HydraFS:
a High-Throughput File System for the HYDRAstor Content-Addressable Storage System

Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale, Steve Rago, Grzegorz Calkowski, Cezary Dubnicki, Aniruddha Bohra

Feb 26, 2010

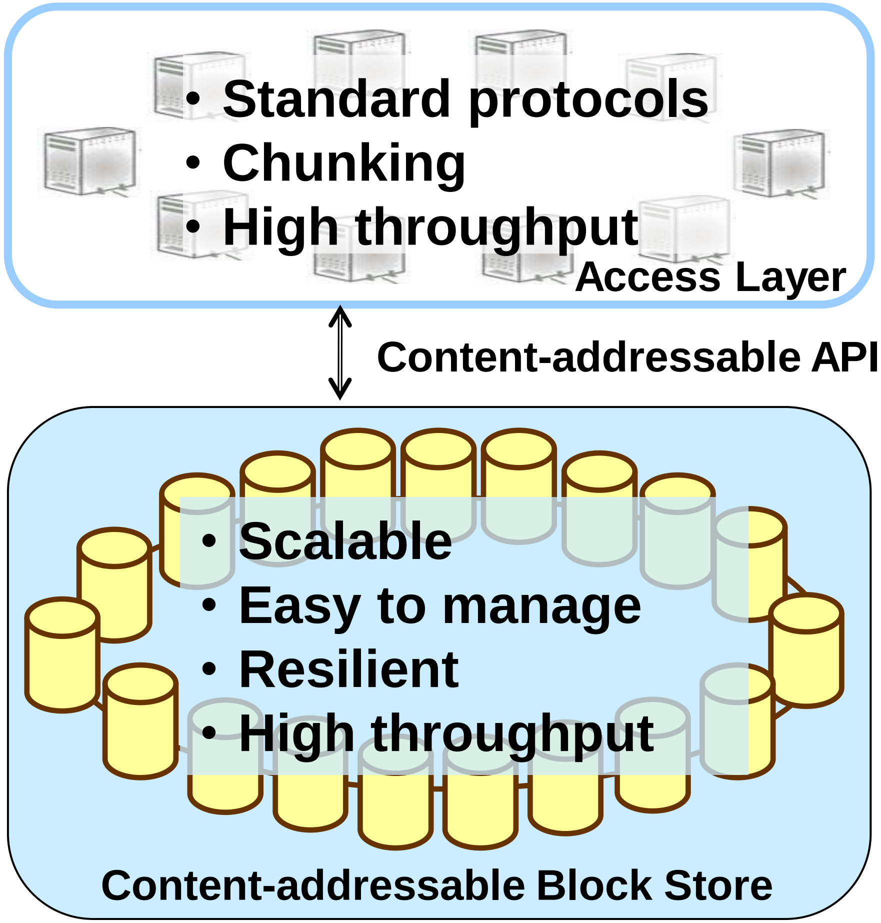
NEC Laboratories America
Relentless passion for innovation

HYDRAstor: De-duplicated Scalable Storage

- Scale-out storage
- With global de-duplication
- Using Content-Defined Chunking
- Resilient to multiple failures
- Easy to manage (self-healing,…)
- High throughput for streaming access
- Std. interfaces (NFS/CIFS, VTL,…)

FAST'10

- *HydraFS: a High Throughput Filesystem*
- *Bimodal CDC for Backup Streams*



- Standard protocols
- Chunking
- High throughput

Access Layer

Content-addressable API

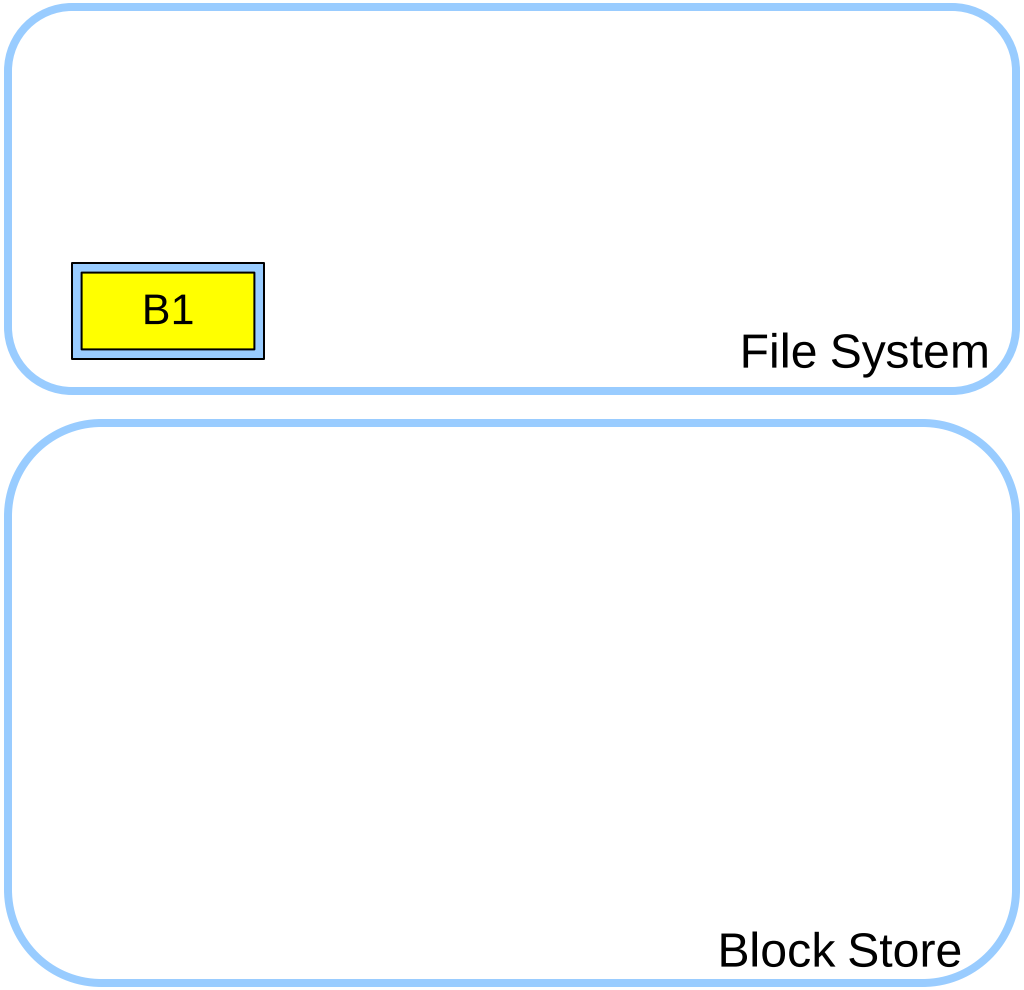
- Scalable
- Easy to manage
- Resilient
- High throughput

Content-addressable Block Store

FAST'09

HYDRAstor: a Scalable Secondary Storage

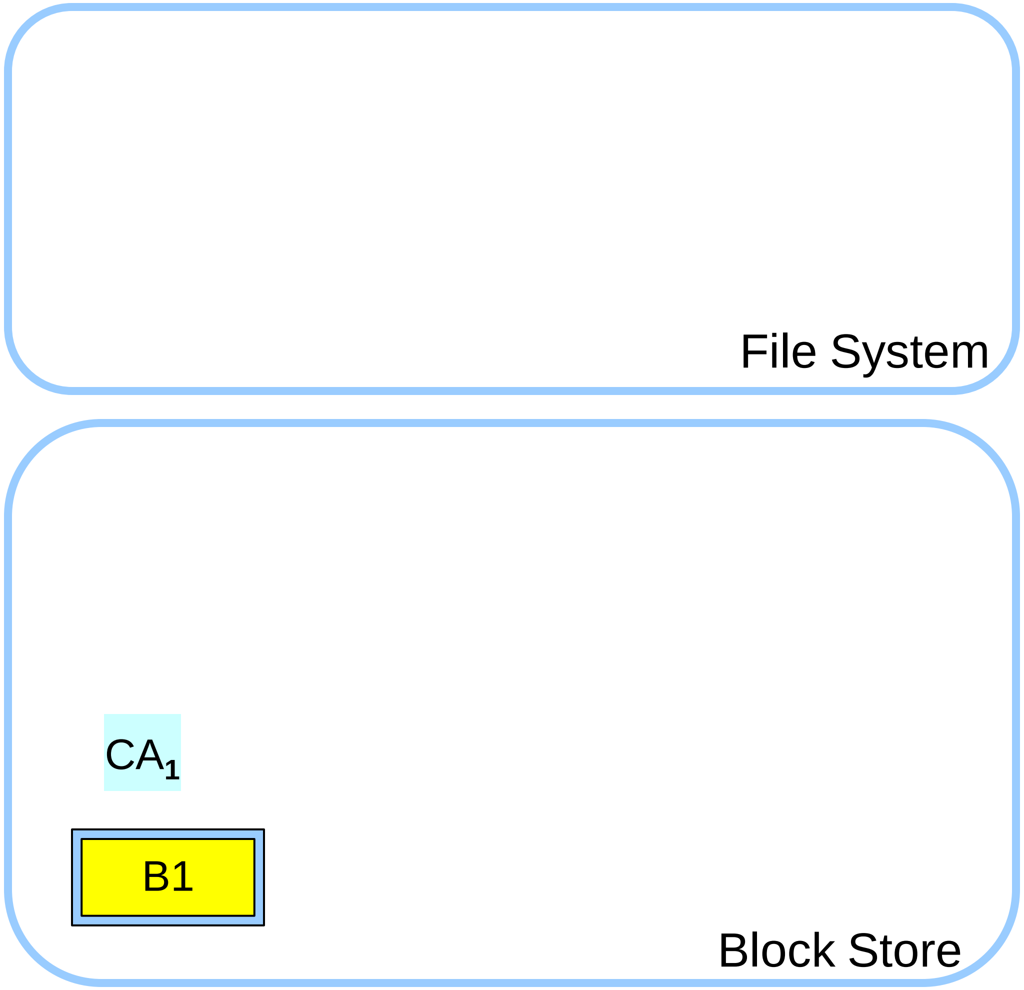
HYDRAstor Usage Example



Block Store (CAS) API

- Variable-size blocks

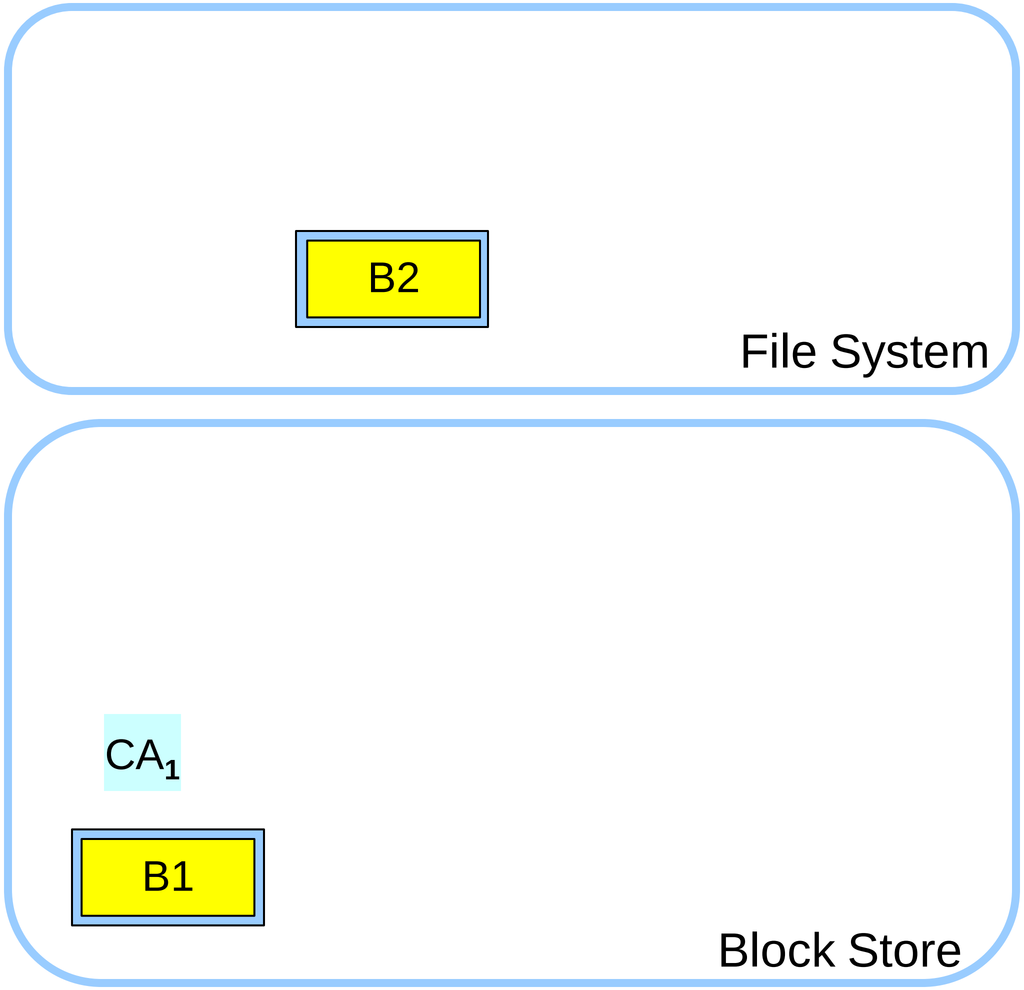
HYDRAstor Usage Example



Block Store (CAS) API

- Variable-size blocks
- Content-addressable
- Address decided by the store

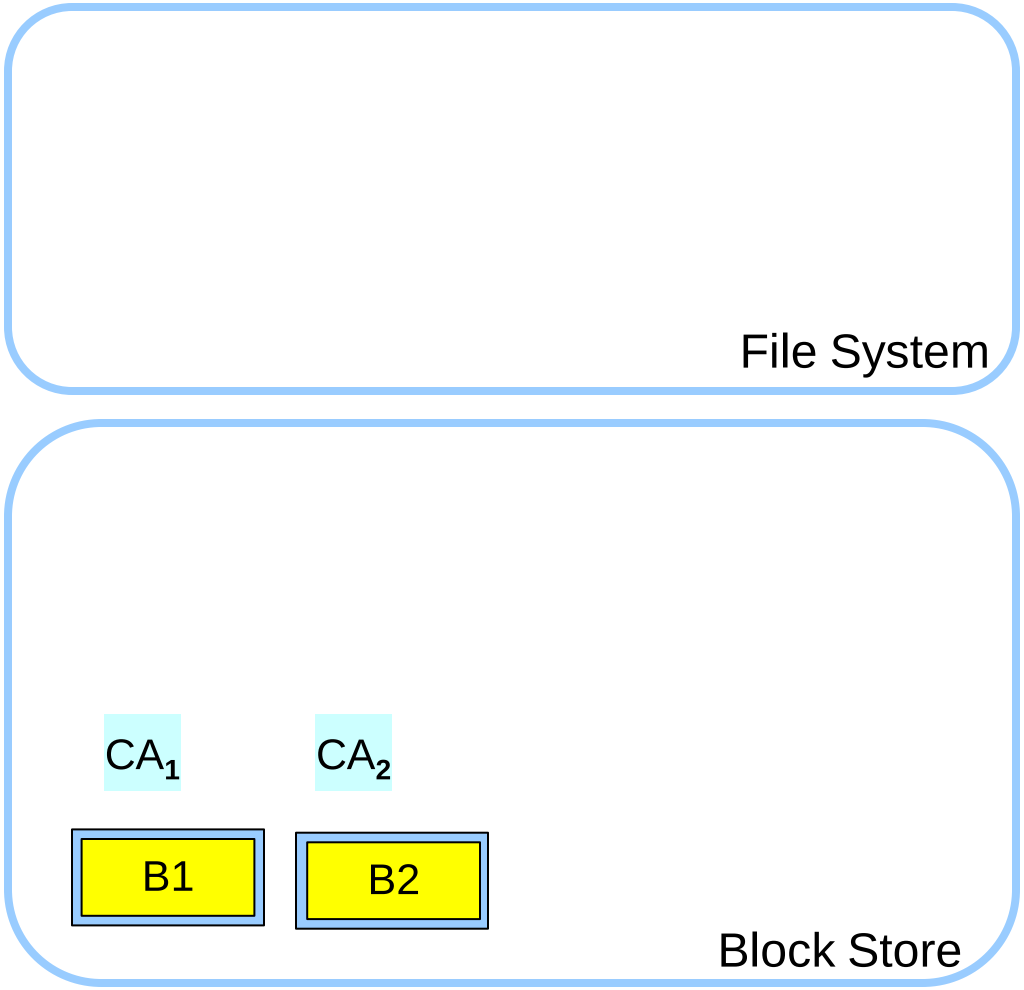
HYDRAstor Usage Example



Block Store (CAS) API

- Variable-size blocks
- Content-addressable
- Address decided by the store

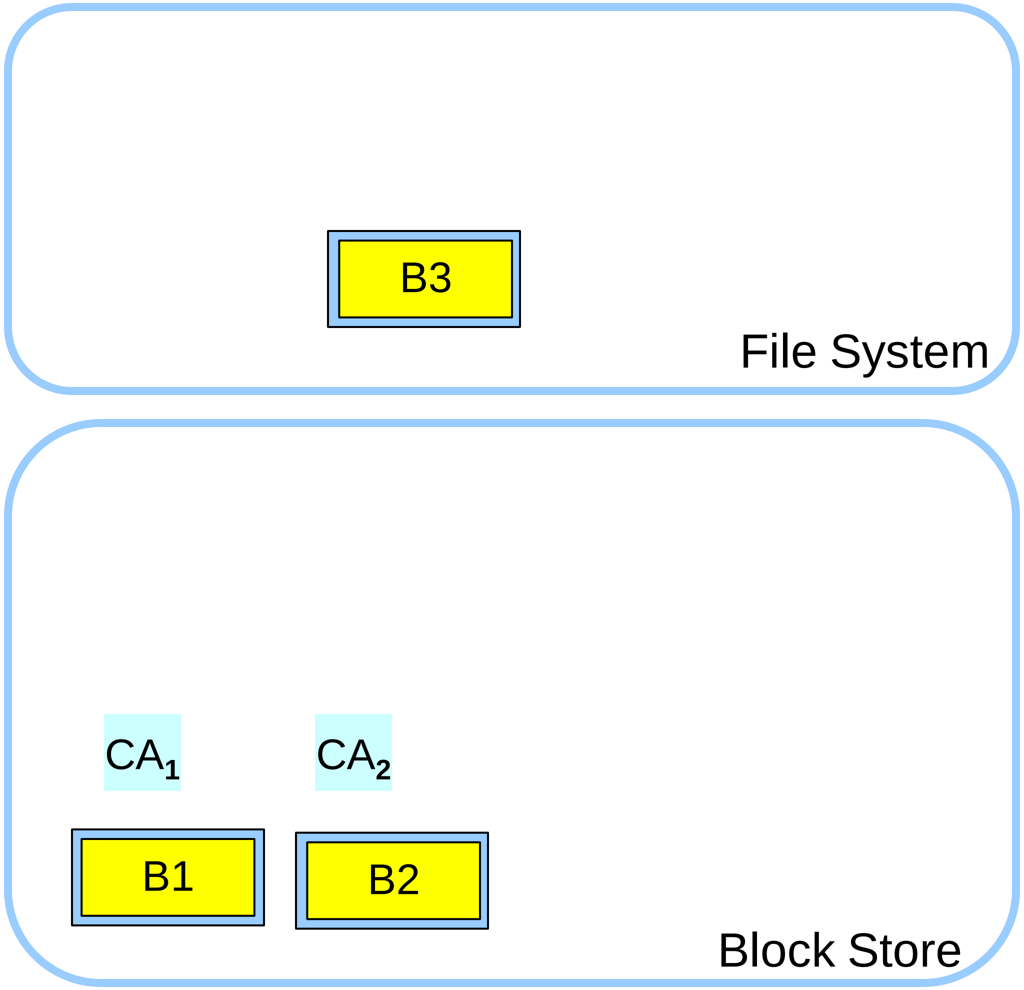
HYDRAstor Usage Example



Block Store (CAS) API

- Variable-size blocks
- Content-addressable
- Address decided by the store

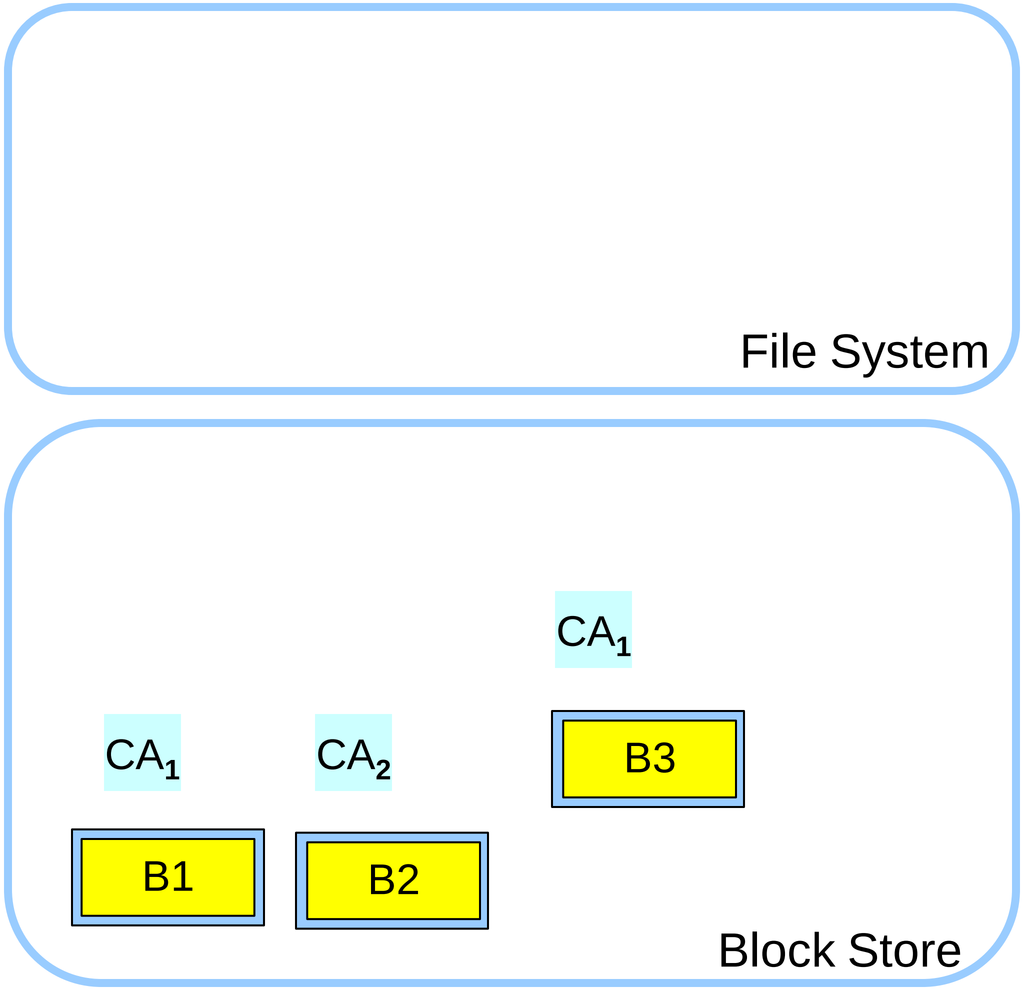
HYDRAstor Usage Example



Block Store (CAS) API

- Variable-size blocks
- Content-addressable
- Address decided by the store

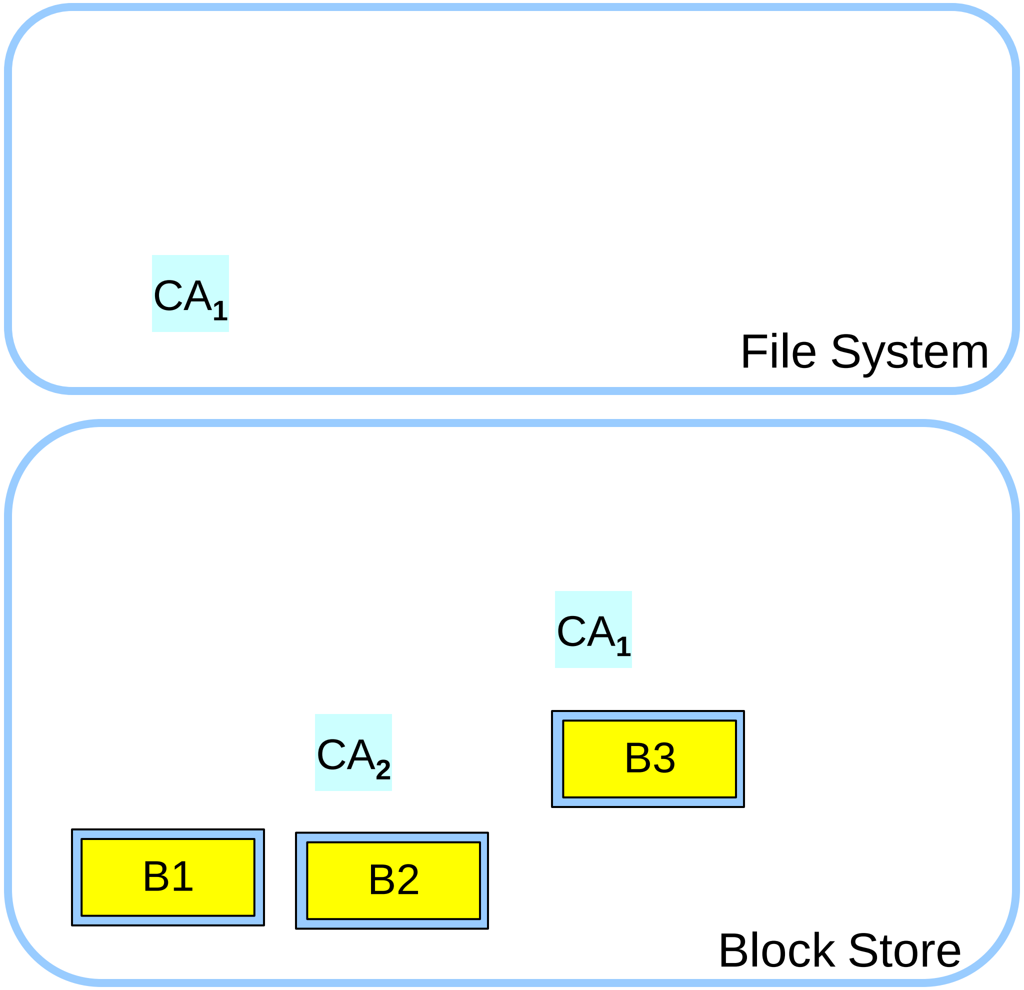
HYDRAstor Usage Example



Block Store (CAS) API

- Variable-size blocks
- Content-addressable
- Address decided by the store
- Duplicates eliminated by store

