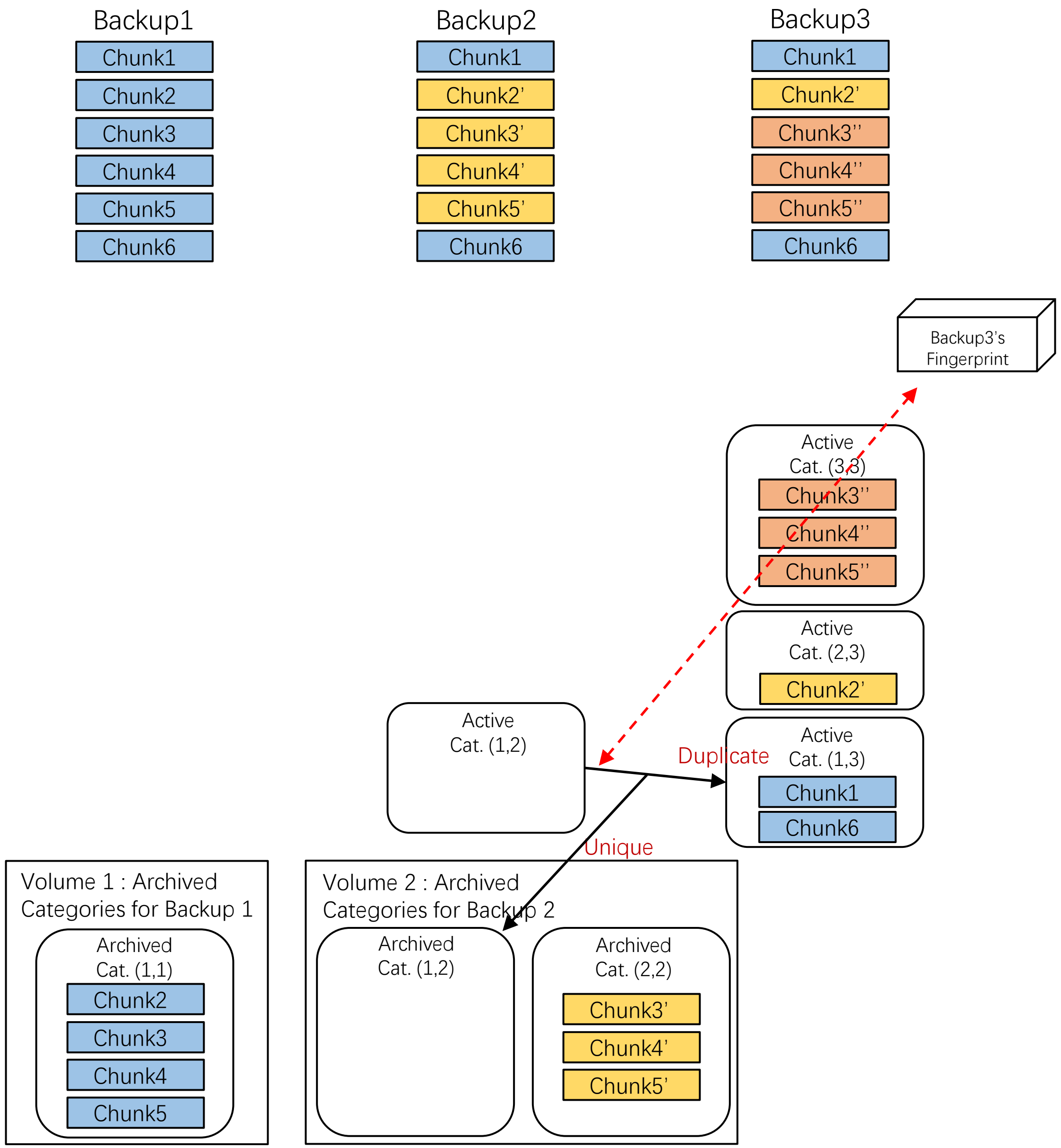
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



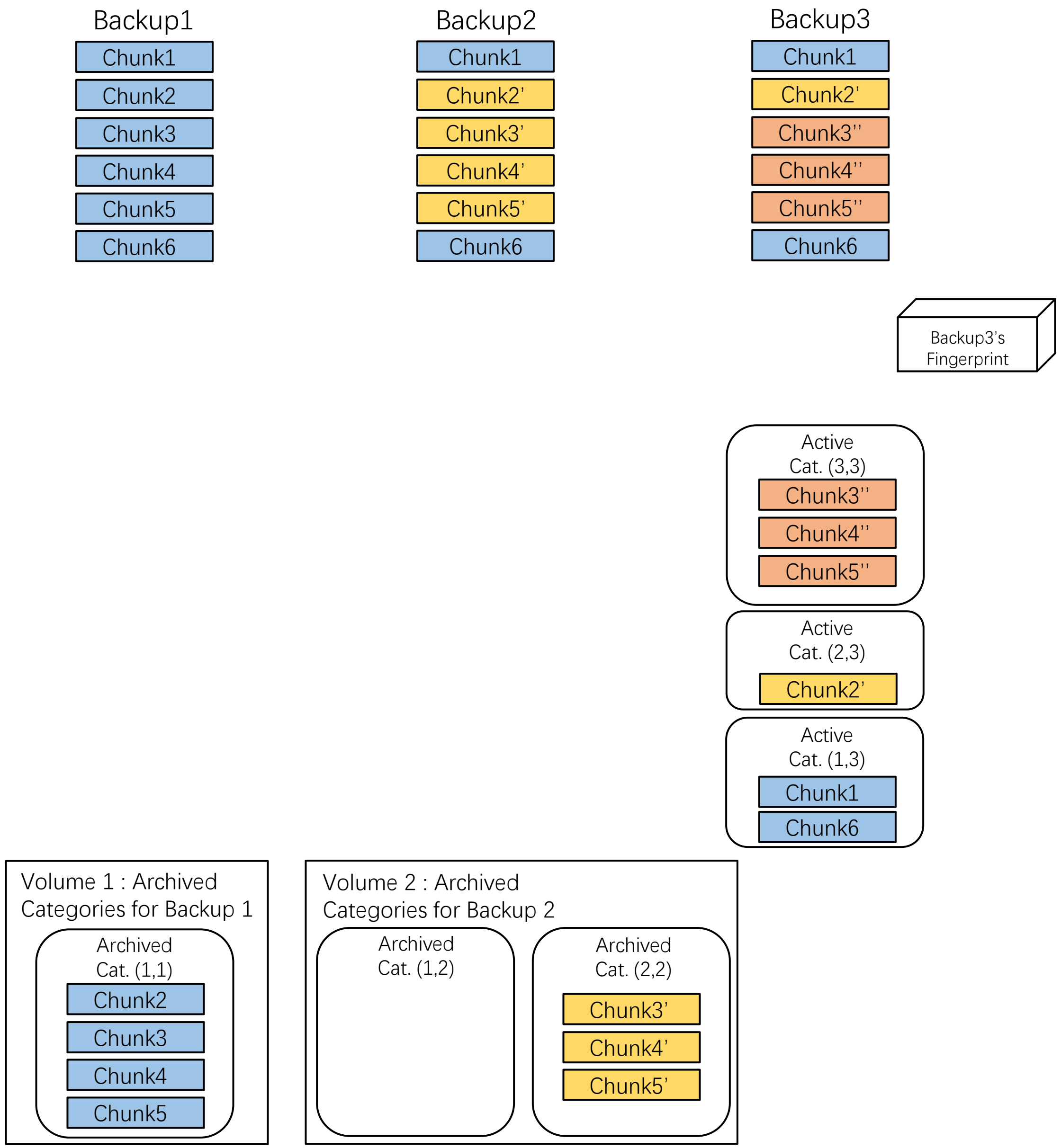
Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj



Iterative Evolution

Cat.(i,j) contains all chunks whose lifecycle is from Bi to Bj

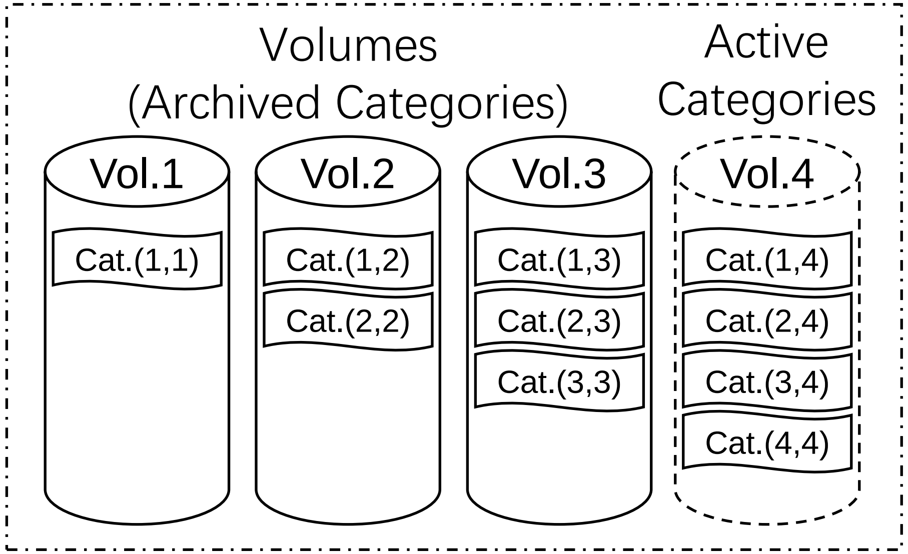


Restore backups

- If n backup versions stored, required categories are always in n sequences.
- Required Cat. = {Cat.(i, j)}, where 1 ≤ i ≤ k ≤ j ≤ n