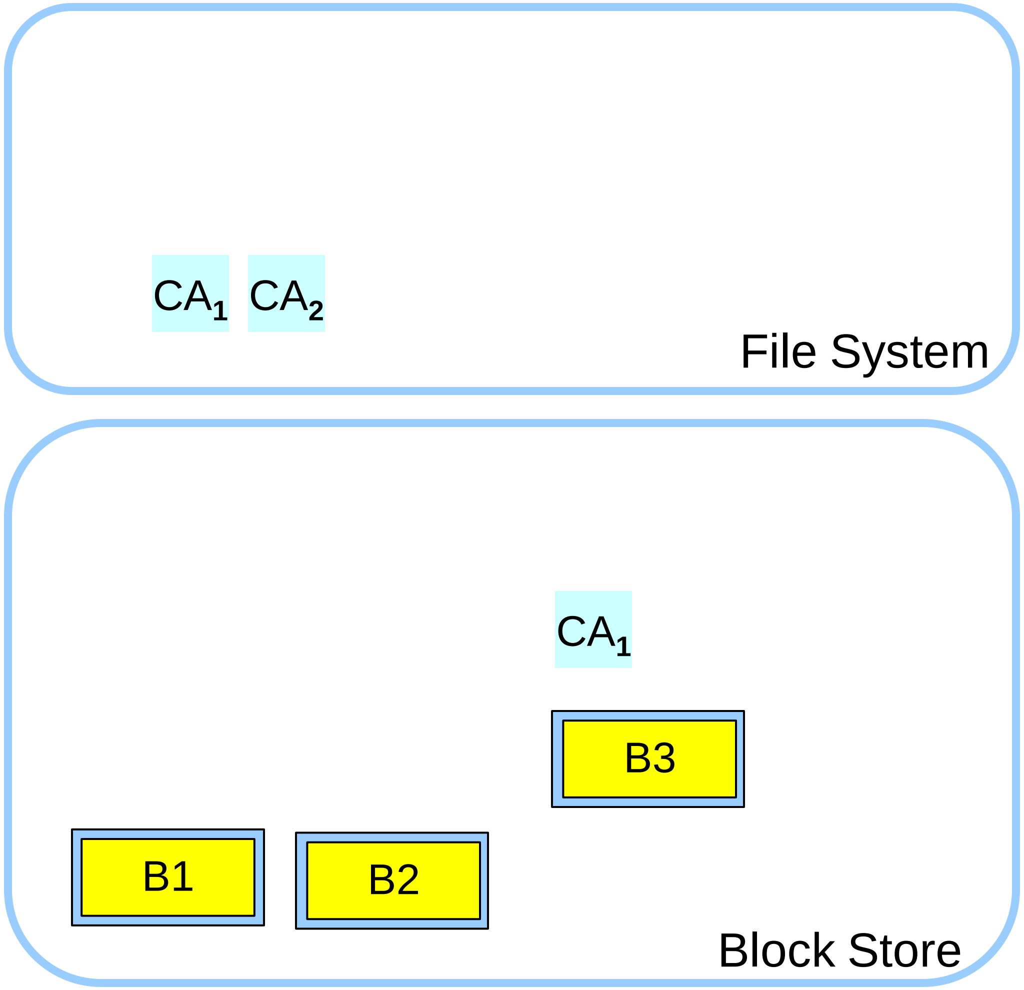
HYDRAstor Usage Example



Block Store (CAS) API

- Variable-size blocks
- Content-addressable
- Address decided by the store
- Duplicates eliminated by store

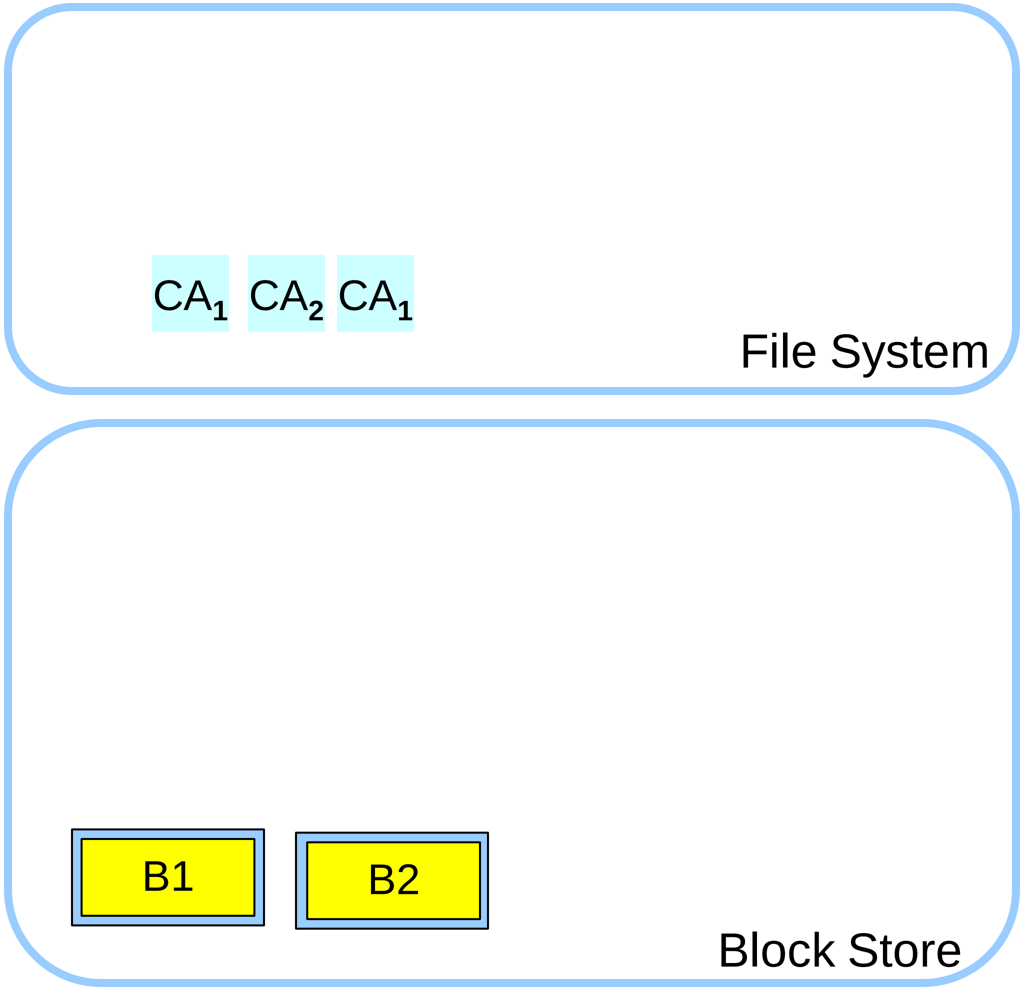
HYDRAstor Usage Example



Block Store (CAS) API

- Variable-size blocks
- Content-addressable
- Address decided by the store
- Duplicates eliminated by store

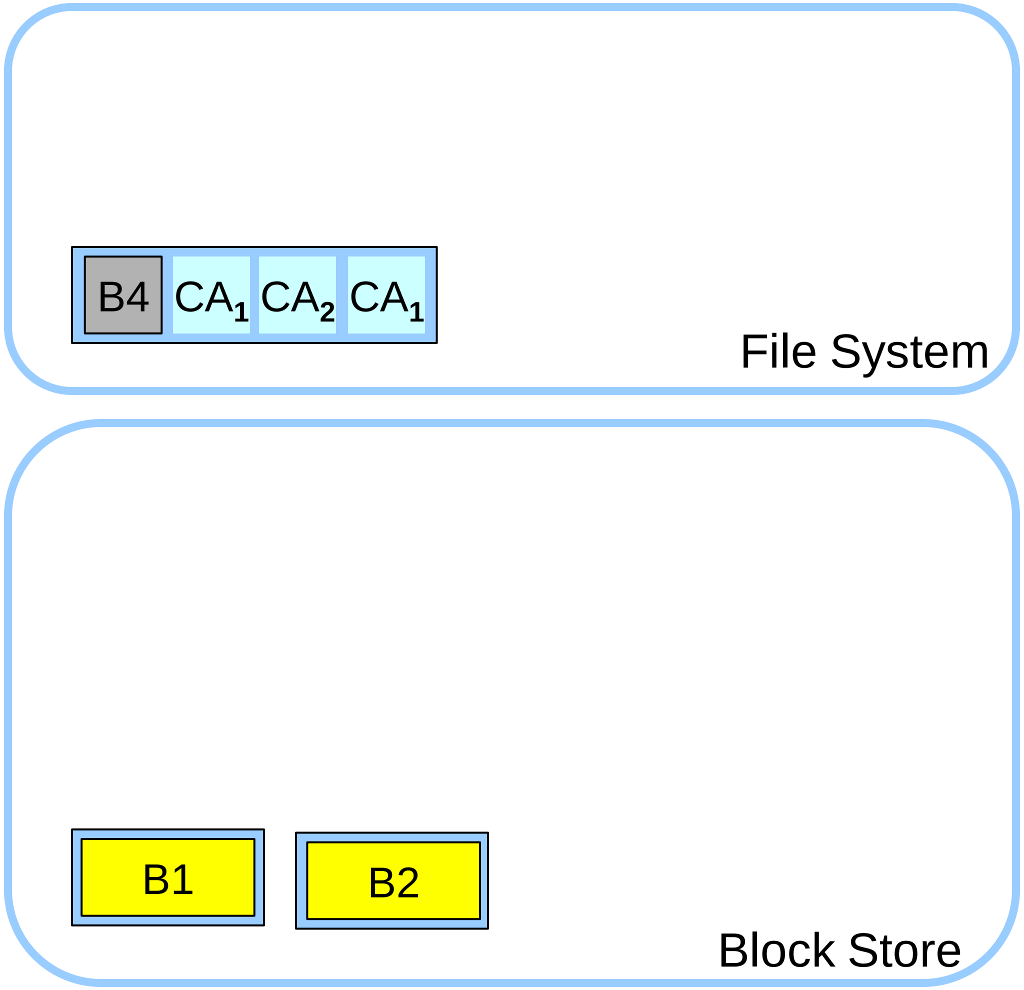
HYDRAstor Usage Example



Block Store (CAS) API

- Variable-size blocks
- Content-addressable
- Address decided by the store
- Duplicates eliminated by store

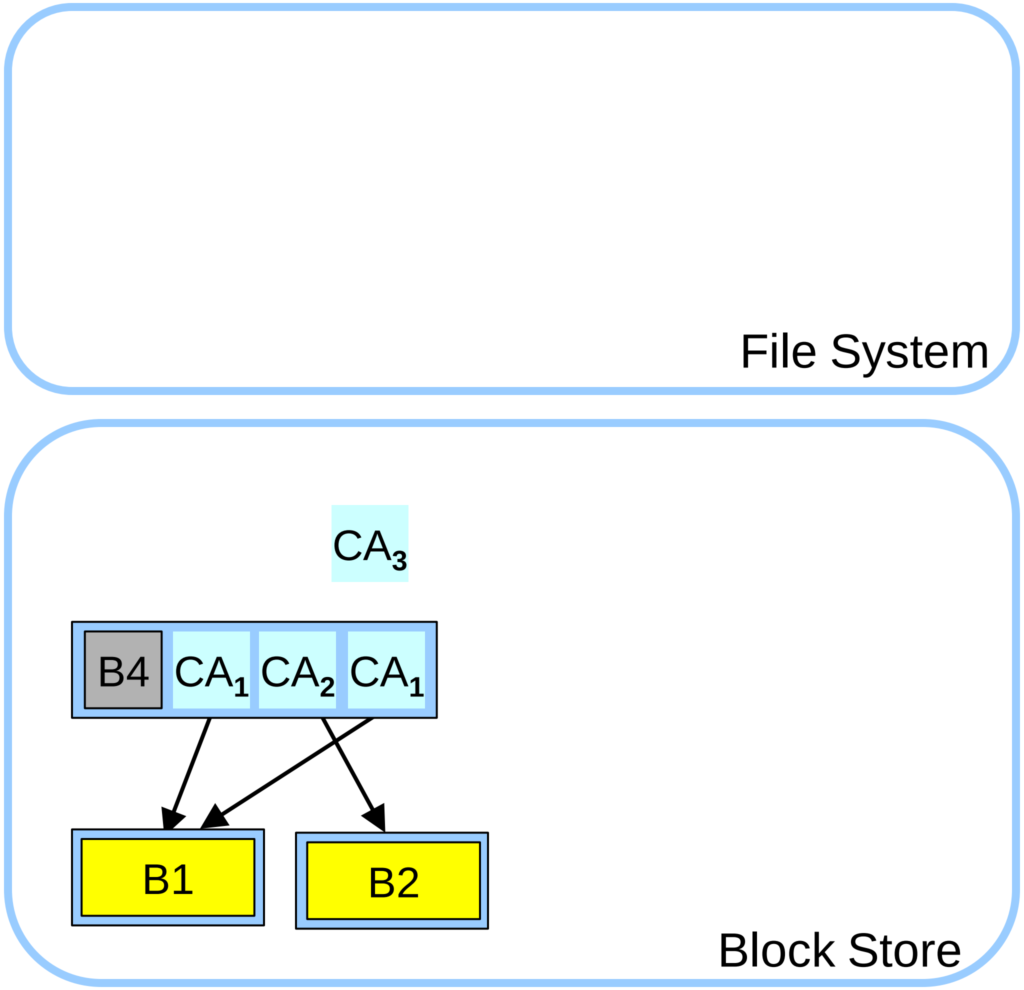
HYDRAstor Usage Example



Block Store (CAS) API

- Variable-size blocks
- Content-addressable
- Address decided by the store
- Duplicates eliminated by store
- Configurable block resilience

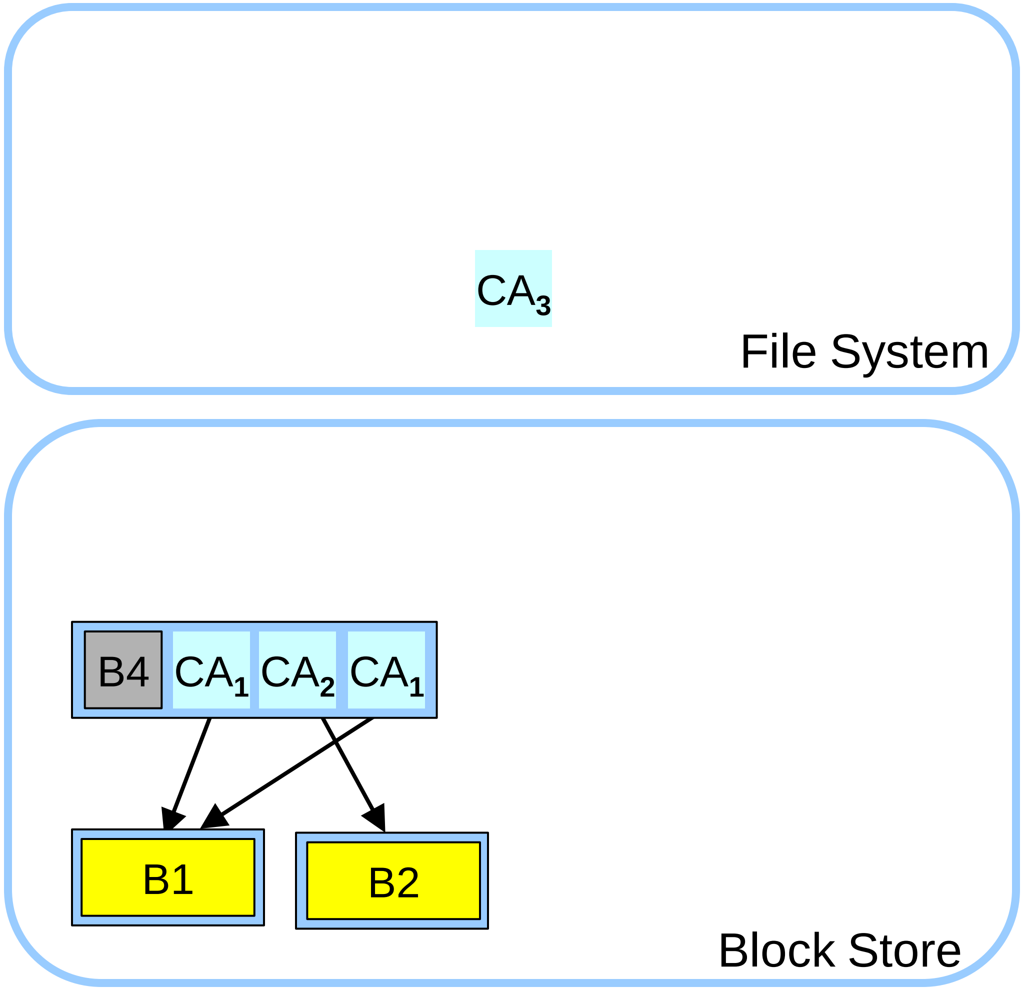
HYDRAstor Usage Example



Block Store (CAS) API

- Variable-size blocks
- Content-addressable
- Address decided by the store
- Duplicates eliminated by store
- Configurable block resilience

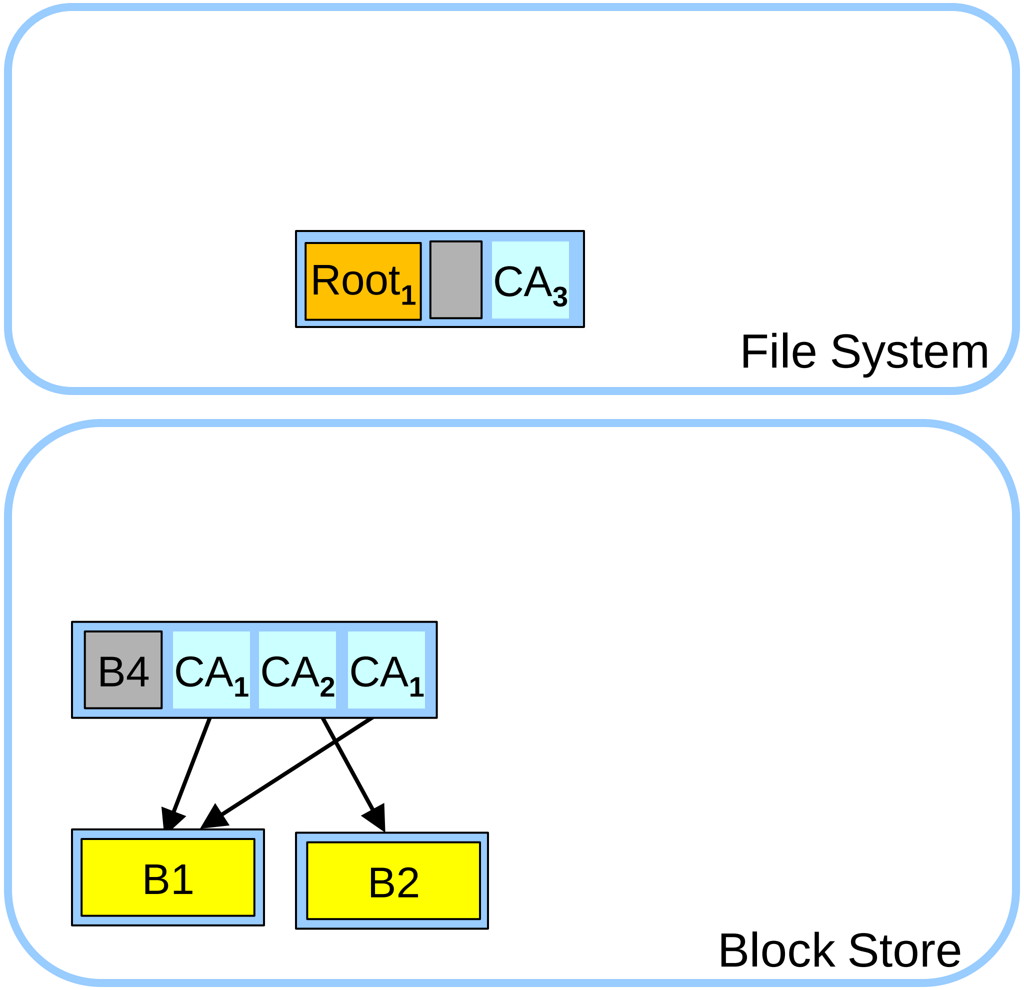
HYDRAstor Usage Example



Block Store (CAS) API

- Variable-size blocks
- Content-addressable
- Address decided by the store
- Duplicates eliminated by store
- Configurable block resilience

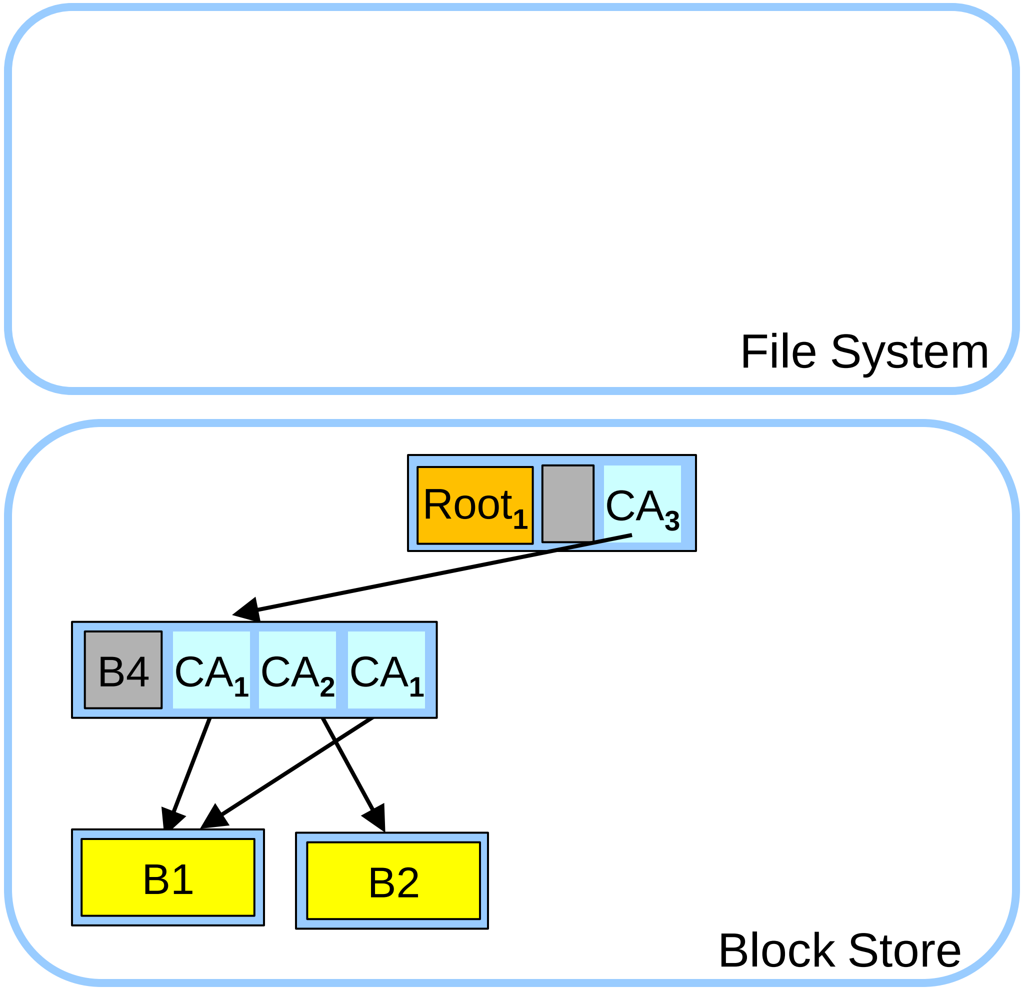
HYDRAstor Usage Example



Block Store (CAS) API

- Variable-size blocks
- Content-addressable
- Address decided by the store
- Duplicates eliminated by store
- Configurable block resilience
- Garbage collection

HYDRAstor Usage Example



Block Store (CAS) API

- Variable-size blocks
- Content-addressable
- Address decided by the store
- Duplicates eliminated by store
- Configurable block resilience
- Garbage collection

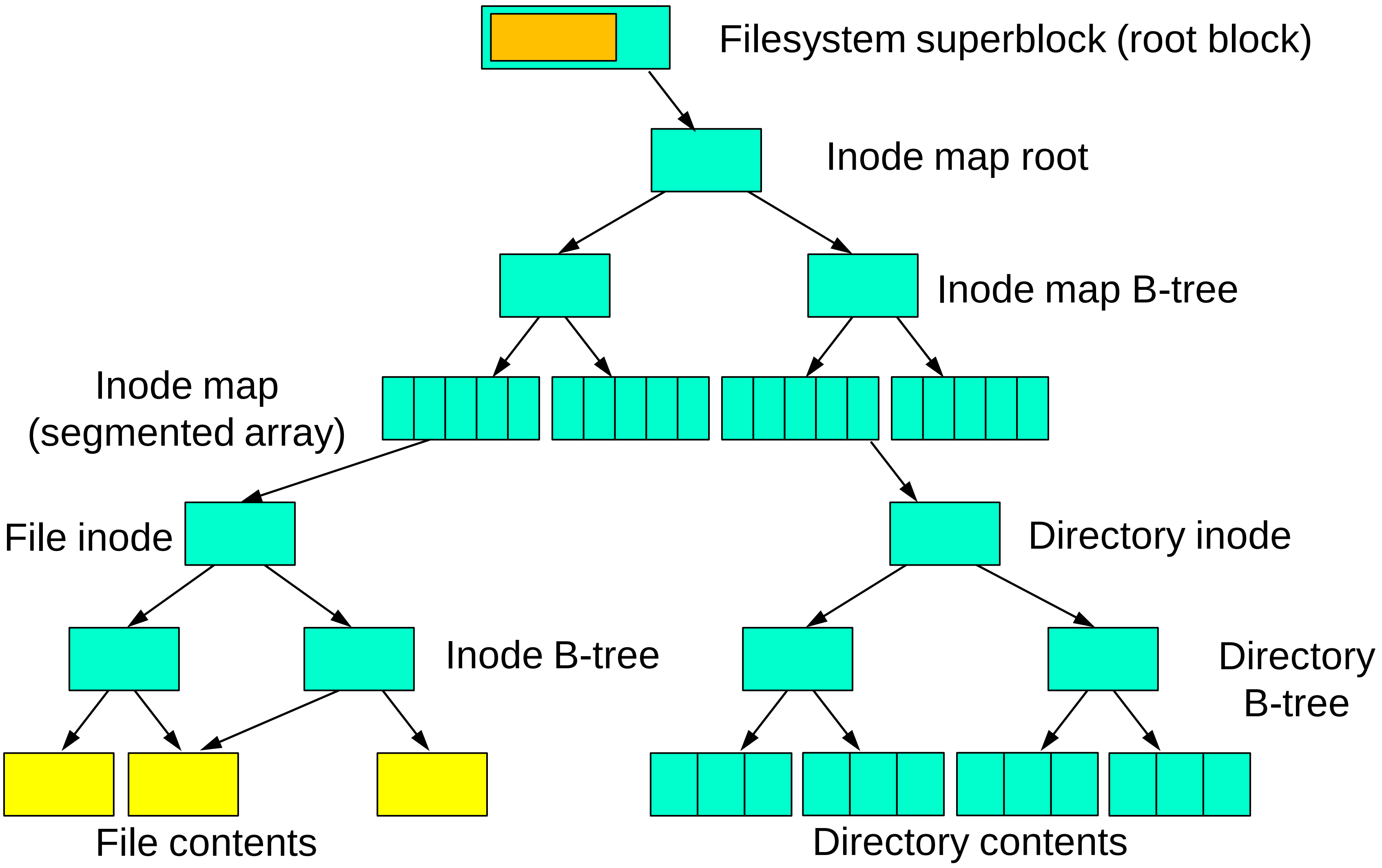
Outline

- HYDRAstor content-addressable API
- Challenges posed to the filesystem
- Filesystem architecture
- Techniques used to overcome the challenges
- Conclusions and future work

Challenges

- Content-addressable blocks
  - A change in a block's contents also changes the block's address
    - All metadata has to change, recursively up to the filesystem root
    - Parent can only be written after the children writes are successful
- Variable-sized *chunking* (splitting file data into blocks)
  - Block boundaries change when content is changed
  - Overwrites cause read-rechunk-rewrite
- High-latency block store operations
  - Why? Hashing, compression, erasure coding, fragment distribution …
  - Exacerbates the above two challenges

Persistent Layout


Filesystem superblock (root block)

Inode map root

Inode map B-tree

Inode map (segmented array)

File inode

Directory inode

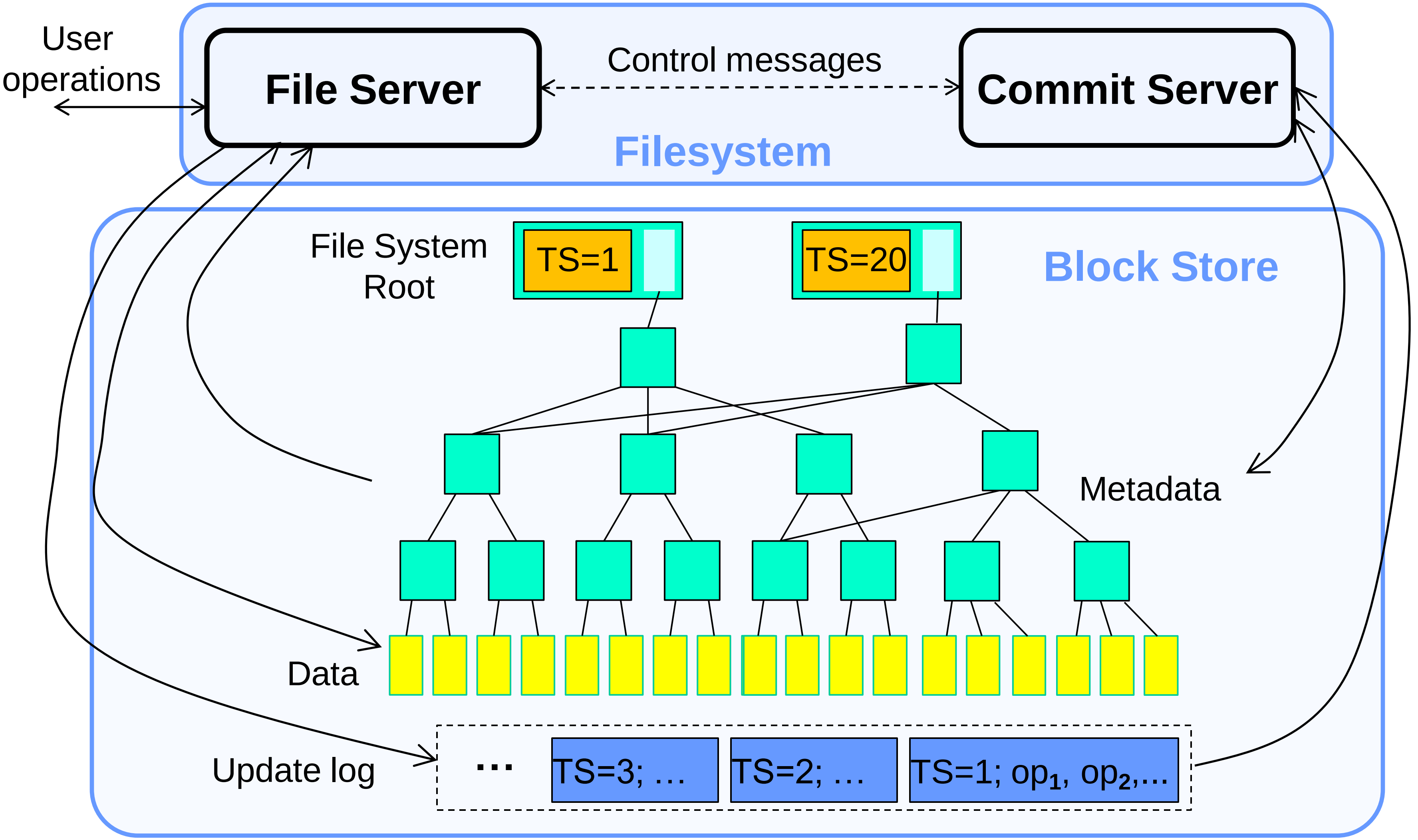
Inode B-tree

Directory B-tree

File contents

Directory contents

HydraFS Architecture



File Server

- Write buffer
  - – Accumulates written data; flushed on sync
  - – Helps re-order NFS packets arriving out-of-order

Write Buffer

(dirty data)

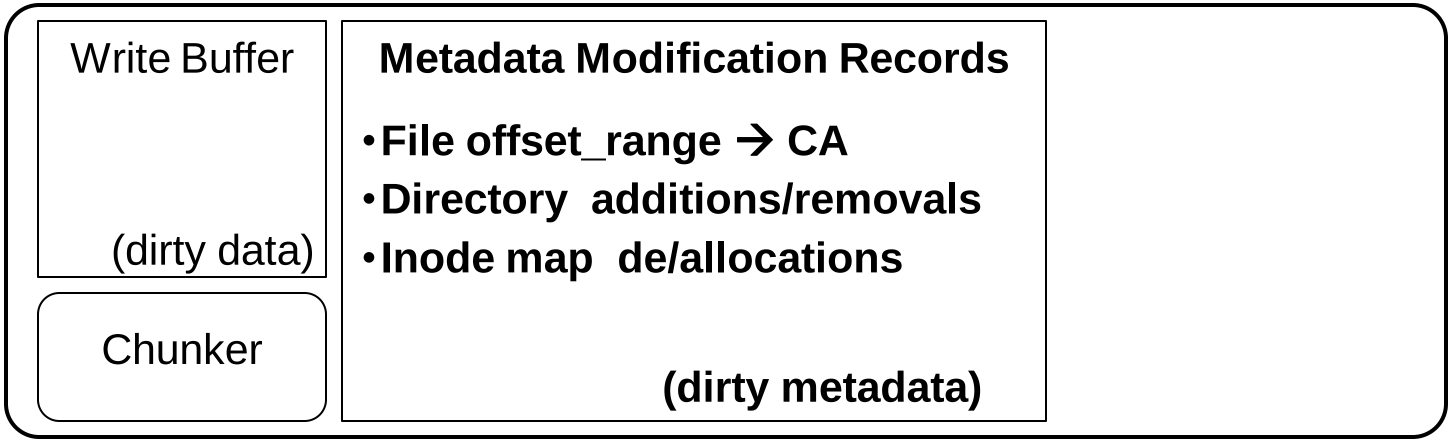
File Server

- Write buffer
  - – Accumulates written data; flushed on sync
  - – Helps re-order NFS packets arriving out-of-order
- Chunker
  - – Decides block boundaries (based on data content)

Write Buffer
(dirty data)
Chunker

File Server

- Write buffer
  - – Accumulates written data; flushed on sync
  - – Helps re-order NFS packets arriving out-of-order
- Chunker
  - – Decides block boundaries (based on data content)
- Metadata modification records (file, directory, inode map)
  - – Dirty metadata annotated with time-stamp (for cleaning)
  - – Written out to log
  - – Large amount of dirty metadata! →
    - Requires efficient cleaning
    - Resource management issues



Write Buffer

(dirty data)

Chunker

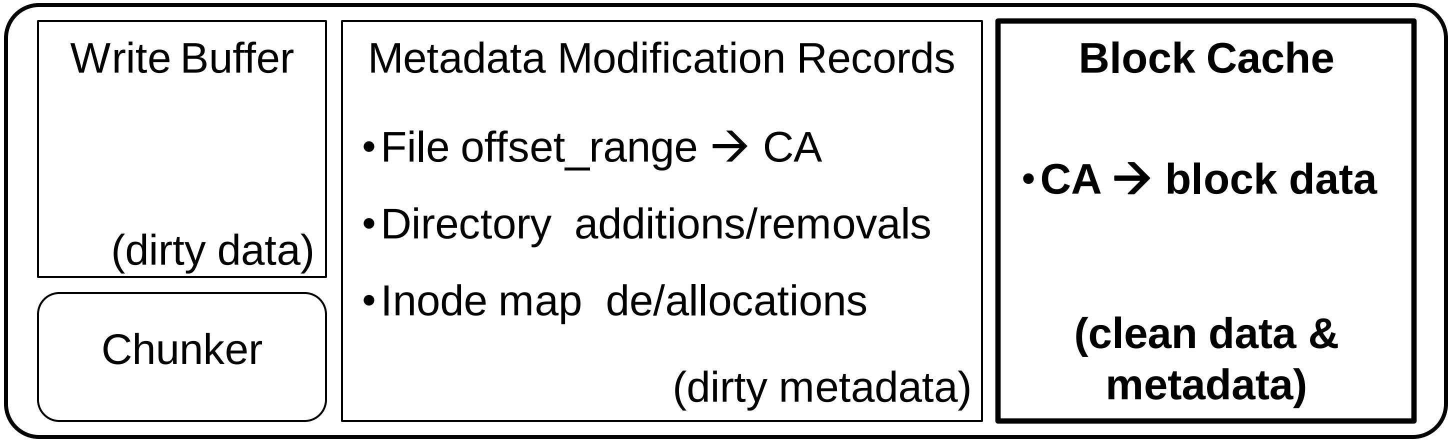
Metadata Modification Records

- **File offset_range → CA**
- **Directory additions/removals**
- **Inode map de/allocations**

(dirty metadata)

File Server

- Write buffer
  - – Accumulates written data; flushed on sync
  - – Helps re-order NFS packets arriving out-of-order
- Chunker
  - – Decides block boundaries (based on data content)
- Metadata modification records (file, directory, inode map)
  - – Dirty metadata annotated with time-stamp (for cleaning)
  - – Written out to log
- Block cache
  - – Clean data and metadata (not de-serialized)



Write Buffer

(dirty data)

Chunker

Metadata Modification Records

- File offset_range → CA
- Directory additions/removals
- Inode map de/allocations

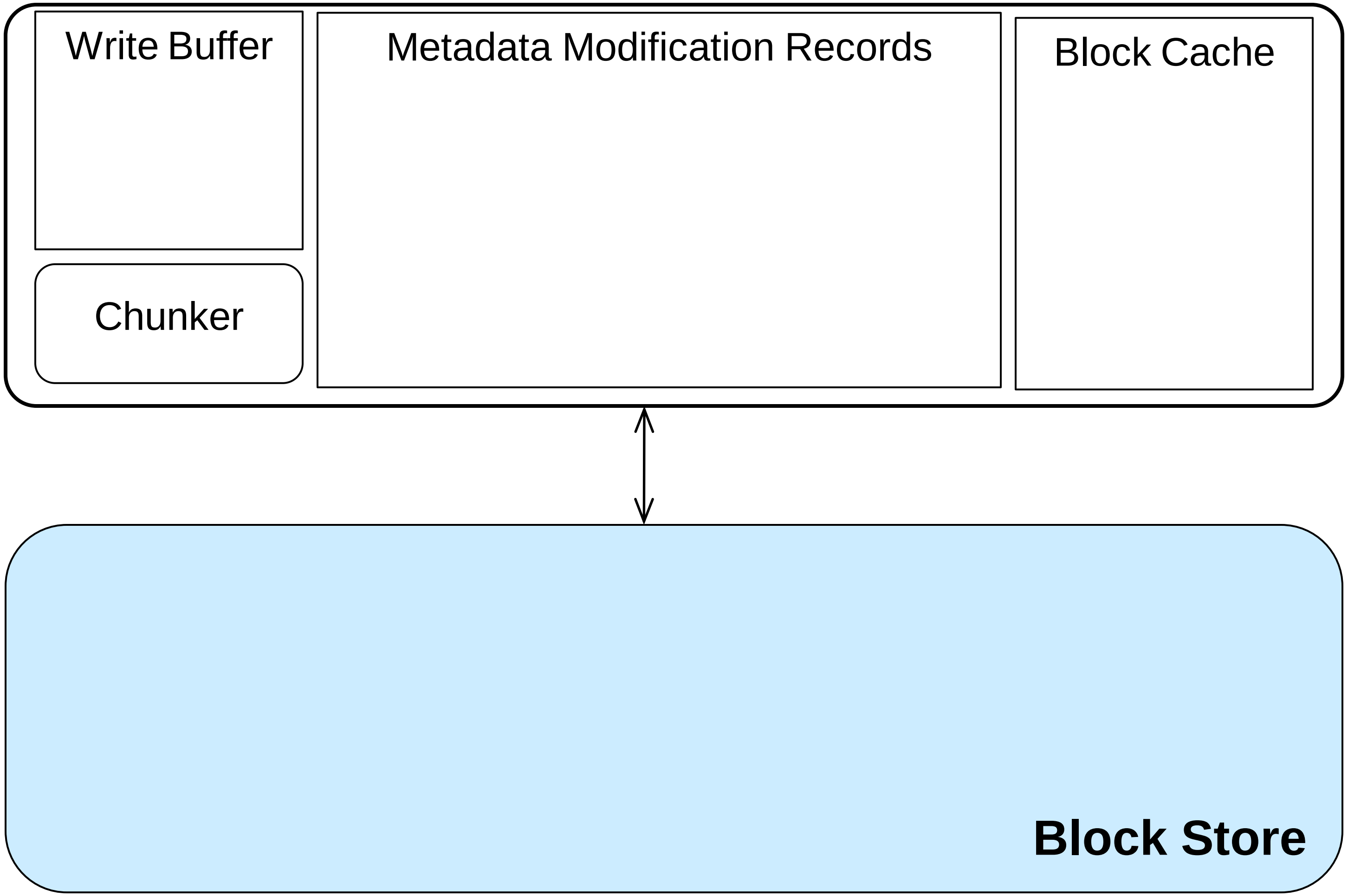
(dirty metadata)

Block Cache

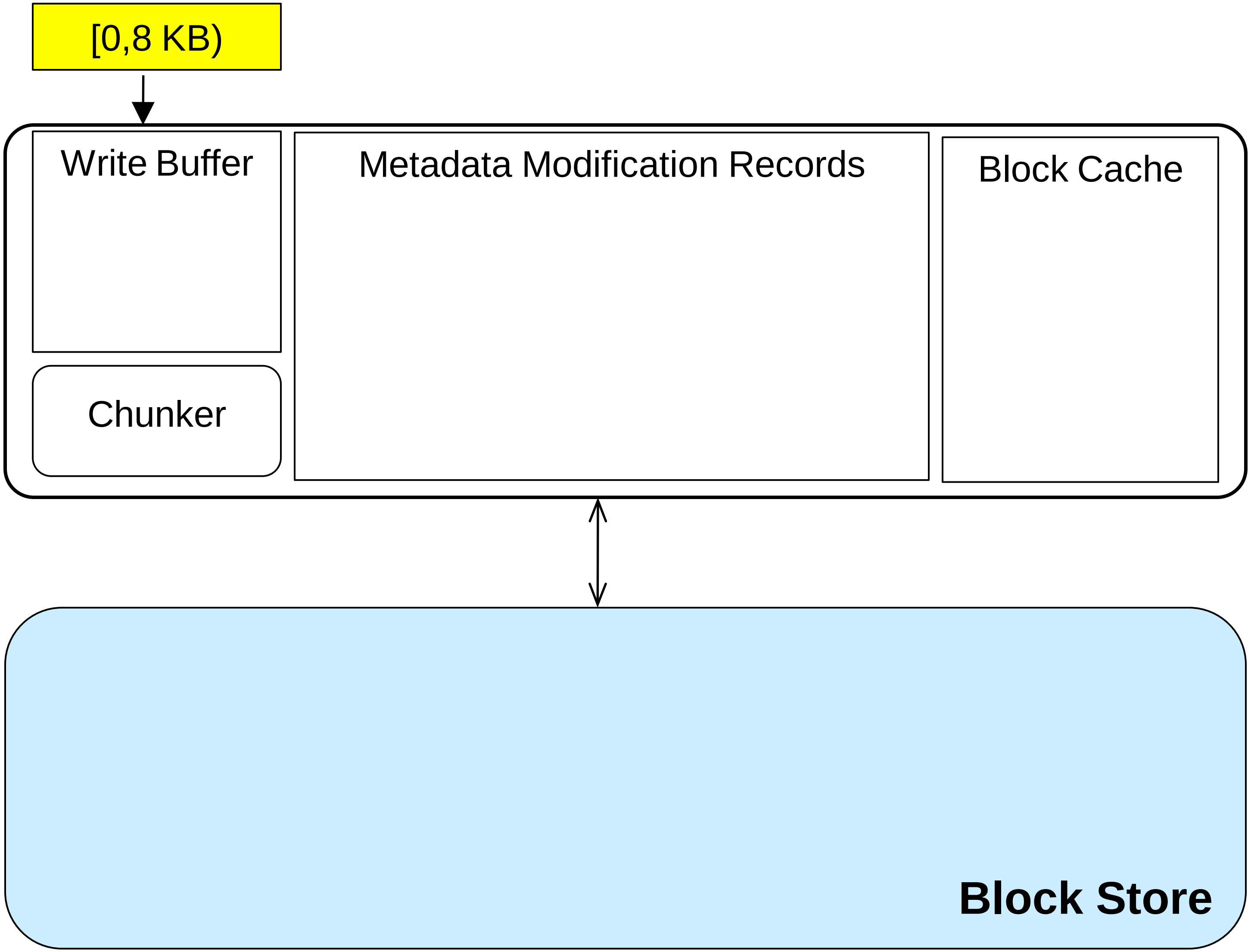
- **CA → block data**

(clean data & metadata)