$$= \cup_{j=k}^{n} \cup_{i=1}^{j} \text{Cat.}(i, j)$$

- No read amplification

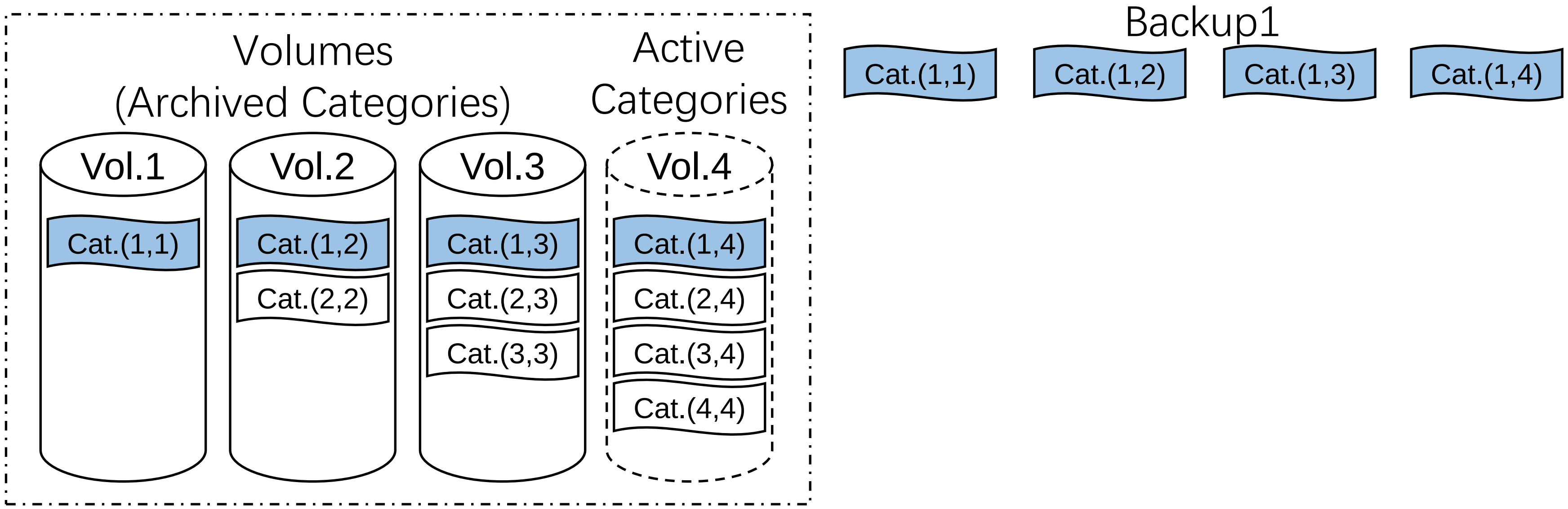


Restore backups

- If n backup versions stored, required categories are always in n sequences.
- Required Cat. = {Cat.(i, j)}, where 1 ≤ i ≤ k ≤ j ≤ n

$$= \cup_{j=k}^{n} \cup_{i=1}^{j} \text{Cat.}(i, j)$$

- No read amplification

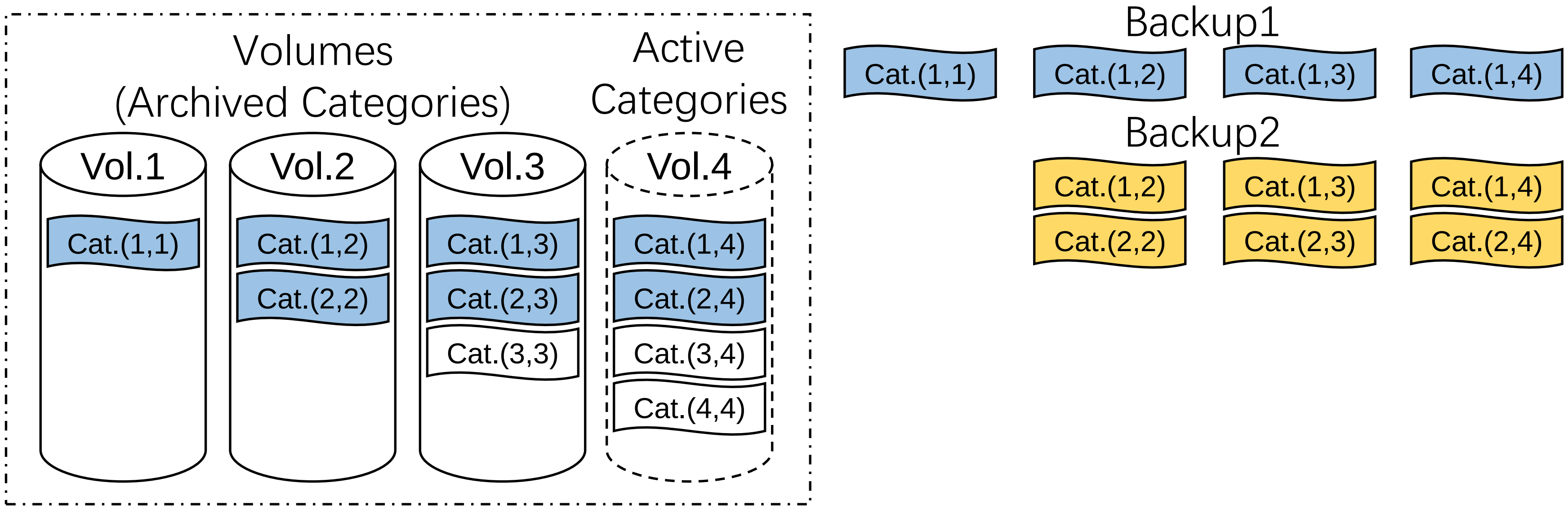


Restore backups

- If n backup versions stored, required categories are always in n sequences.
- Required Cat. = {Cat.(i, j)}, where 1 ≤ i ≤ k ≤ j ≤ n

$$= \cup_{j=k}^{n} \cup_{i=1}^{j} \text{Cat.}(i, j)$$

- No read amplification

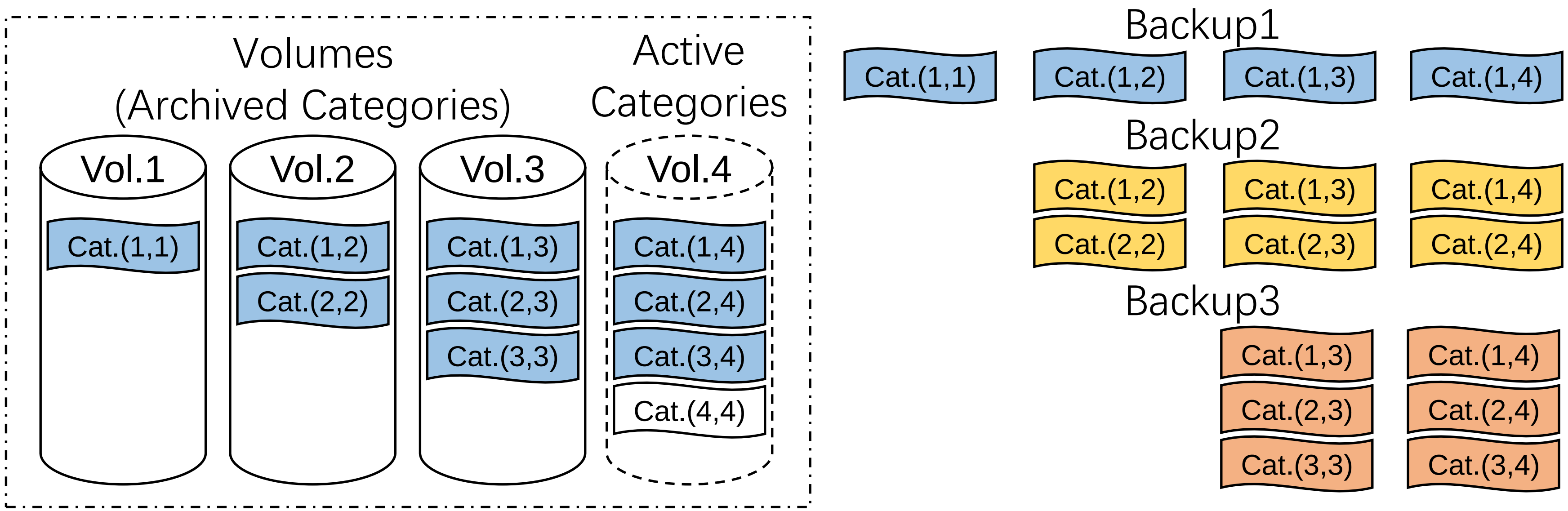


Restore backups

- If n backup versions stored, required categories are always in n sequences.
- Required Cat. = {Cat.(i, j)}, where 1 ≤ i ≤ k ≤ j ≤ n

$$= \cup_{j=k}^{n} \cup_{i=1}^{j} \text{Cat.}(i, j)$$

- No read amplification

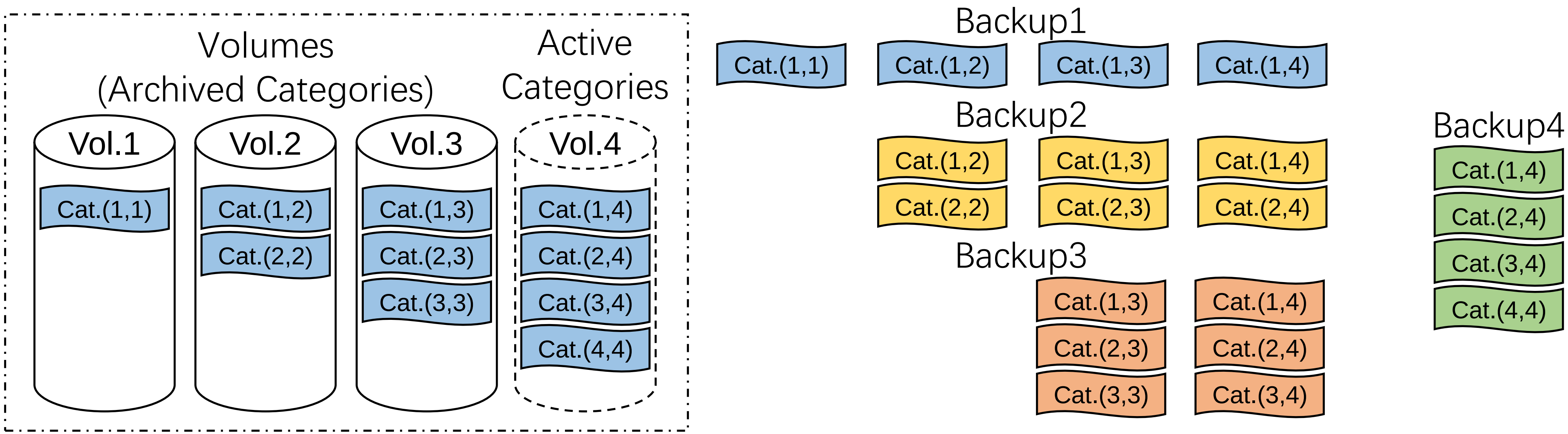


Restore backups

- If n backup versions stored, required categories are always in n sequences.
- Required Cat. = {Cat.(i, j)}, where 1 ≤ i ≤ k ≤ j ≤ n

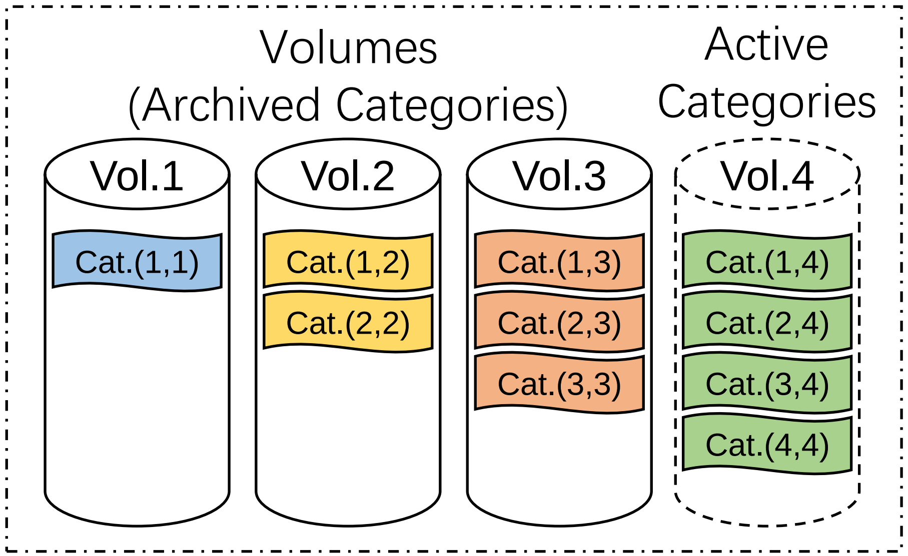
$$= \cup_{j=k}^{n} \cup_{i=1}^{j} \text{Cat.}(i, j)$$