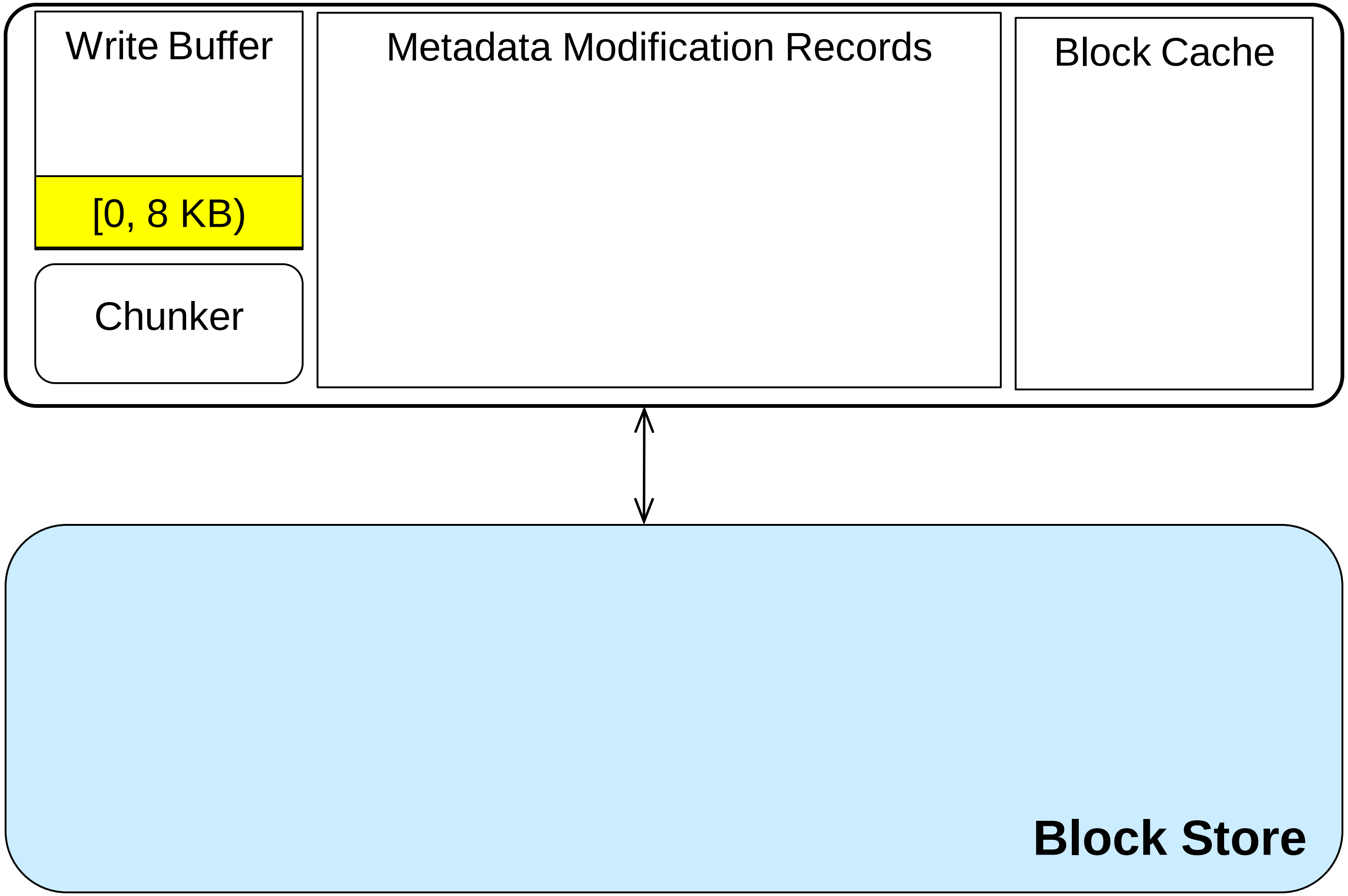
Write Processing



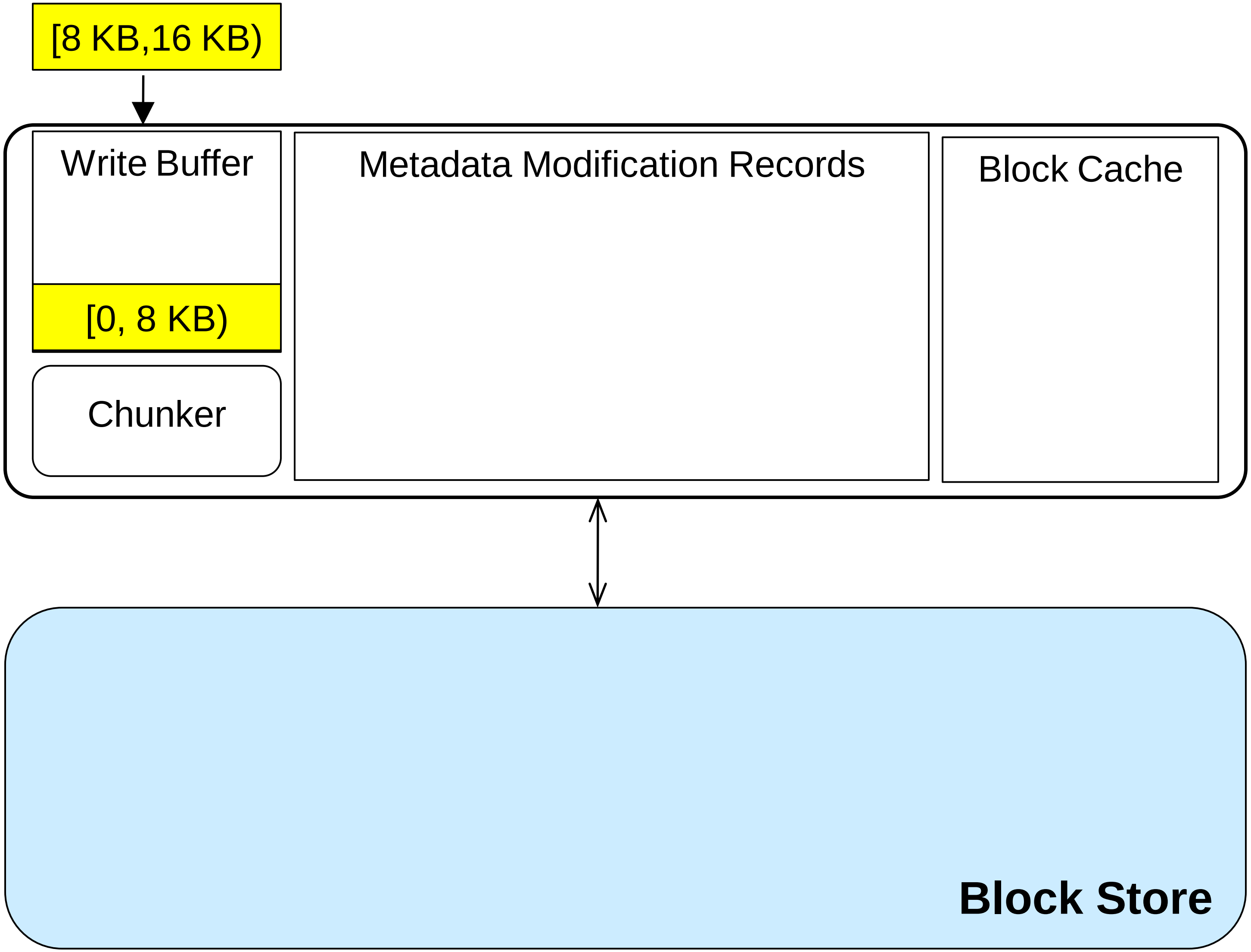
Write Processing



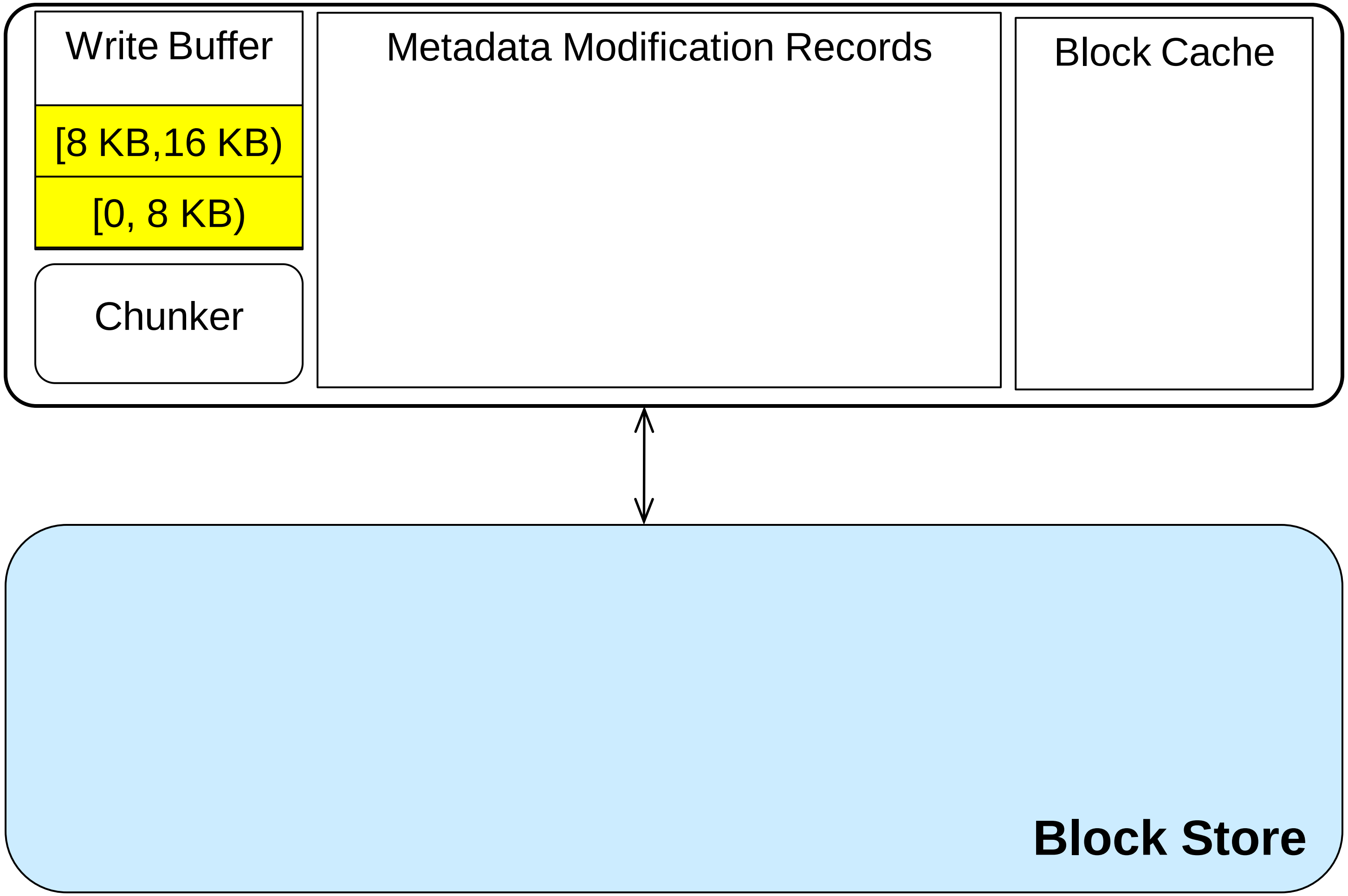
Write Processing



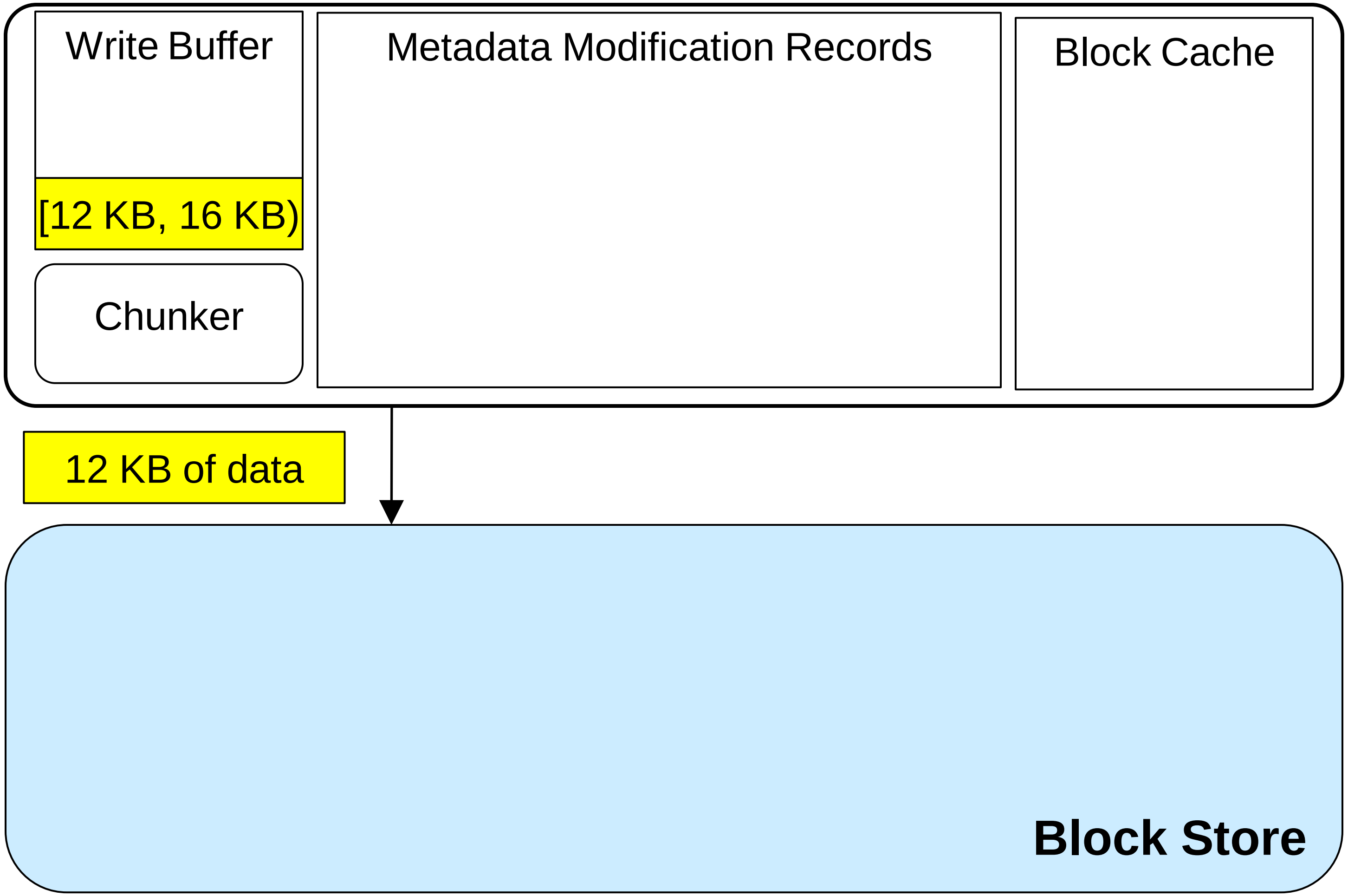
Write Processing



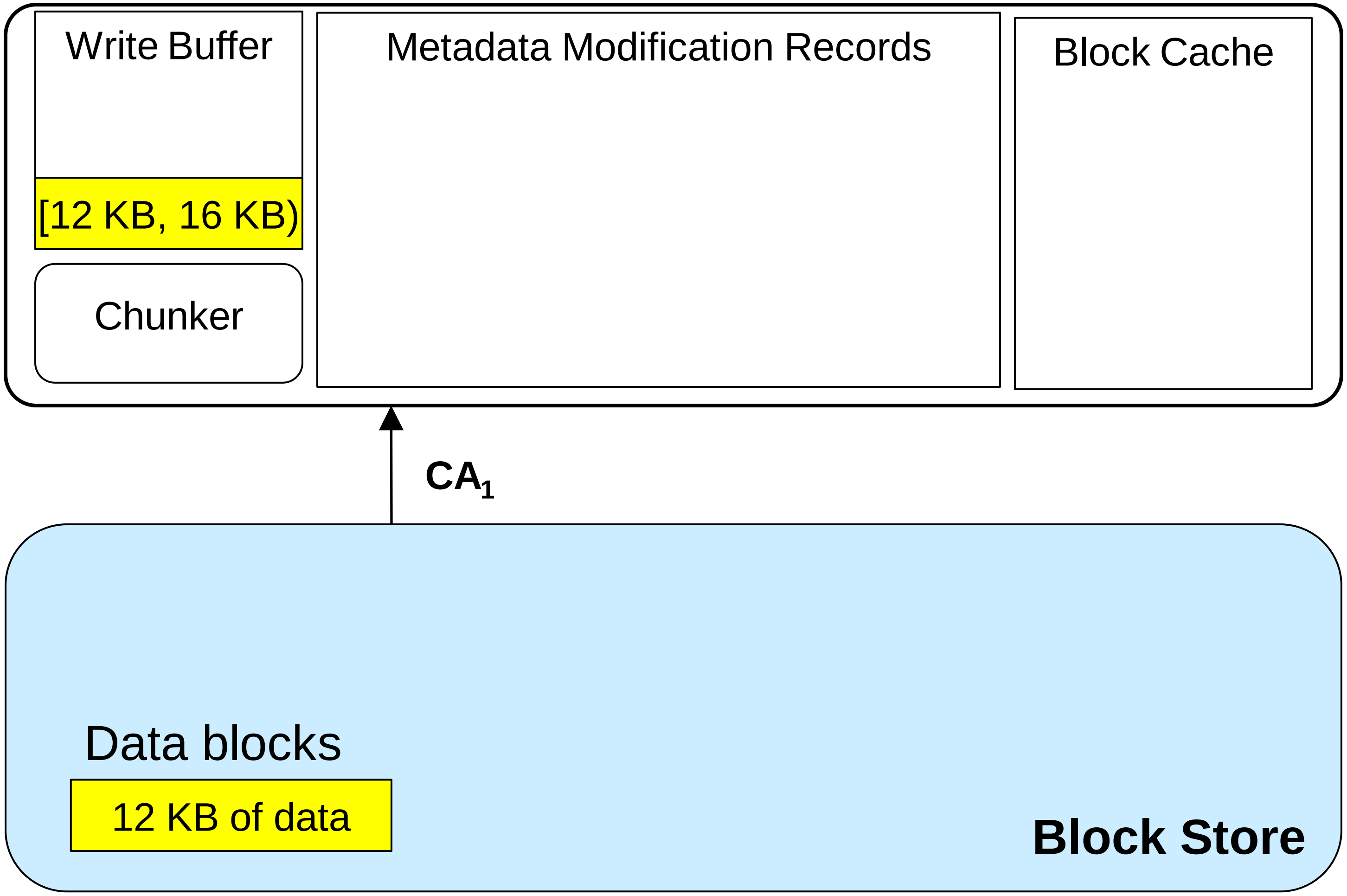
Write Processing



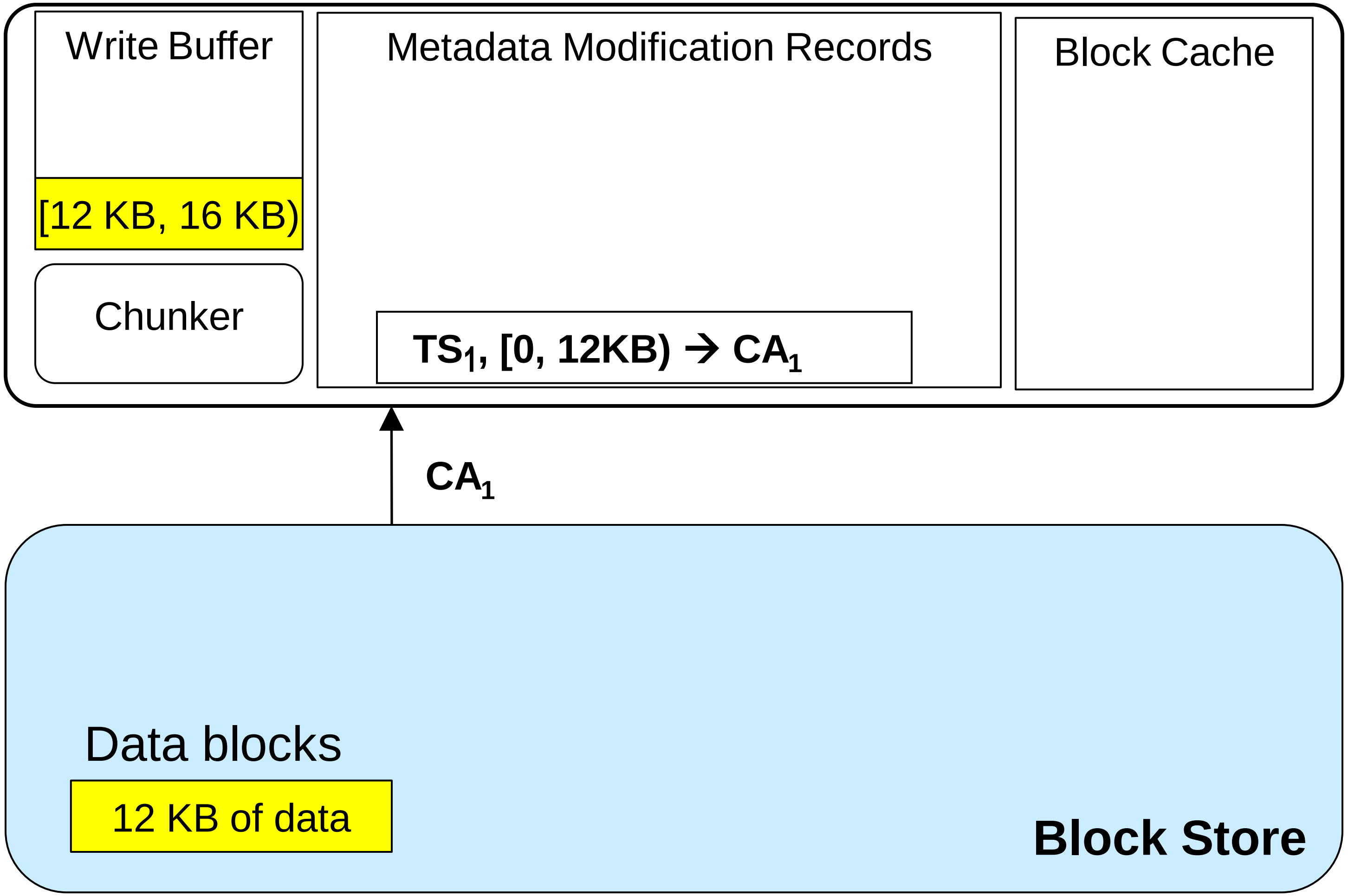
Write Processing



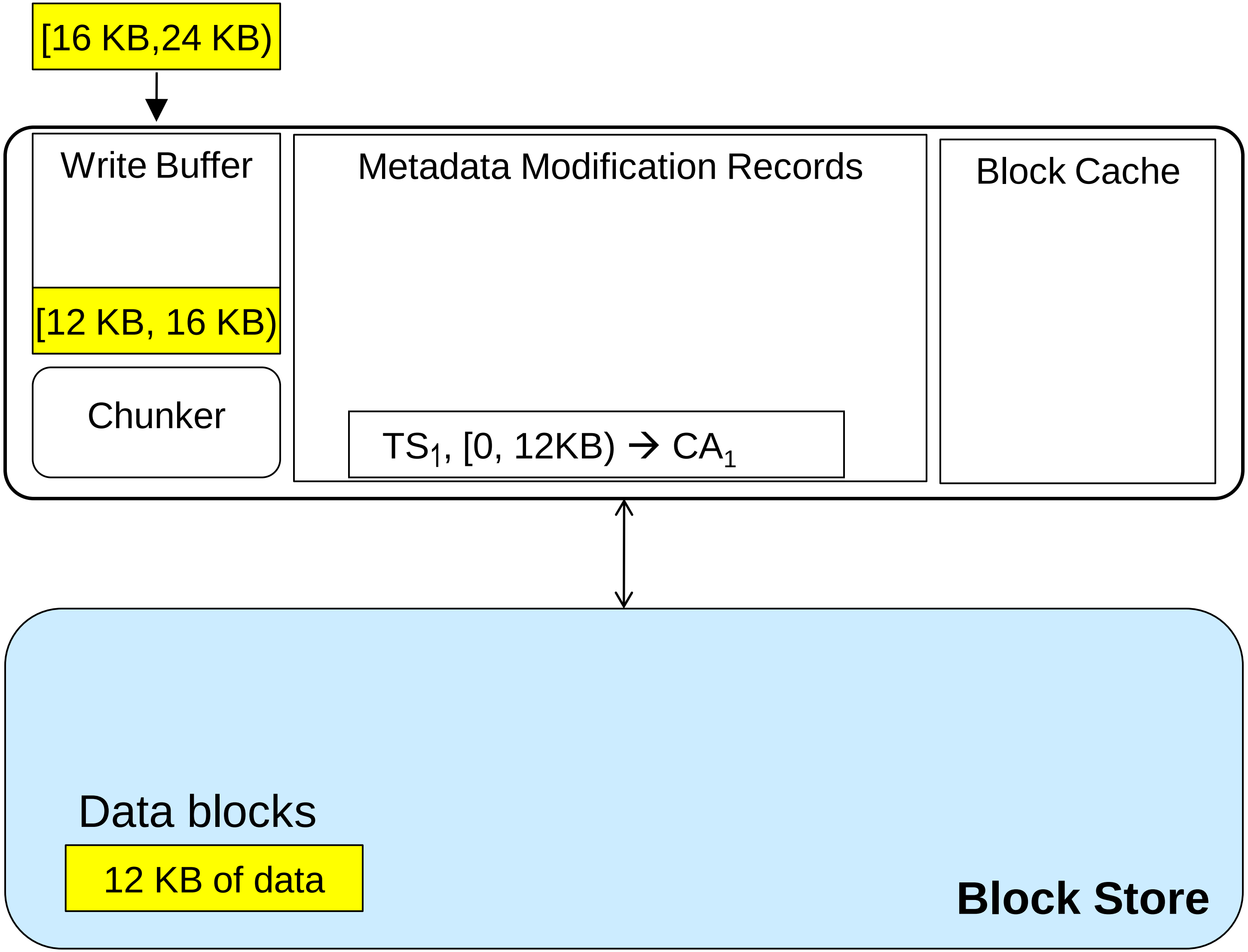
Write Processing



Write Processing



Write Processing



[16 KB,24 KB)

Write Buffer

[12 KB, 16 KB)