- No read amplification



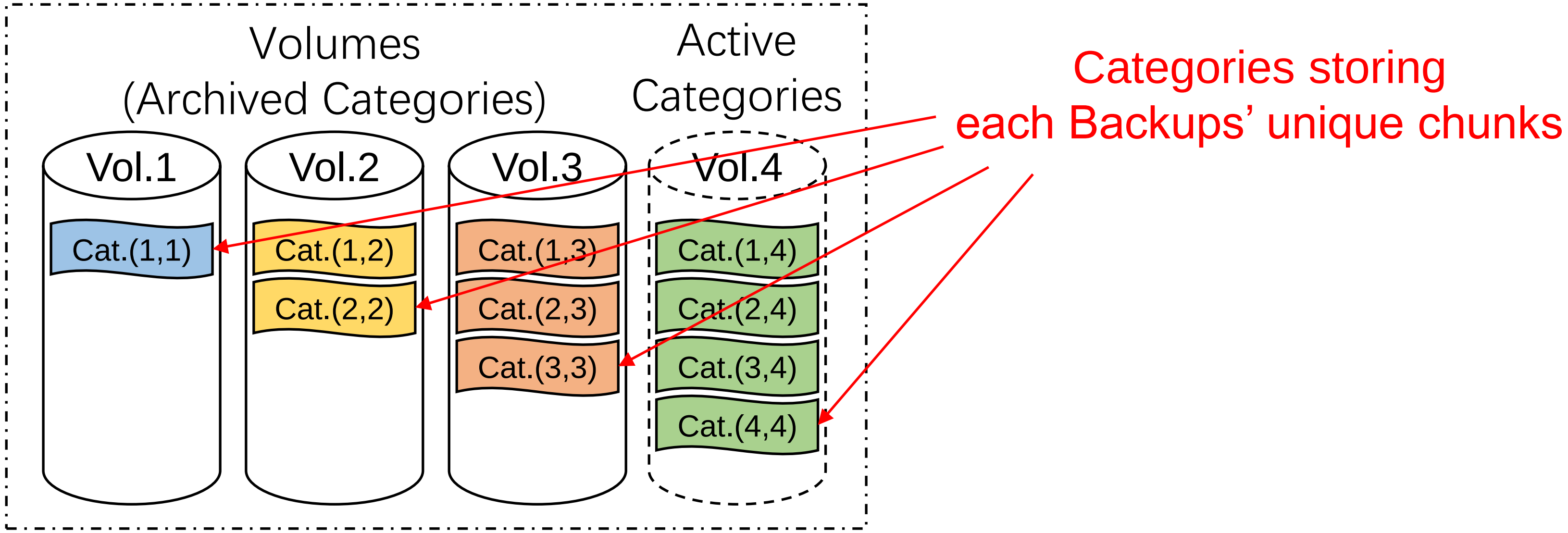
Deletion

- Deleting Backup k means reclaiming all unique chunks of Backup k



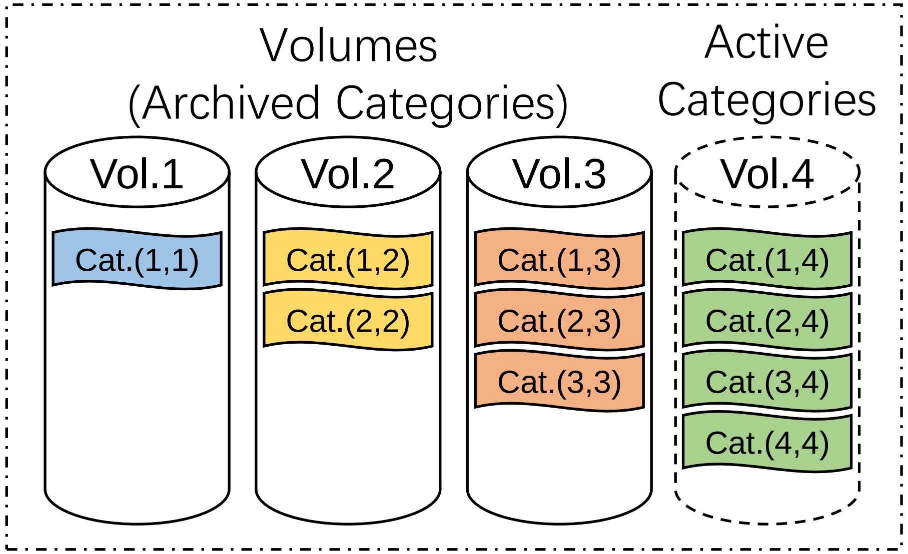
Deletion

- Deleting Backup k means reclaiming all unique chunks of Backup k



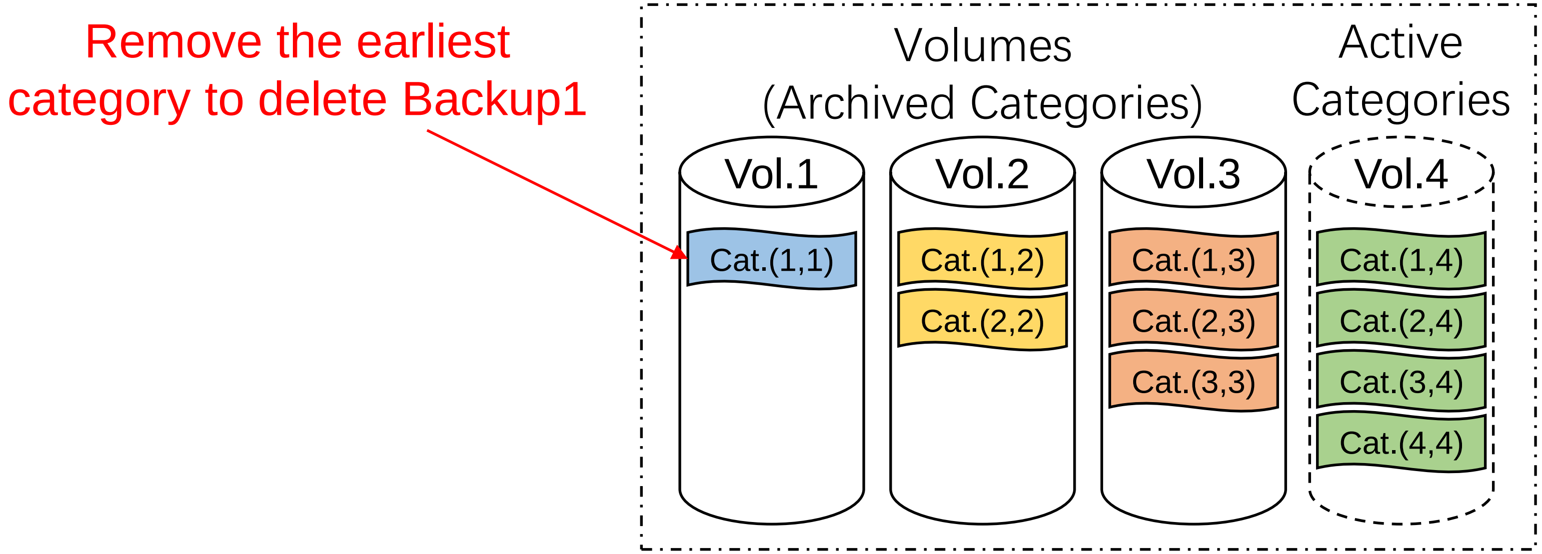