Chunker

Metadata Modification Records

TS_1, [0, 12KB) → CA_1

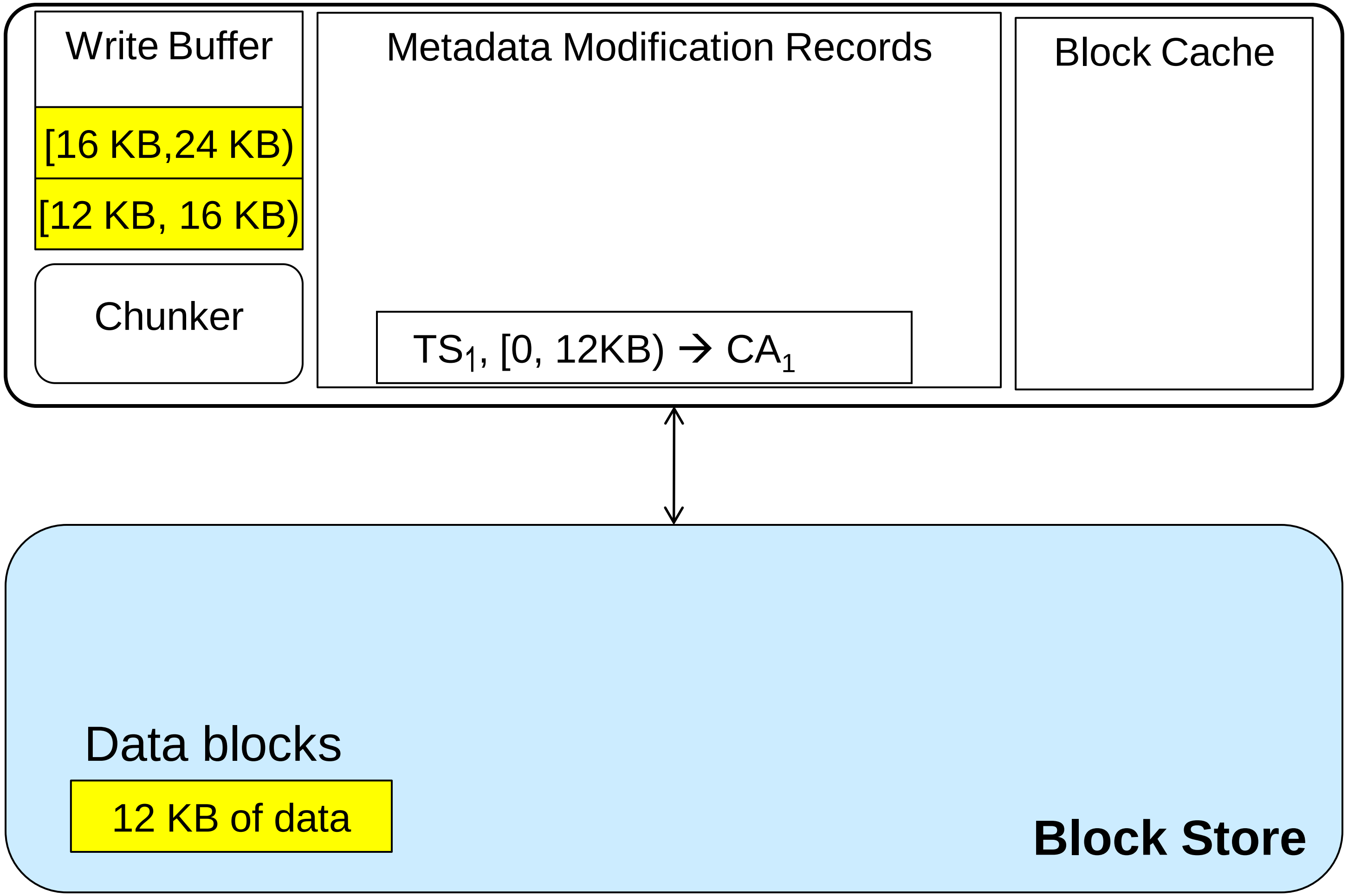
Block Cache

Data blocks

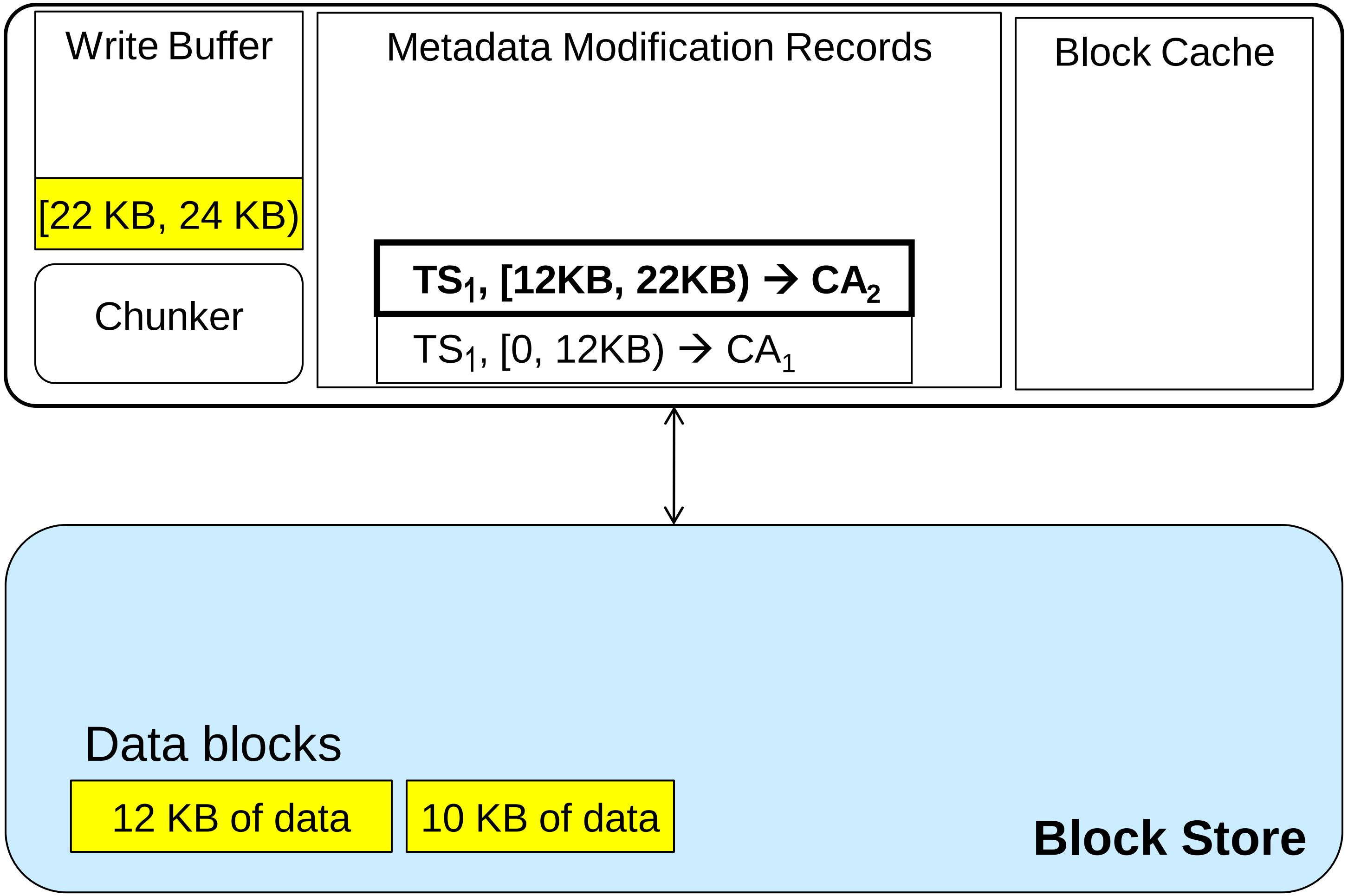
12 KB of data

Block Store

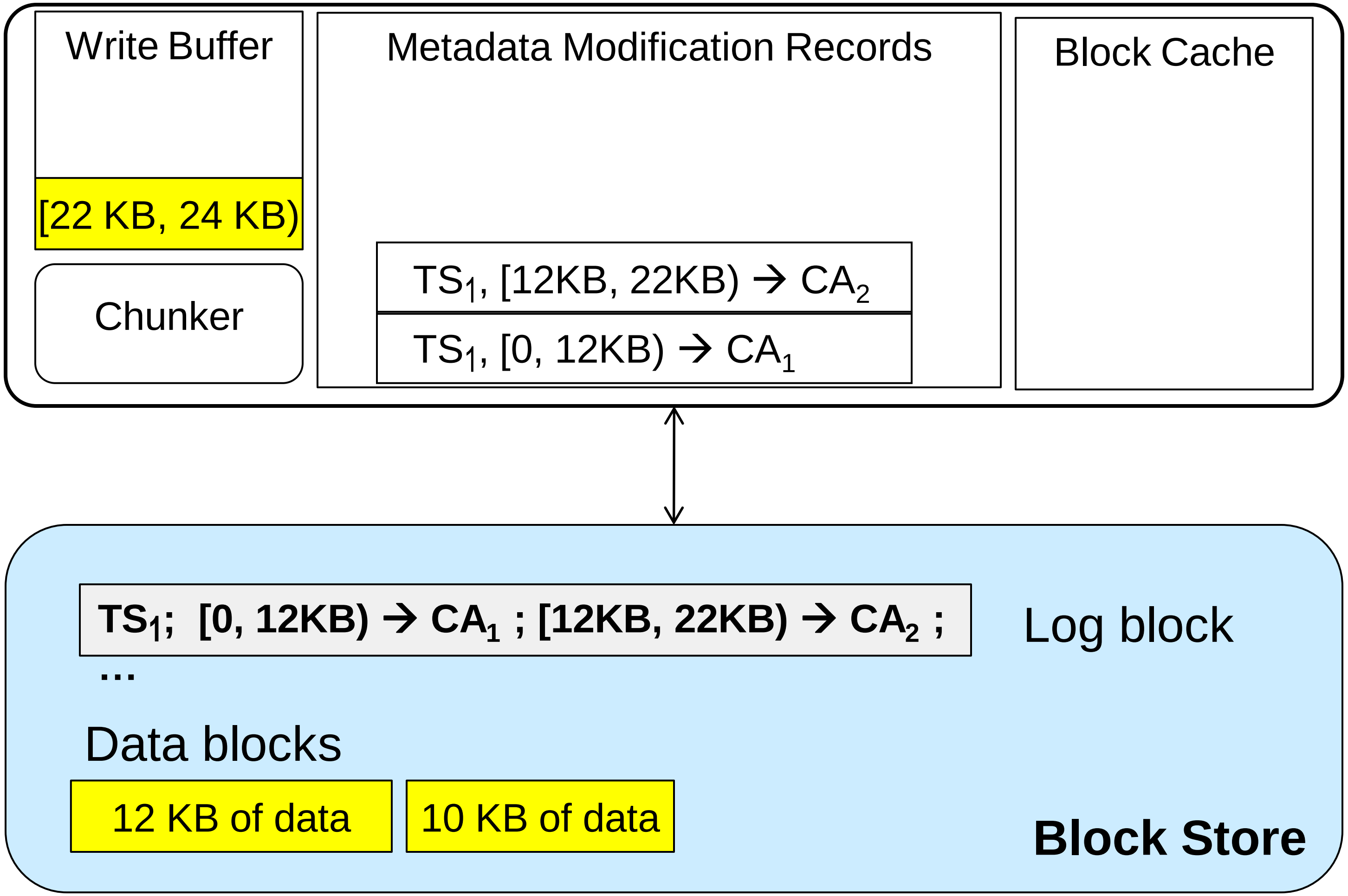
Write Processing



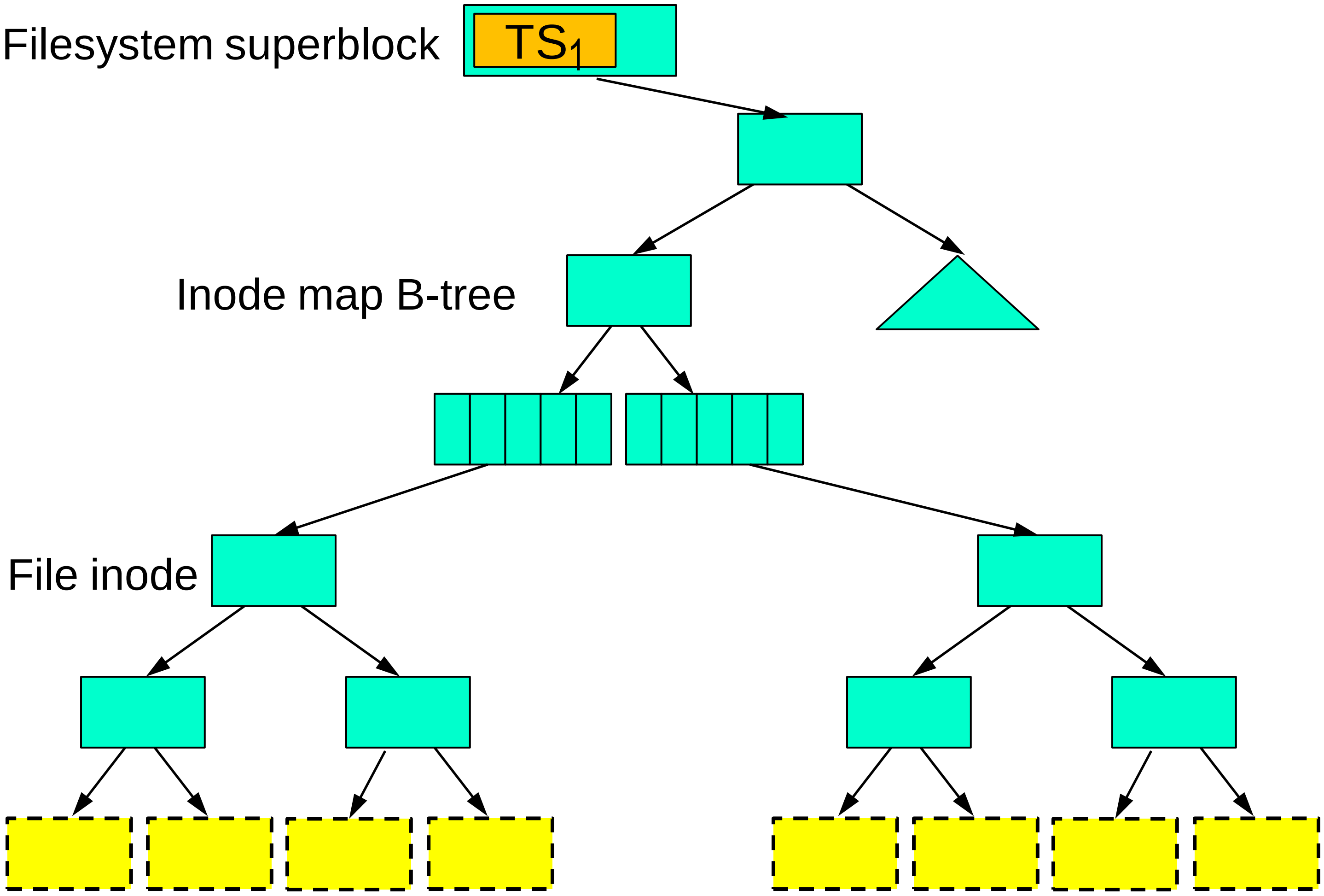
Write Processing



Write Processing

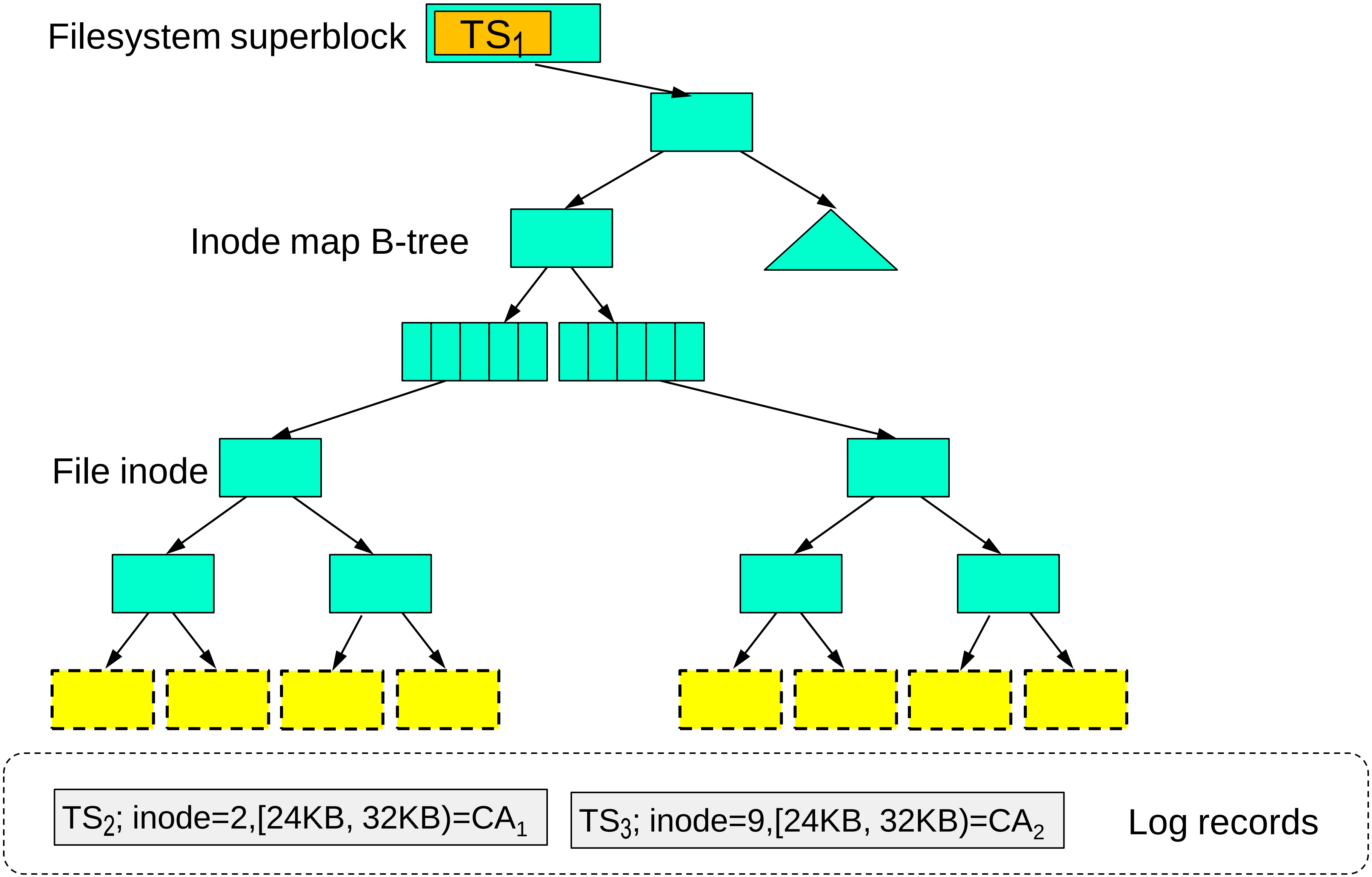

Write Buffer
[22 KB, 24 KB)
Chunker
Metadata Modification Records
TS_1, [12KB, 22KB) → CA_2
TS_1, [0, 12KB) → CA_1
Block Cache
TS_1; [0, 12KB) → CA_1 ; [12KB, 22KB) → CA_2 ;
…
Log block
Data blocks
12 KB of data
10 KB of data
Block Store

Commit Server



- Commit server does not read data

Commit Server



Filesystem superblock

TS_1

Inode map B-tree

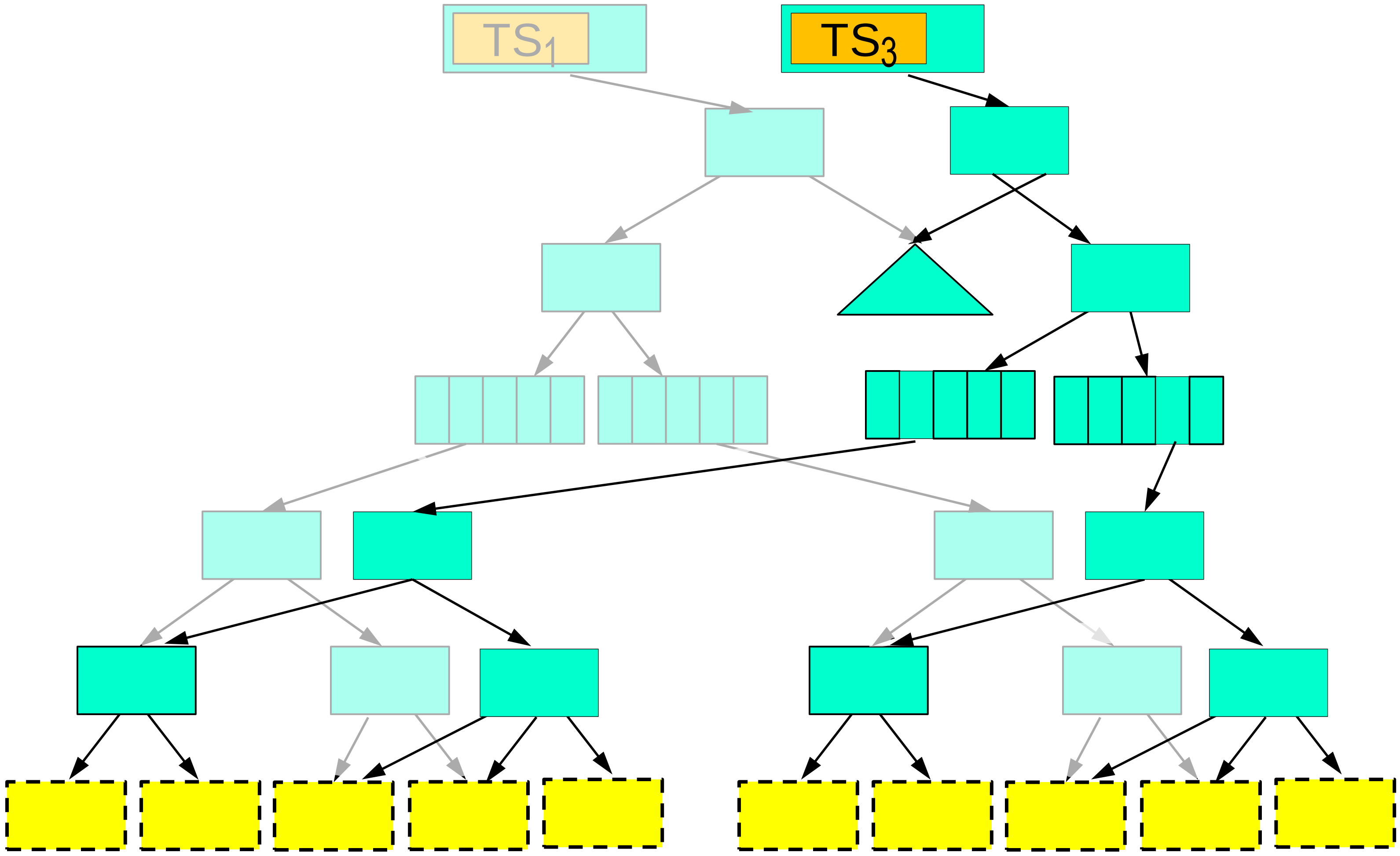
File inode

TS_2; inode=2,[24KB, 32KB)=CA_1

TS_3; inode=9,[24KB, 32KB)=CA_2

Log records

Commit Server



- Amortize updates over many log records
- Recovery time == the time to re-apply log

Metadata Cleaning

File Modification Records

inode #1; root= CA_{101} ; min=791; max=791

791, [8KB, 18KB) → CA_1

791, [18KB, 31KB) → CA_2

inode #2; root= CA_{122} ; min=801; max=803

801, [8KB, 18KB) → CA_3

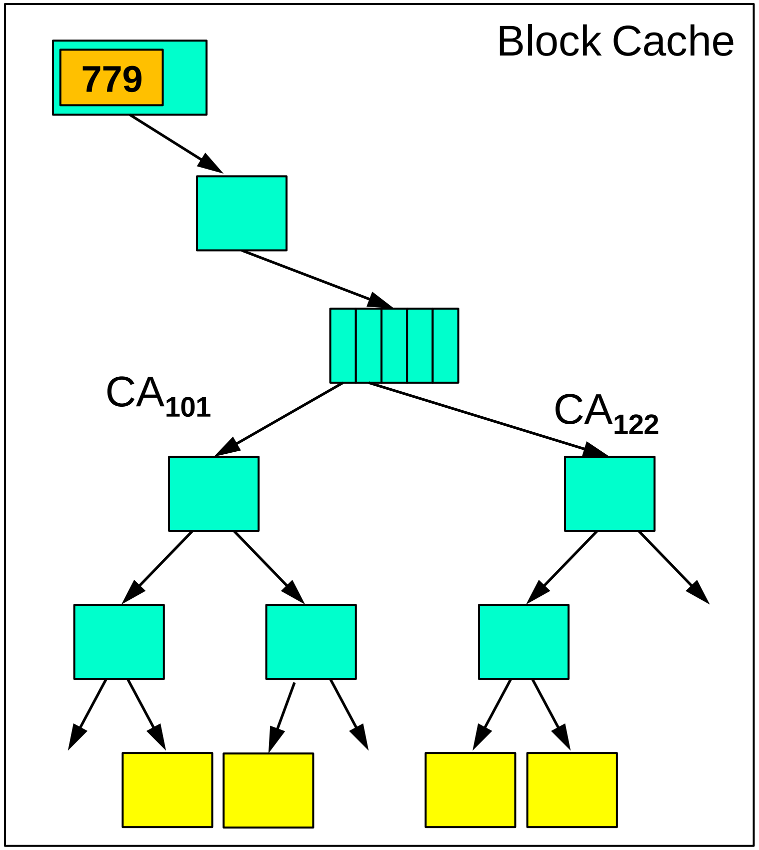
803, [18KB, 29KB) → CA_4

inode #3; root= CA_{303} ; min=805; max=806

805, [0KB, 10KB) → CA_1

806, [10KB, 21KB) → CA_4

⋮



Metadata Cleaning

update(TimeStamp=802)