Deletion

- Deleting Backup k means reclaiming all unique chunks of Backup k
- FIFO deletion: simply delete the earliest category



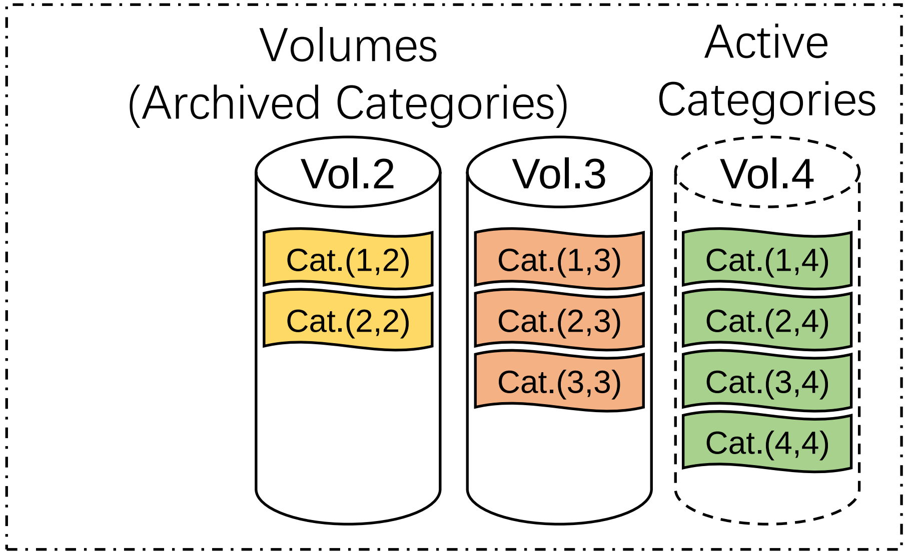
Deletion

- Deleting Backup k means reclaiming all unique chunks of Backup k
- FIFO deletion: simply delete the earliest category



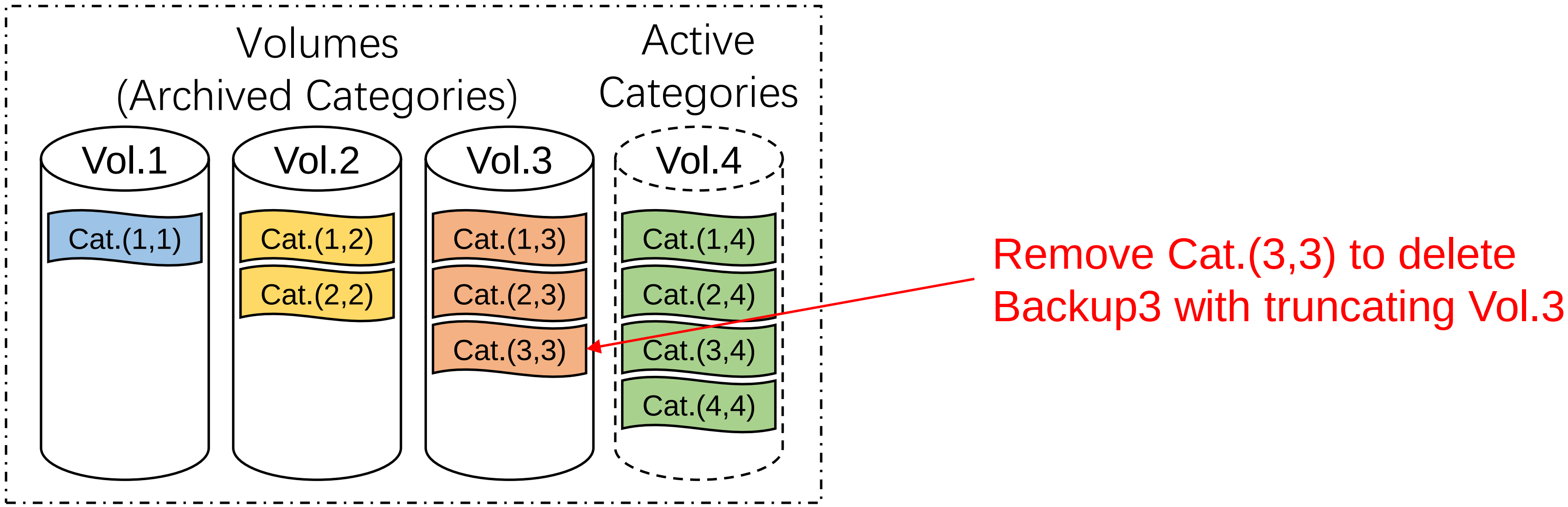
Deletion

- Deleting Backup k means reclaiming all unique chunks of Backup k
- FIFO deletion: simply delete the earliest category



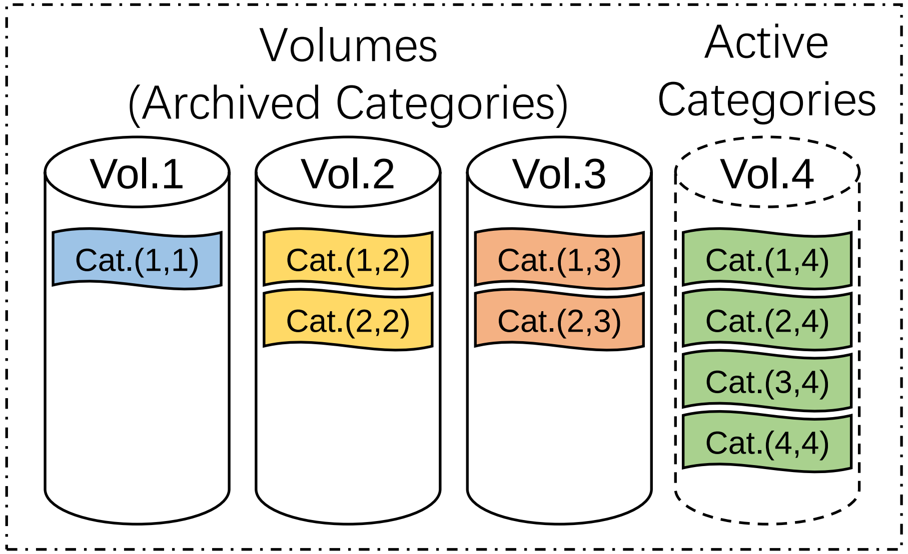
Deletion

- Deleting Backup k means reclaiming all unique chunks of Backup k
- FIFO deletion: simply delete the earliest category
- Out-of-order deletion
  - Truncating corresponding Volume to remove the category which storing unique chunks of Backup *k*



Deletion

- Deleting Backup k means reclaiming all unique chunks of Backup k
- FIFO deletion: simply delete the earliest category
- Out-of-order deletion
  - Truncating corresponding Volume to remove the category which storing unique chunks of Backup *k*



Evaluations

- Storage is divided into backup space (HDD) and user space (SSD).
- Tested datasets are backed up from the user space to the backup space version by version while the restore runs in the reverse direction.
- Retaining the most recent 20 versions.

Table 1: Four backup datasets used in evaluation.

Name	Total Size Before Dedup	Versions	Workload Descriptions
WEB	269 GB	100	Backup snapshots of website: news.sina.com, captured from June to September in 2016.
CHM	279 GB	100	Source codes of Chromium project from v82.0.4066 to v85.0.4165
VMS	1.55 TB	100	Backups of an Ubuntu 12.04 Virtual Machine
SYN	1.38 TB	200	Synthetic backups by simulating file create/delete/modify operations [36]

Evaluations: Actual Deduplication Ratio

- Actual Deduplication Ratio is defined as $\frac{Total\ Size\ of\ the\ Dataset}{Size\ after\ Running\ an\ Approach}$
- The storage cost of rewriting techniques, not perfect garbage collections, and other issues are considered.
- CMA means the laziest GC strategy, and PGC means the greediest GC strategy, and they give two kinds of extreme impacts of GC.