File Modification Records

inode #1; root= CA_{101} ; min=791; max=791

791, [8KB, 18KB) → CA_1

791, [18KB, 31KB) → CA_2

inode #2; root= CA_{122} ; min=801; max=803

801, [8KB, 18KB) → CA_3

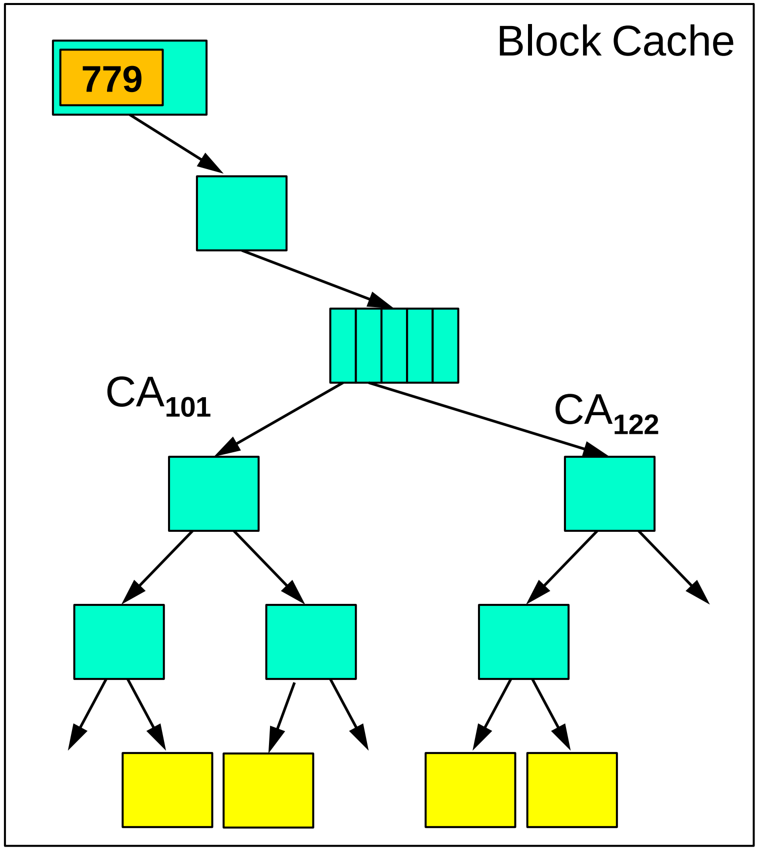
803, [18KB, 29KB) → CA_4

inode #3; root= CA_{303} ; min=805; max=806

805, [0KB, 10KB) → CA_1

806, [10KB, 21KB) → CA_4

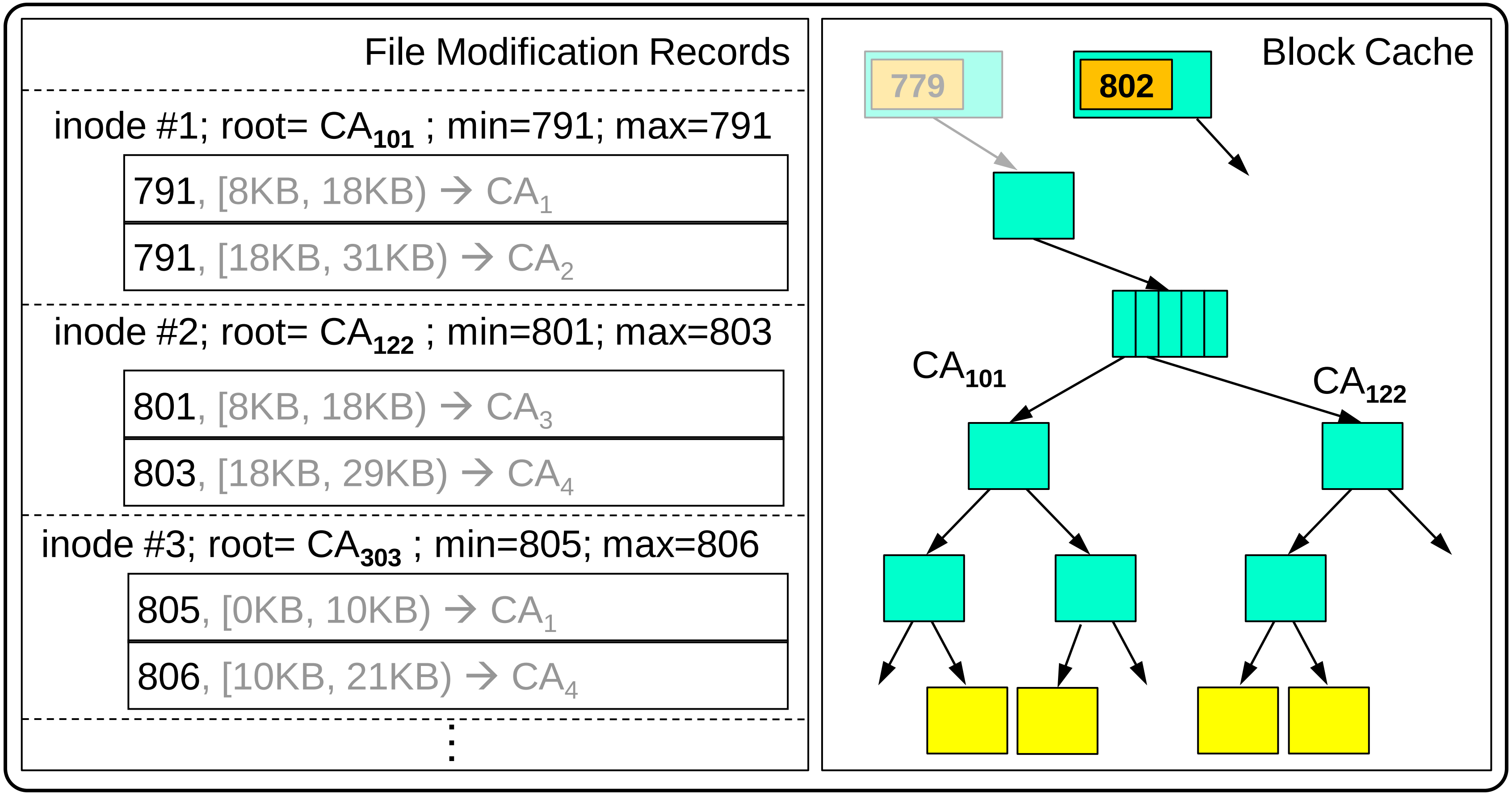
⋮



Metadata Cleaning

Read new, evict old superblock

update(TimeStamp=802)



File Modification Records

inode #1; root= CA_{101} ; min=791; max=791

791, [8KB, 18KB) → CA_1

791, [18KB, 31KB) → CA_2

inode #2; root= CA_{122} ; min=801; max=803

801, [8KB, 18KB) → CA_3

803, [18KB, 29KB) → CA_4

inode #3; root= CA_{303} ; min=805; max=806

805, [0KB, 10KB) → CA_1

806, [10KB, 21KB) → CA_4

⋮

Metadata Cleaning

Process dirty inodes one by one

update(TimeStamp=802)

File Modification Records

inode #1; root= CA$_{101}$; min=791; max=791

791, [8KB, 18KB) → CA$_1$

791, [18KB, 31KB) → CA$_2$

inode #2; root= CA$_{122}$; min=801; max=803

801, [8KB, 18KB) → CA$_3$

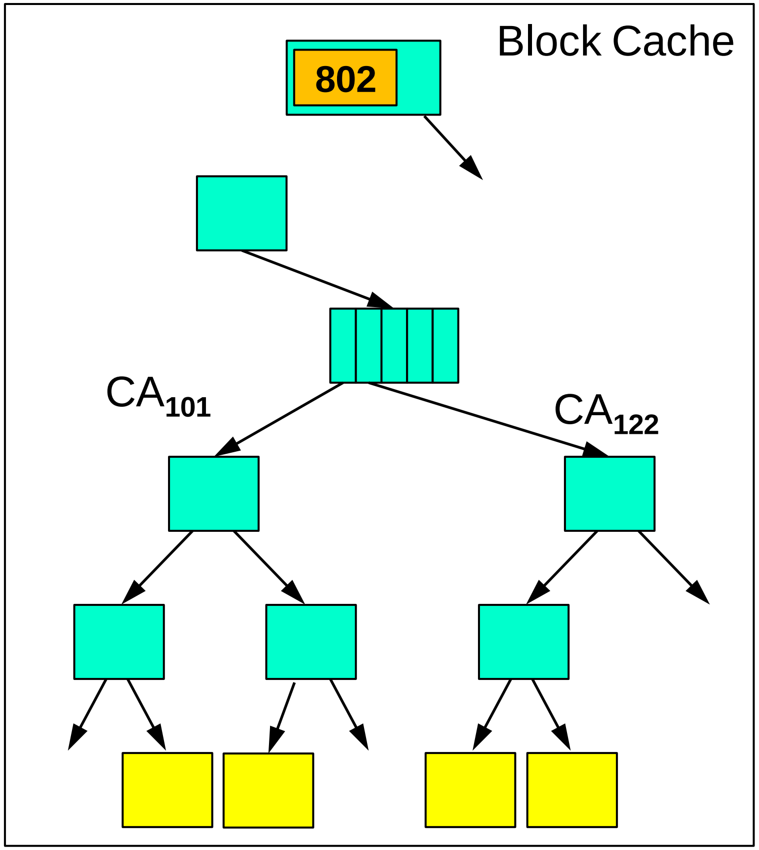
803, [18KB, 29KB) → CA$_4$

inode #3; root= CA$_{303}$; min=805; max=806

805, [0KB, 10KB) → CA$_1$

806, [10KB, 21KB) → CA$_4$

⋮



Metadata Cleaning

Case 1: 802 ≥ max → evict entire inode

update(TimeStamp=802)

File Modification Records

inode #1; root= CA$_{101}$; min=791; max=791

791, [8KB, 18KB) → CA$_{1}$

791, [18KB, 31KB) → CA$_{2}$

inode #2; root= CA$_{122}$; min=801; max=803

801, [8KB, 18KB) → CA$_{3}$

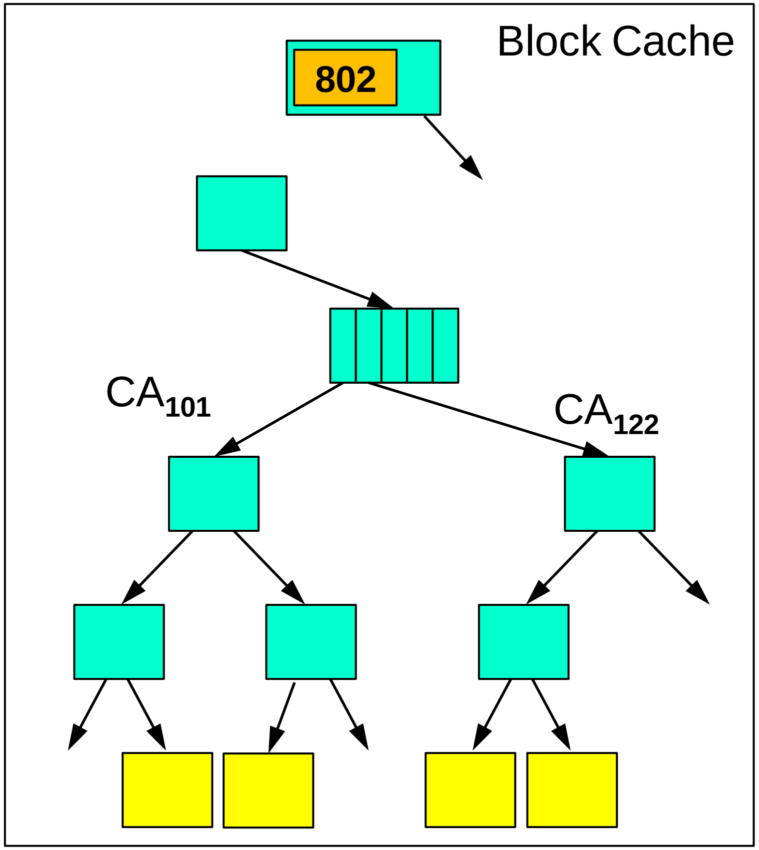
803, [18KB, 29KB) → CA$_{4}$

inode #3; root= CA$_{303}$; min=805; max=806

805, [0KB, 10KB) → CA$_{1}$

806, [10KB, 21KB) → CA$_{4}$

⋮



Metadata Cleaning

Case 2: 802 ≥ min and 802 < max

→ drop root CA

→ drop records with time_stamp ≤ 802

update(TimeStamp=802)

File Modification Records

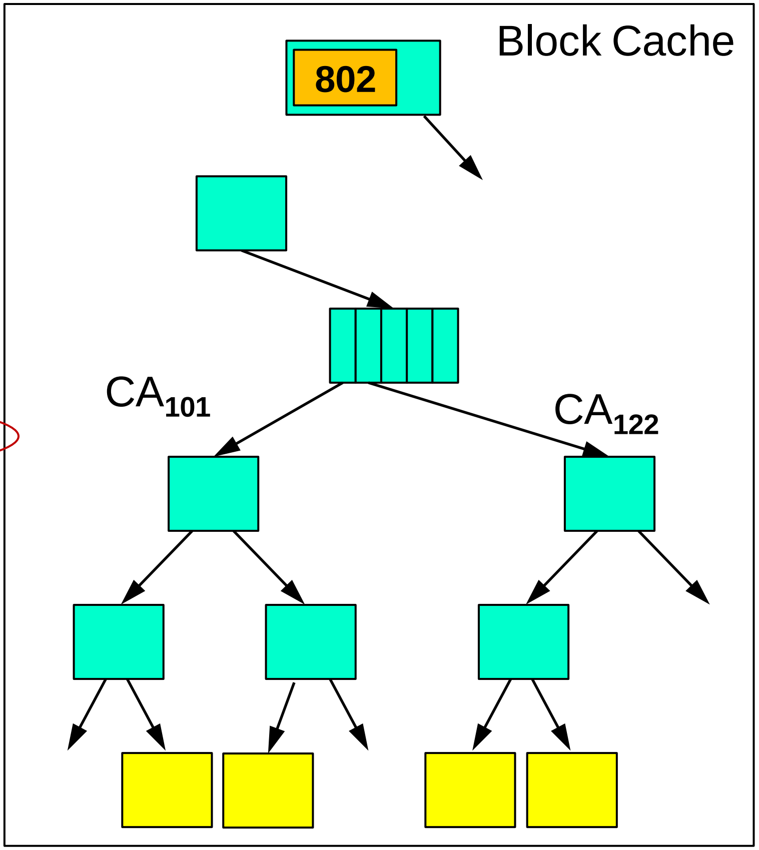
inode #2; root= CA_{122} ; min=801; max=803

- 801, [8KB, 18KB) → CA_3
- 803, [18KB, 29KB) → CA_4

inode #3; root= CA_{303} ; min=805; max=806

- 805, [0KB, 10KB) → CA_1
- 806, [10KB, 21KB) → CA_4

⋮



Metadata Cleaning

Case 3: 802 < min

→skip record processing (all are newer)

→inode root remains unchanged

update(TimeStamp=802)

File Modification Records

inode #2; root= ? ; min=801; max=803

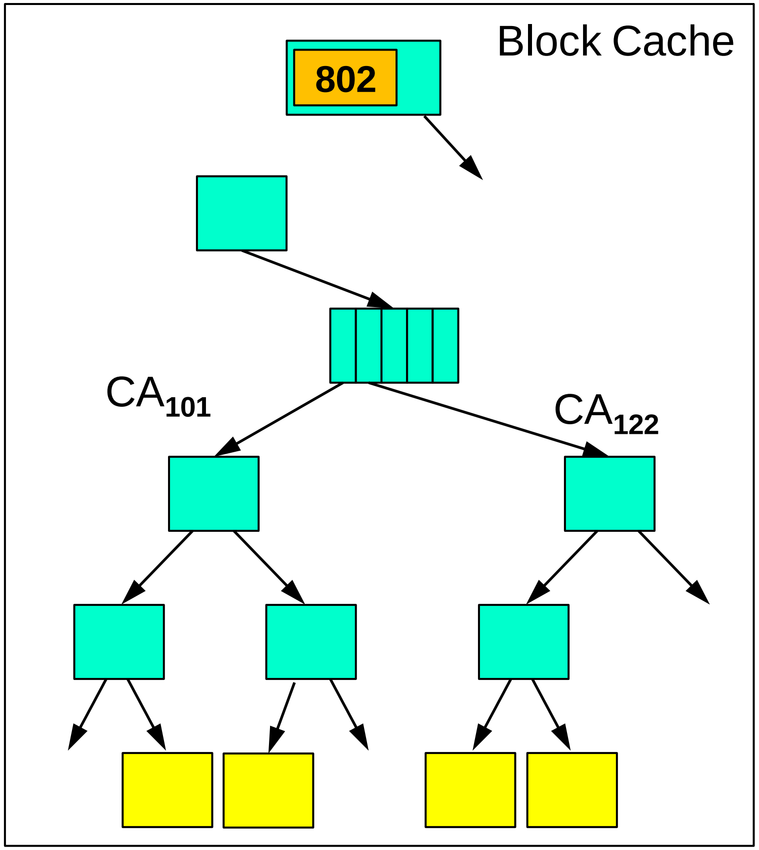
803, [18KB, 29KB) → CA_4

inode #3; root= CA_{303} ; min=805; max=806

805, [0KB, 10KB) → CA_1

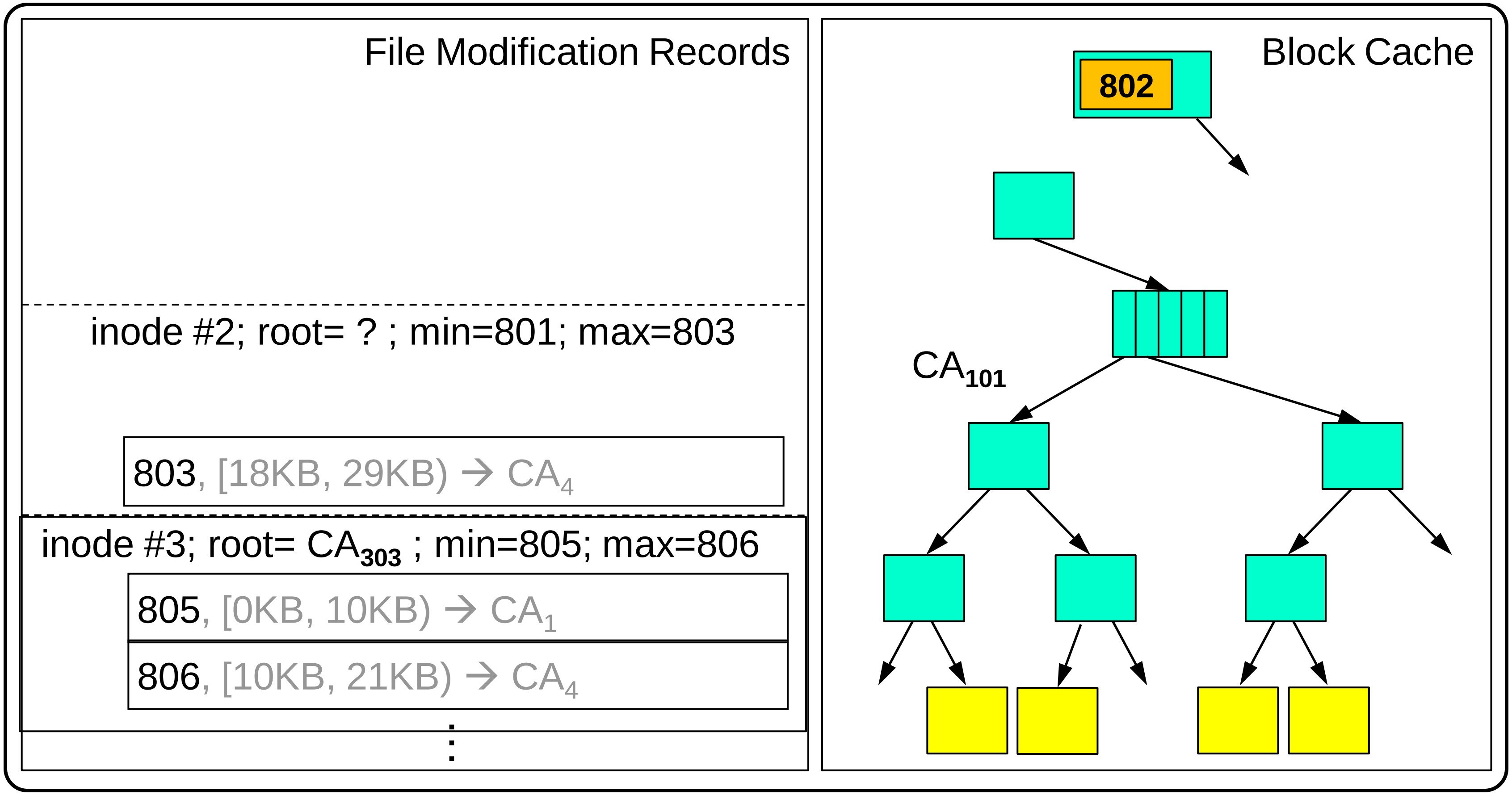
806, [10KB, 21KB) → CA_4

⋮



Metadata Cleaning

- Locks only one inode at a time (no tree locking)
- No I/O done with the lock held



File Modification Records

inode #2; root= ? ; min=801; max=803

803, [18KB, 29KB) → CA_4

inode #3; root= CA_{303} ; min=805; max=806

805, [0KB, 10KB) → CA_1

806, [10KB, 21KB) → CA_4

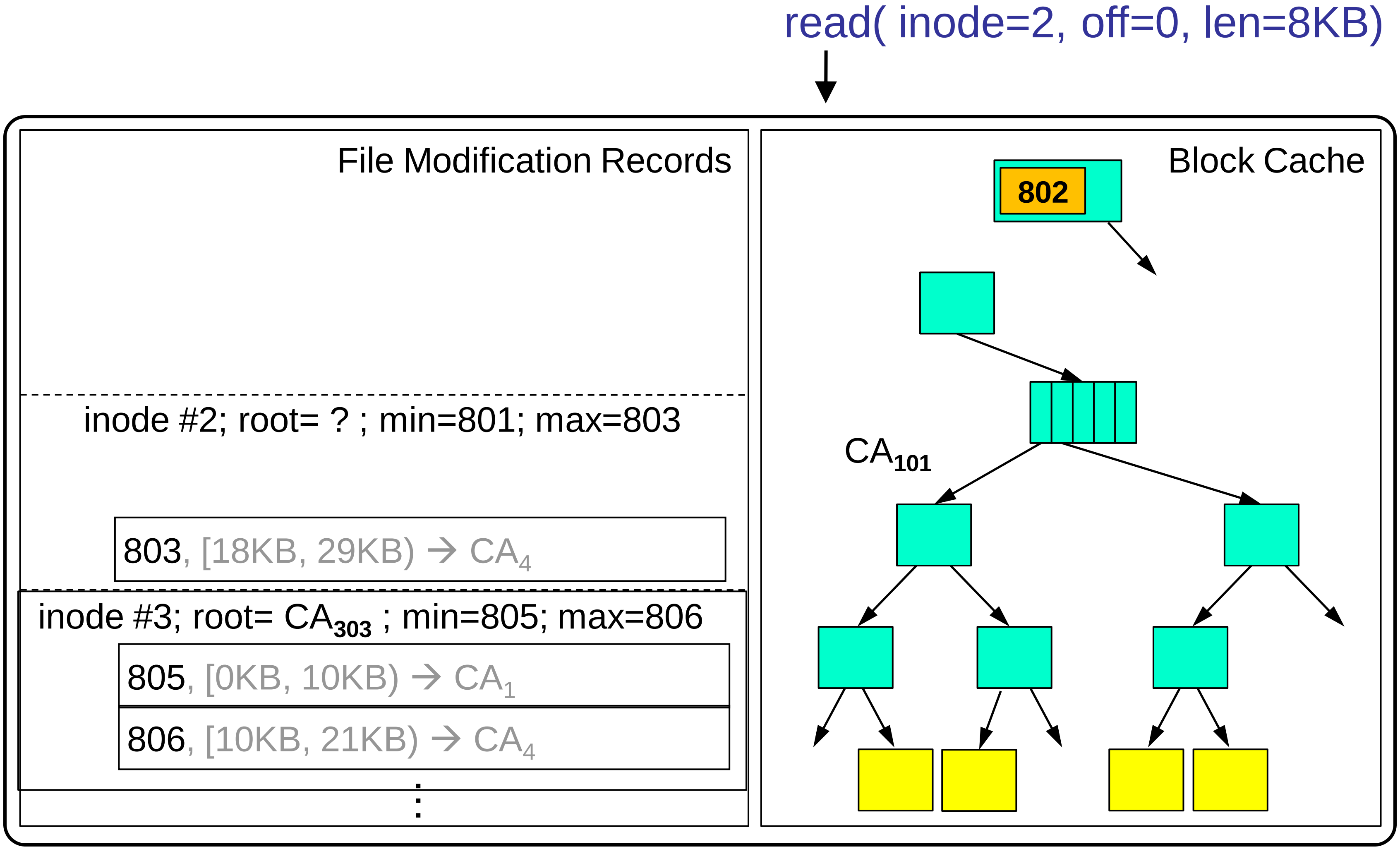
⋮

Block Cache

802

CA_{101}

Read Processing



read(inode=2, off=0, len=8KB)

File Modification Records

inode #2; root= ? ; min=801; max=803

803, [18KB, 29KB) → CA_4

inode #3; root= CA_{303} ; min=805; max=806

805, [0KB, 10KB) → CA_1

806, [10KB, 21KB) → CA_4

⋮

Block Cache

802

CA_{101}

Read Processing

read(inode=2, off=0, len=8KB)