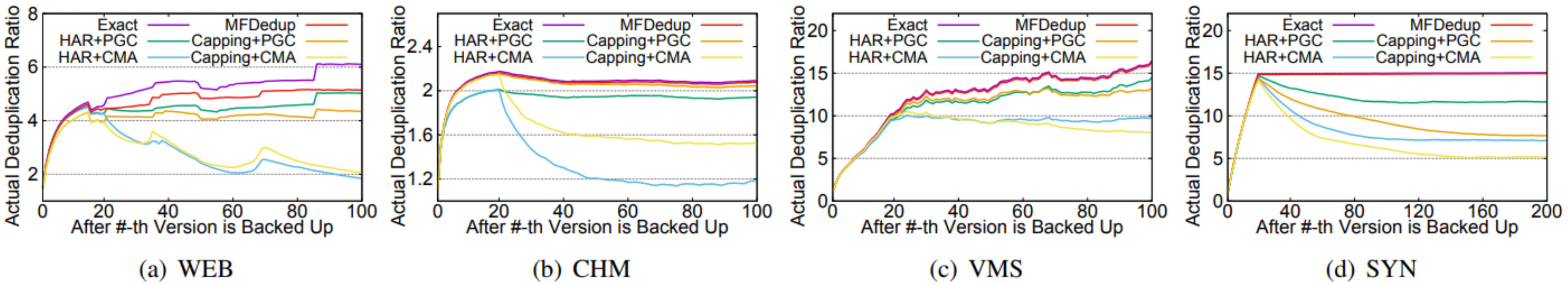
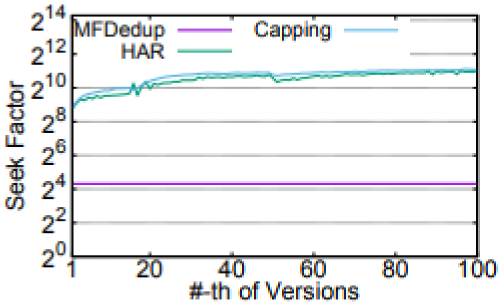


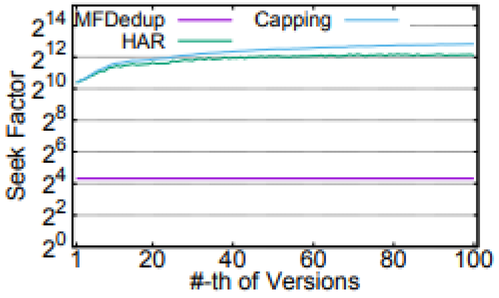
Figure 7: Actual Deduplication Ratio of MFDedup and five approaches running on four datasets (retaining 20 backups).

Evaluations:Restore Performance (Metric)

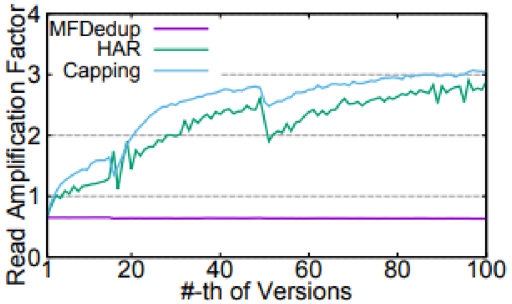
- Speed factor is not feasible, we extend it to two metrics
  - Seek Factor
  - Read Amplification Factor
- MFDedup's Seek Factor is always to be 20.
  - The number of retained backup versions.
- Because of internal duplicate chunks, MFDedup's Read Amplification Factor could be smaller than 1.



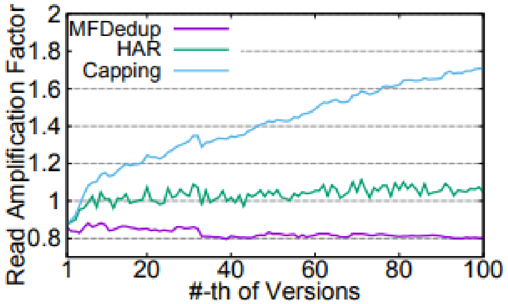
(a) WEB



(b) VMS



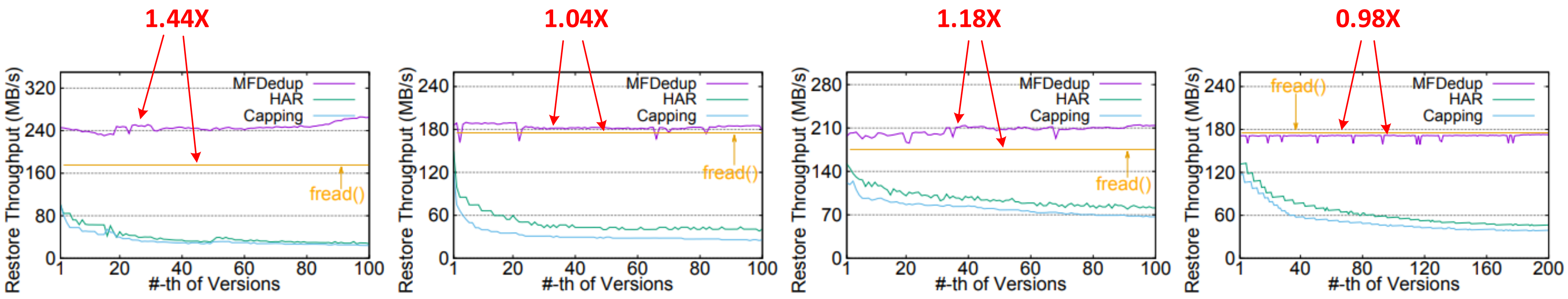
(a) WEB



(b) VMS

Evaluations:Restore Performance(Speed)

- fread() denotes sequential throughput of the backup device.
- According to the share of internal chunks in datasets, MFDedup nearly completely utilize the performance of storage devices



(a) WEB (b) CHM (c) VMS (d) SYN

Q & A

- Thanks for listening
- Our code is available at https://github.com/Borelset/MFDedup/
- Email: xiangyu.zou@hotmail.com