File Modification Records

inode #2; root= ? ; min=801; max=803

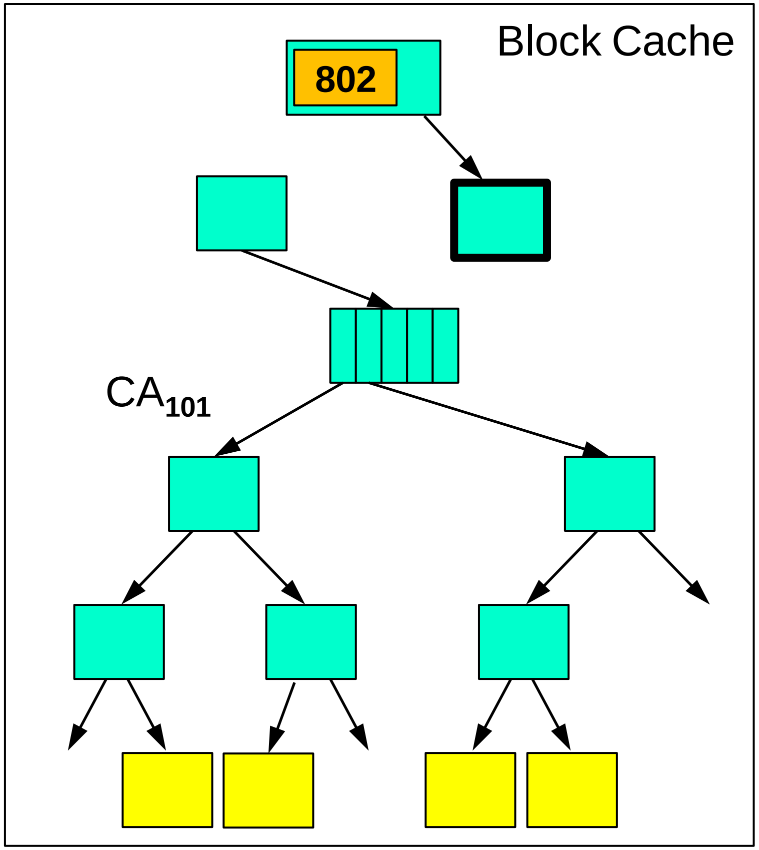
803, [18KB, 29KB) → CA_4

inode #3; root= CA_{303} ; min=805; max=806

805, [0KB, 10KB) → CA_1

806, [10KB, 21KB) → CA_4

⋮



Read Processing

read(inode=2, off=0, len=8KB)

File Modification Records

inode #2; **root=CA_{412}** ; min=801; max=803

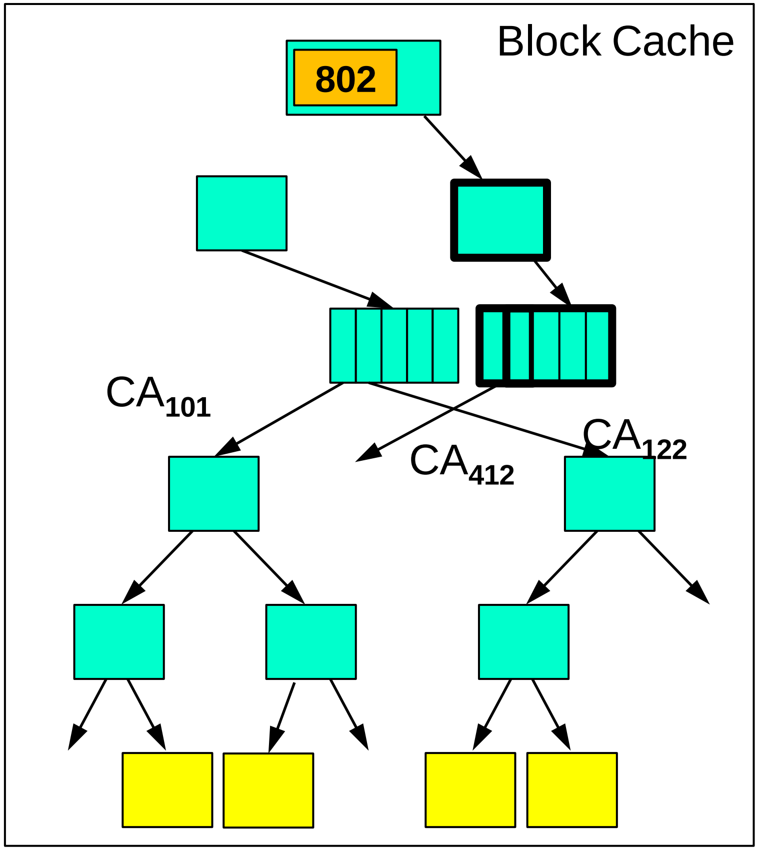
803, [18KB, 29KB) → CA_4

inode #3; root= CA_{303} ; min=805; max=806

805, [0KB, 10KB) → CA_1

806, [10KB, 21KB) → CA_4

⋮



Read Processing

read(inode=2, off=0, len=8KB)

File Modification Records

inode #2; root=CA$_{412}$; min=801; max=803

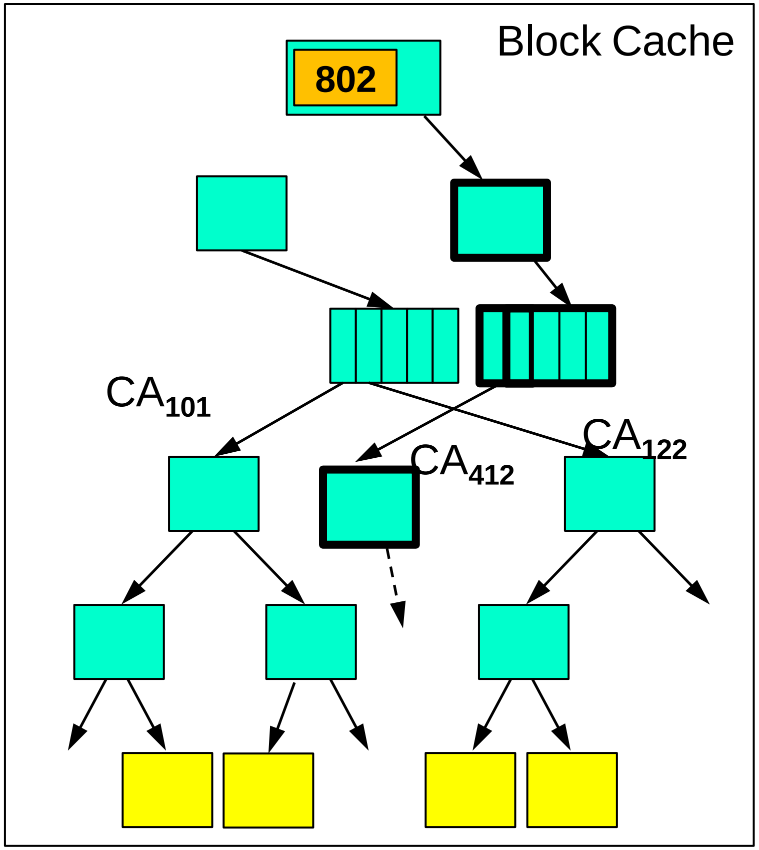
803, [18KB, 29KB) → CA$_4$

inode #3; root= CA$_{303}$; min=805; max=806

805, [0KB, 10KB) → CA$_1$

806, [10KB, 21KB) → CA$_4$

⋮



Read Performance

NEC Laboratories America
Relentless passion for innovation

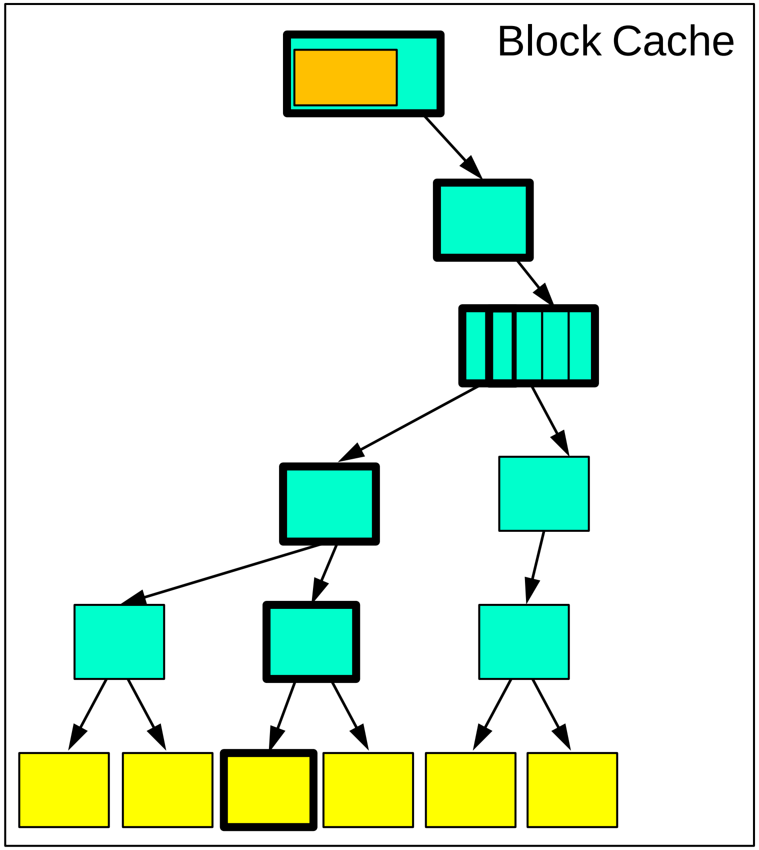
Just pre-fetch?

Problems

- High latency → high read-ahead
- Poor cache locality for metadata

Solutions

- Separate data and meta-data pre-fetch
- Weighted-LRU Policy for Block Cache



Data and Metadata Pre-fetch

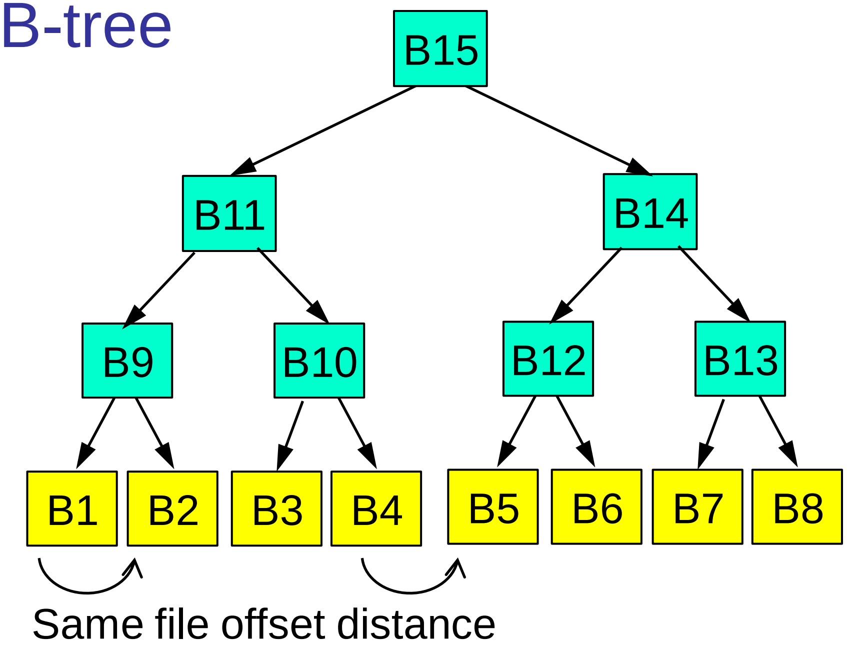
- Problem: time to pre-fetch a data block varies with its position in the B-tree

Compare: B1 – B2 with B4 – B5

- B1: B15 – B11 – B9 – B1
- B2: B15 – B11 – B9 – **B2**

Likely cache miss

- B4: B15 – B11 – B10 – B4
- B5: B15 – **B14 – B12 – B5**



Same file offset distance

- Solution
  - Pre-fetch metadata more aggressively than data

Weighted LRU Policy for Block Cache

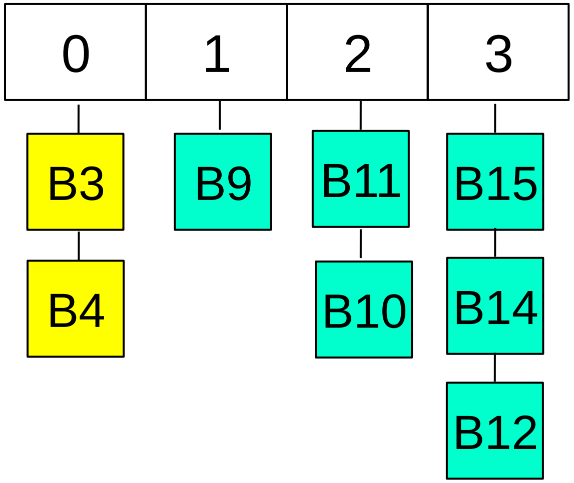
- Problem: different access pattern for data and metadata blocks
  - Data blocks being read
    - Clean pages, pinned until read completes
    - Looked-up once, then unlikely to be needed again (for streaming workloads)
  - Data blocks pre-fetched
    - Clean pages, not pinned
    - Should avoid evicting *before* they are read
  - Metadata blocks
    - Looked-up more than once, but with large duration between accesses

- Solution: cache eviction policy that favors metadata blocks
  - **`Insert`** → Assign weight based on block type
  - **`Lookup`** → Reset to initial weight, and make MRU in that bucket
  - **`Reclaim`** → Evict blocks with zero weight;
    Decrease everybody else's weight with 1

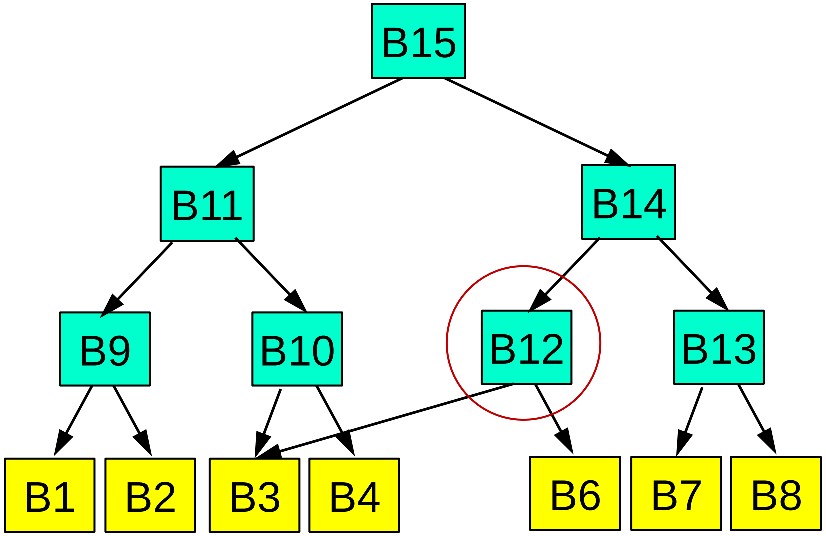
Weighted LRU

Different block weights
- initial metadata block weight: 3
- initial data block weight: 1

Reclamation
- evict 0-weight blocks
- reduce all weights by 1



0	1	2	3
B3	B9	B11	B15
B4		B10	B14
			B12



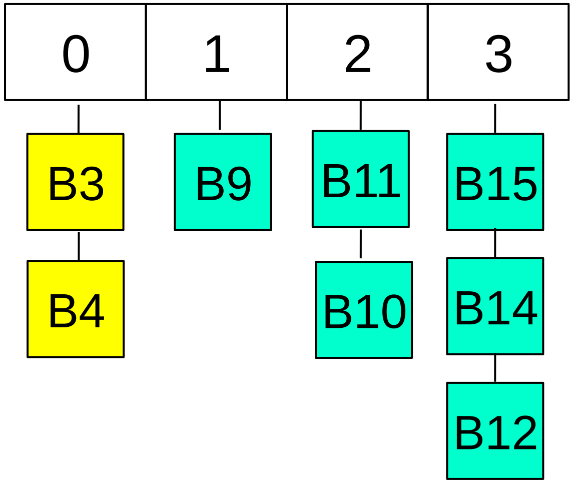
Weighted LRU

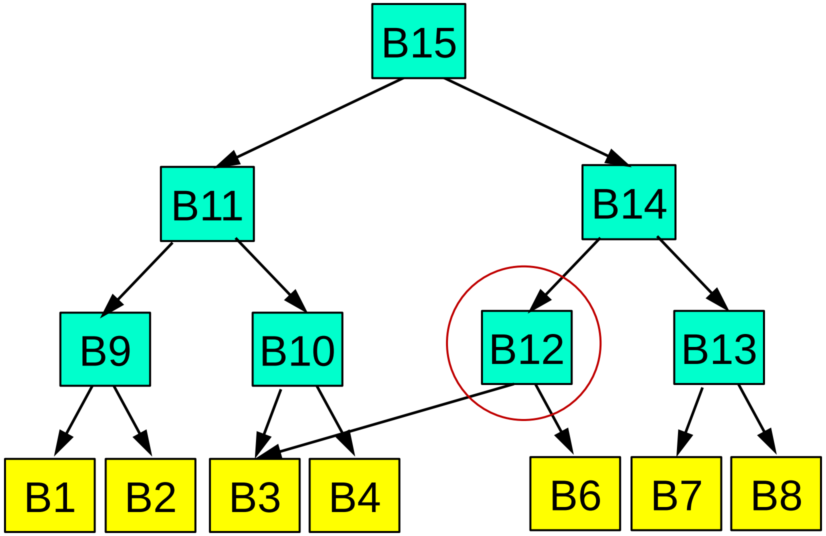
Different block weights

- initial metadata block weight: 3
- initial data block weight: 1

Reclamation

- evict 0-weight blocks
- reduce all weights by 1





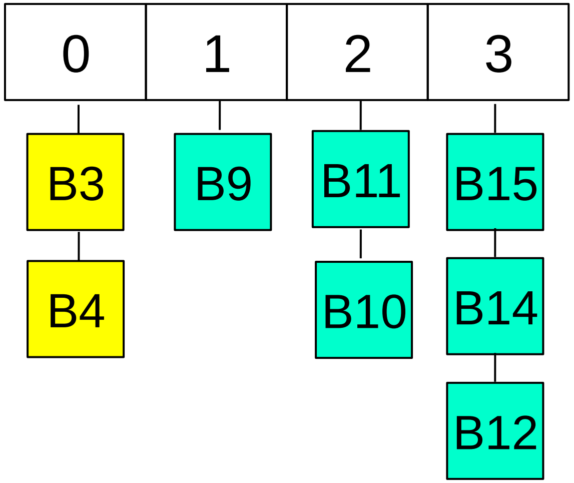
Weighted LRU

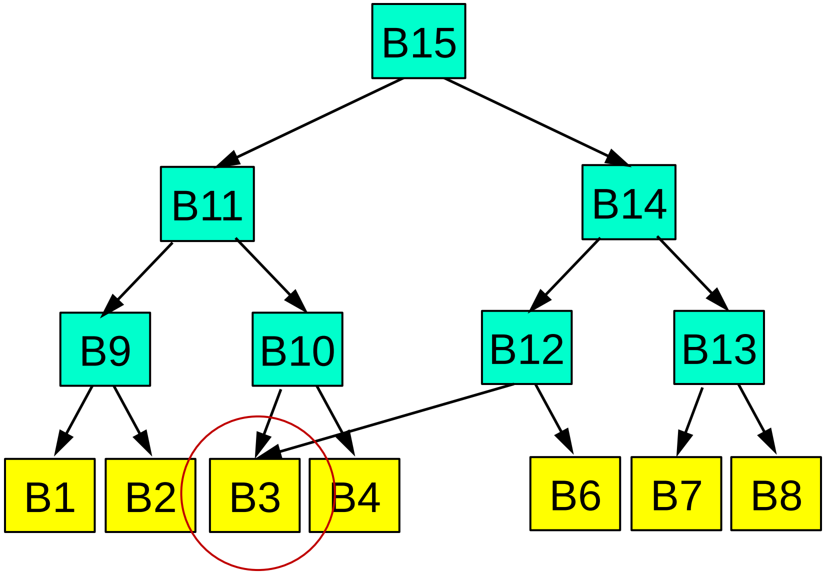
Different block weights

- initial metadata block weight: 3
- initial data block weight: 1

Reclamation

- evict 0-weight blocks
- reduce all weights by 1





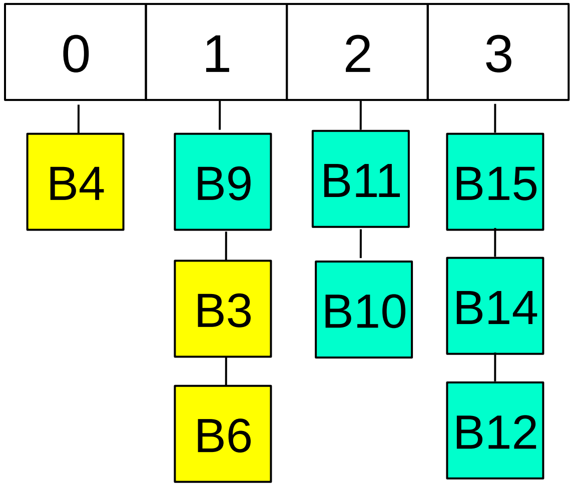
Weighted LRU

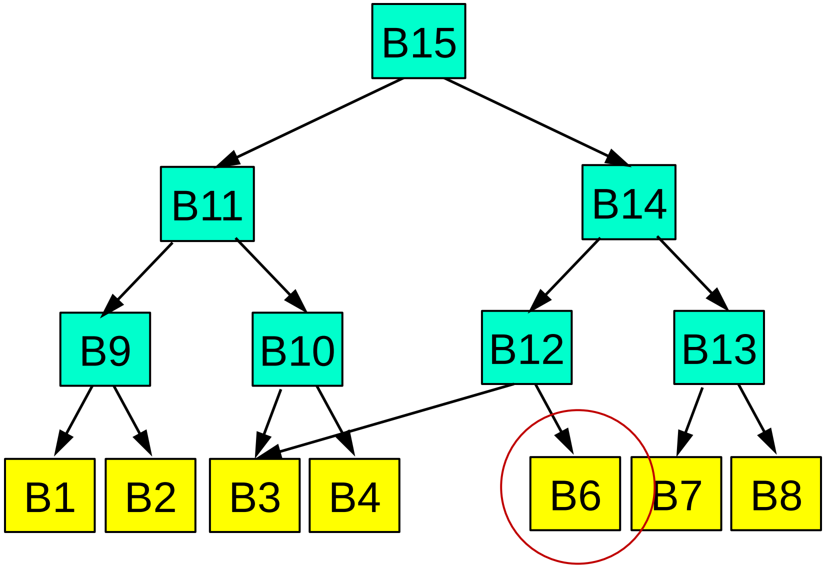
Different block weights

- initial metadata block weight: 3
- initial data block weight: 1

Reclamation

- evict 0-weight blocks
- reduce all weights by 1





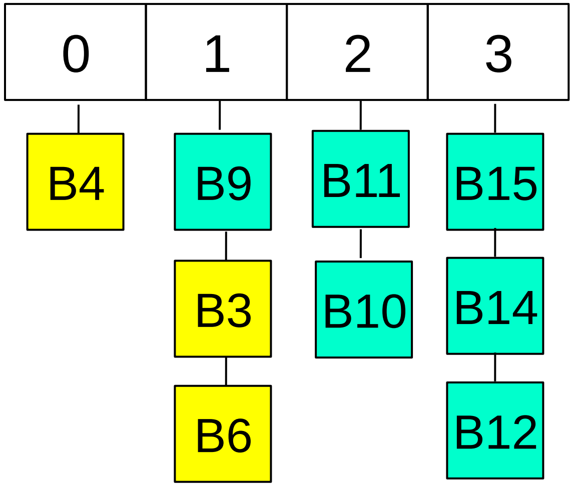
Weighted LRU

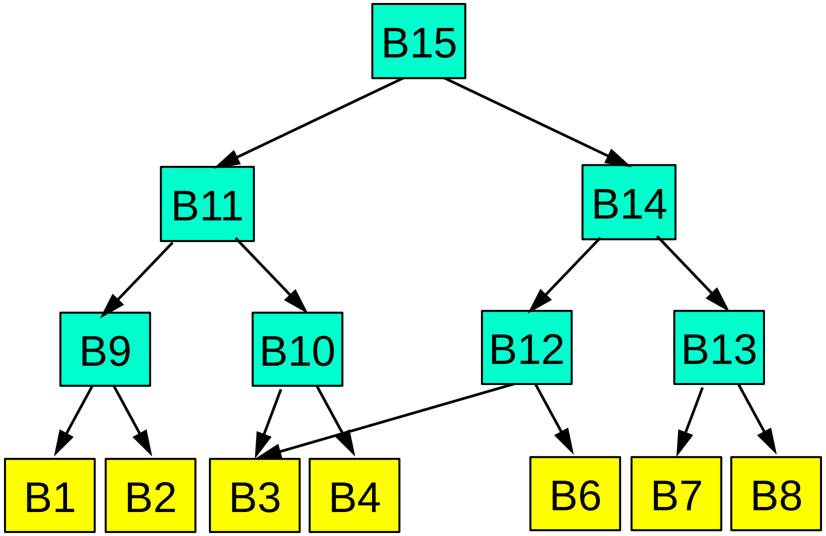
Different block weights

- initial metadata block weight: 3
- initial data block weight: 1

Reclamation

- evict 0-weight blocks
- reduce all weights by 1





Reclamation !

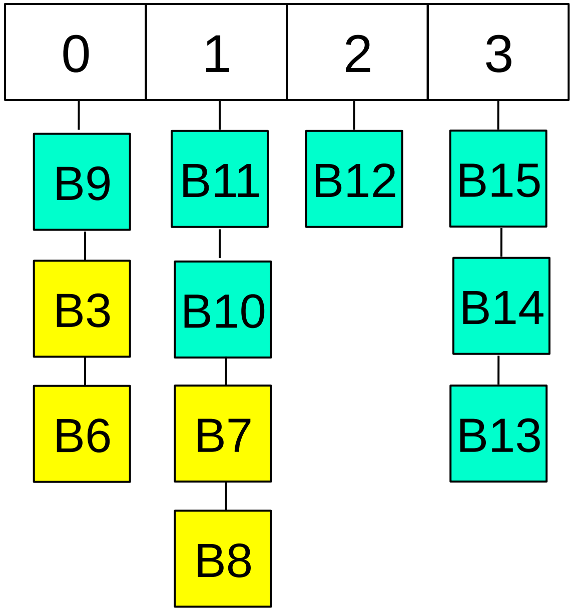
Weighted LRU

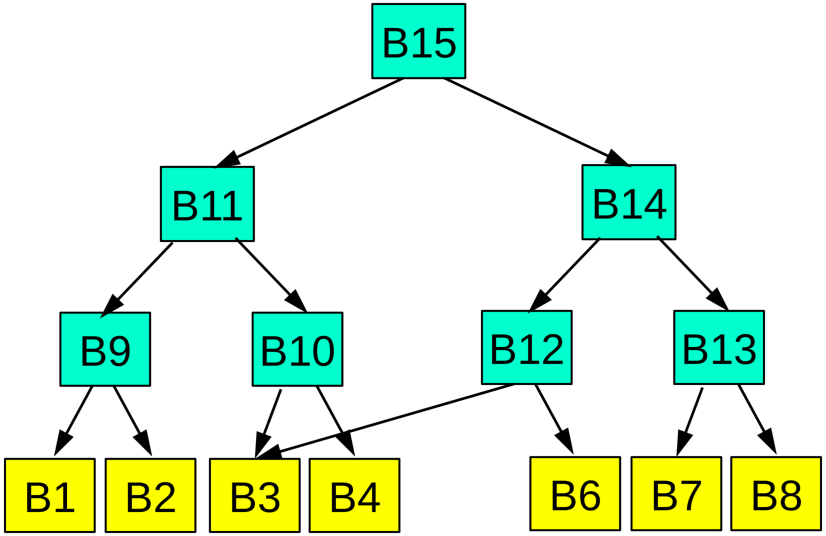
Different block weights

- initial metadata block weight: 3
- initial data block weight: 1

Reclamation

- evict 0-weight blocks
- reduce all weights by 1





Effectiveness of Read Path Optimizations

- Main techniques
  - Pre-fetch metadata more aggressively than data
  - Weighted-LRU to evict data more aggressively than metadata

- Experiment
  - Read a large file

	Accesses	Misses		Throughput (MB/s)
		Data	Metadata	
Base	486,966	1577	1011	134.3
Optimized	211,632	438	945	183.2

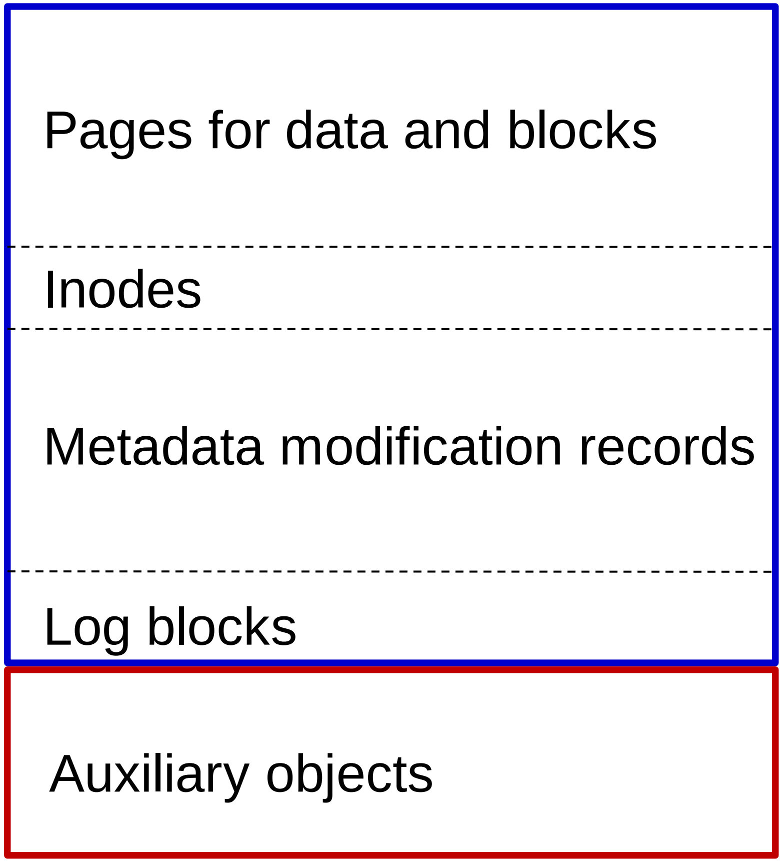
Resource Management

Pre-allocated, managed

- Fixed-size pools of fixed-size objects
  - – pages are 4 KB
  - – inodes are 8 KB
  - – log blocks are up to 128 KB
  - – etc.

Unmanaged heap

- Objects' number is bound by that of some managed objects



Pages for data and blocks

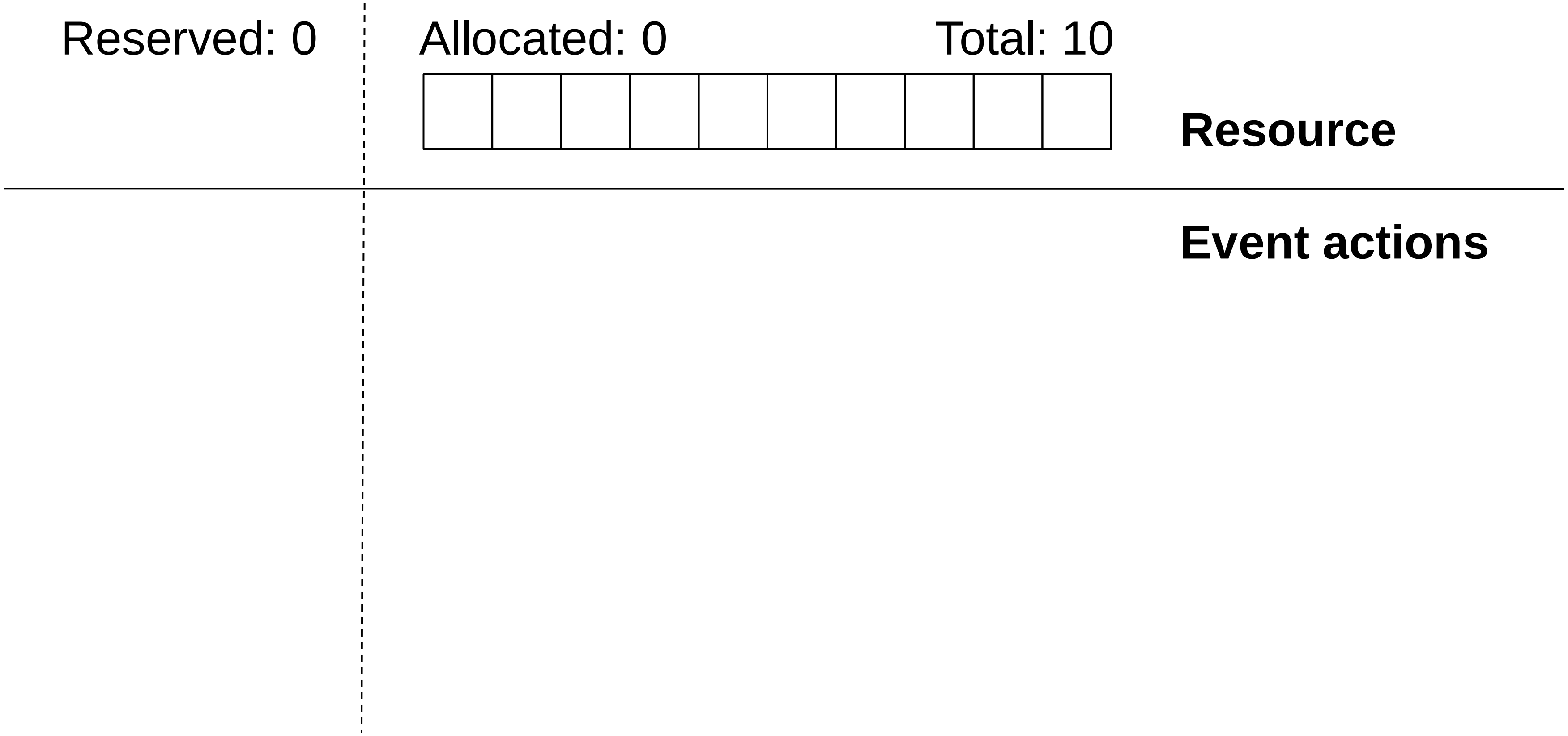
Inodes

Metadata modification records

Log blocks

Auxiliary objects

Resource Management



Reserved: 0

Allocated: 0

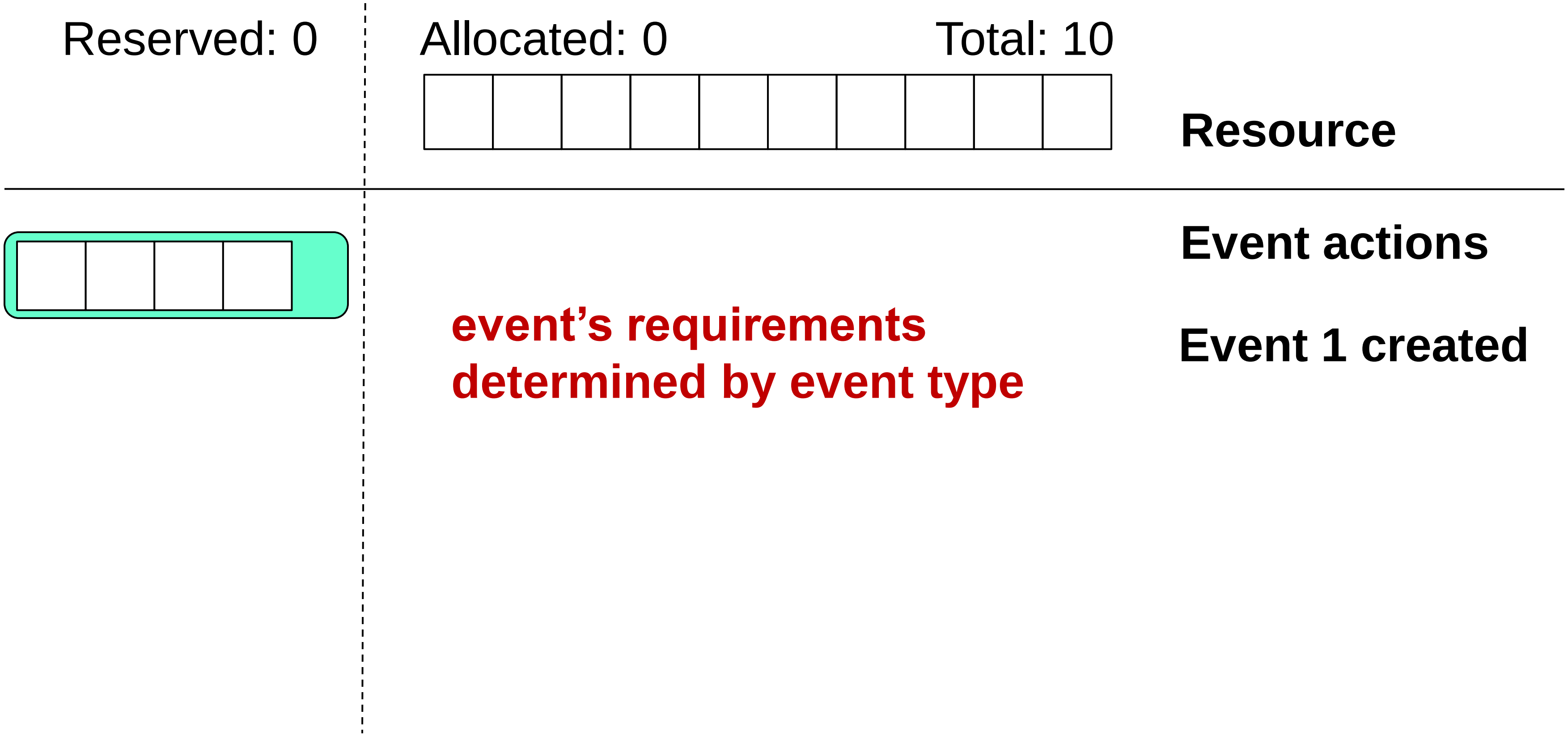
Total: 10

Resource

Event actions

Admission condition: **Requested + Reserved + Allocated ≤ Total**

Resource Management



Reserved: 0

Allocated: 0

Total: 10

Resource

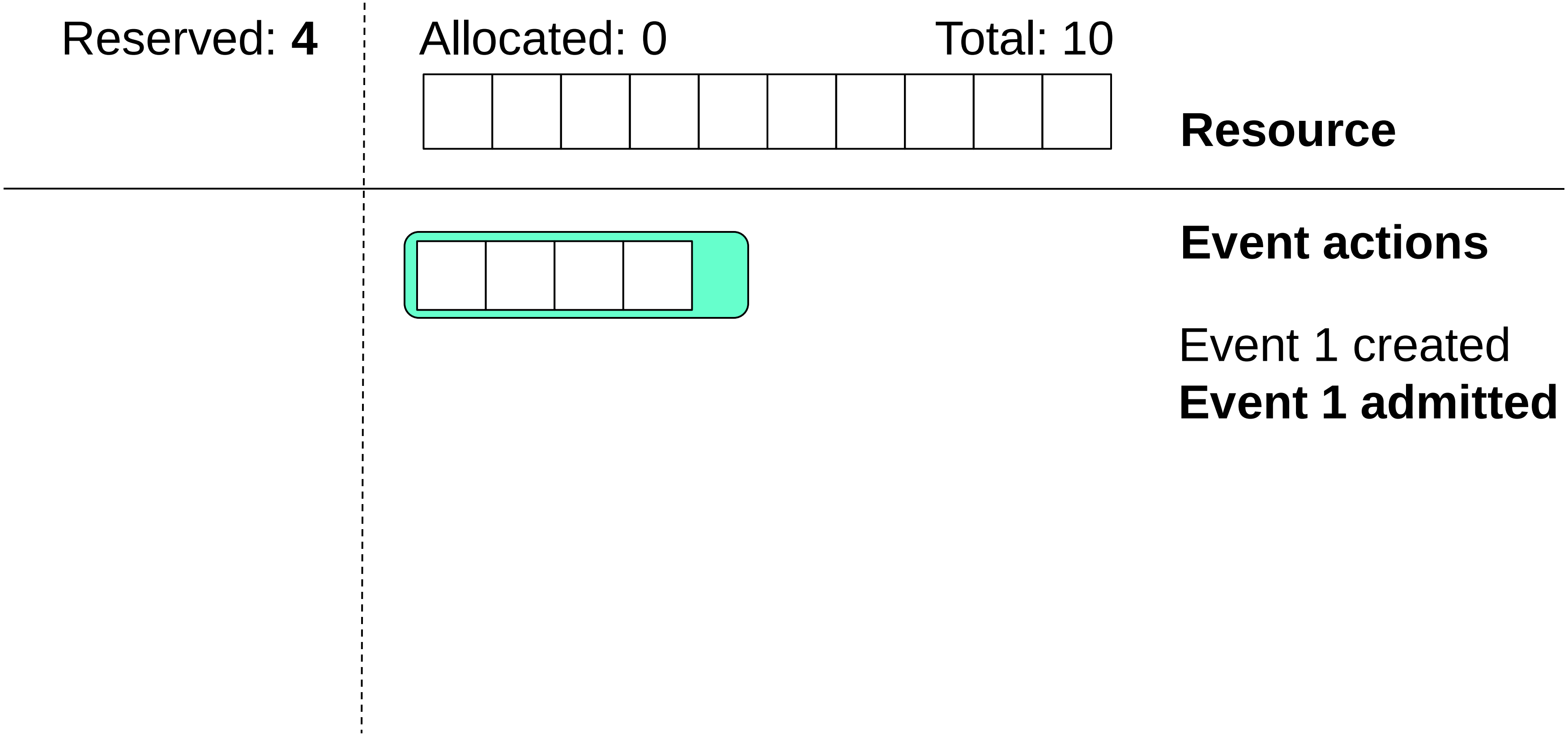
Event actions

event's requirements determined by event type

Event 1 created

4 + 0 + 0 ≤ 10

Resource Management



Reserved: **4**

Allocated: 0

Total: 10

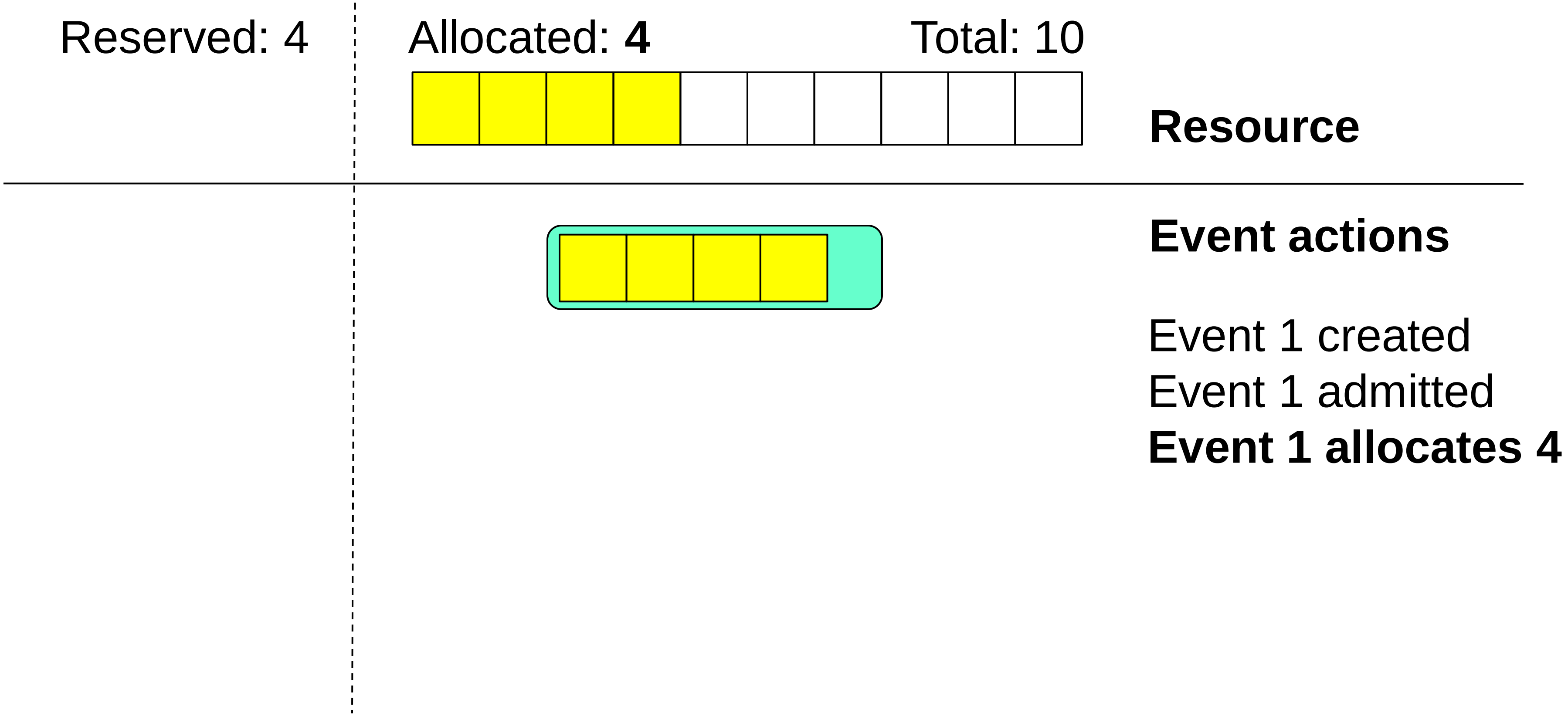
Resource

Event actions

Event 1 created

Event 1 admitted

Resource Management



Reserved: 4

Allocated: **4**

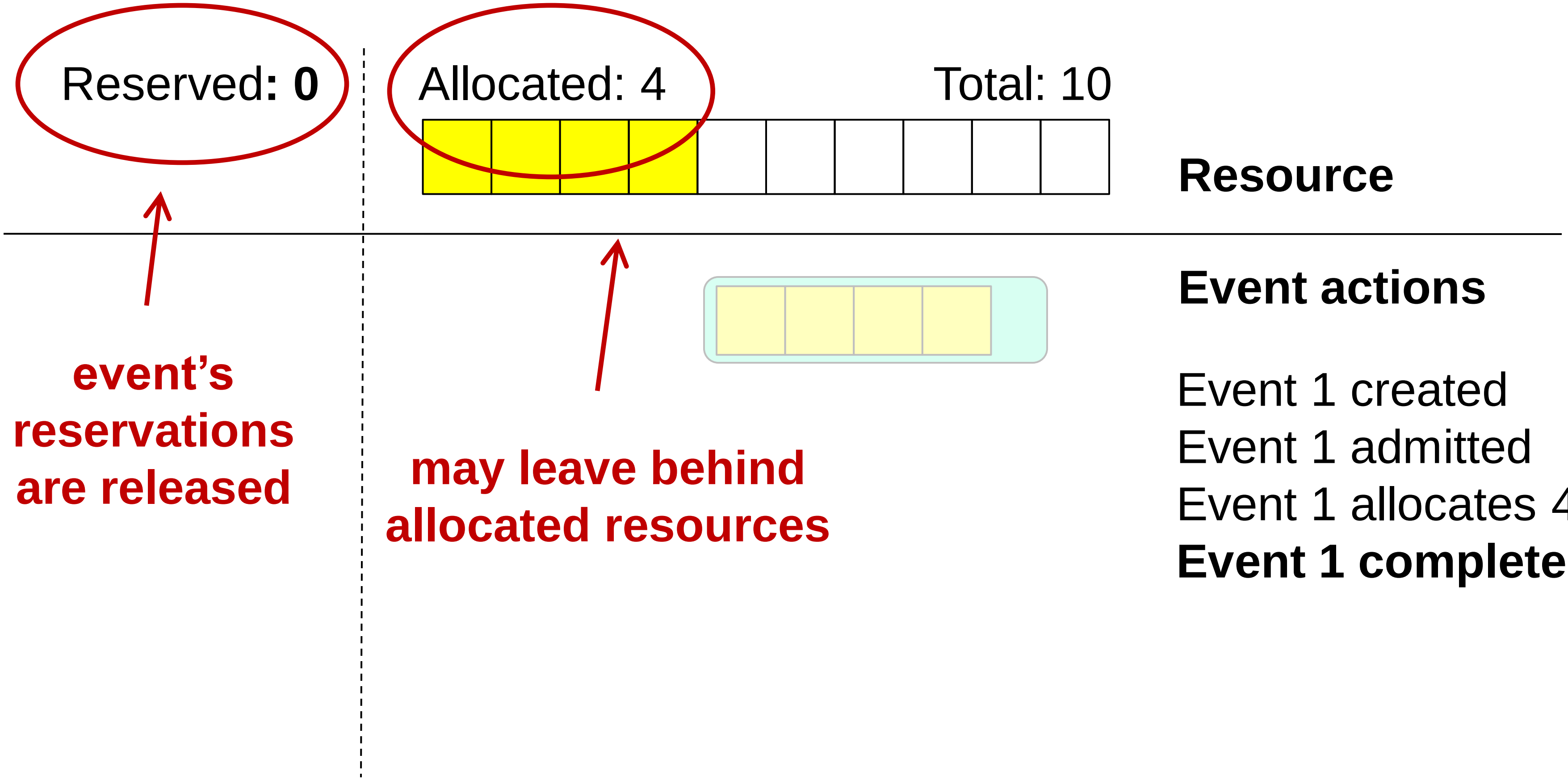
Total: 10

Resource

Event actions

Event 1 created
Event 1 admitted
Event 1 allocates 4

Resource Management



Reserved: **0**

Allocated: 4

Total: 10

Resource

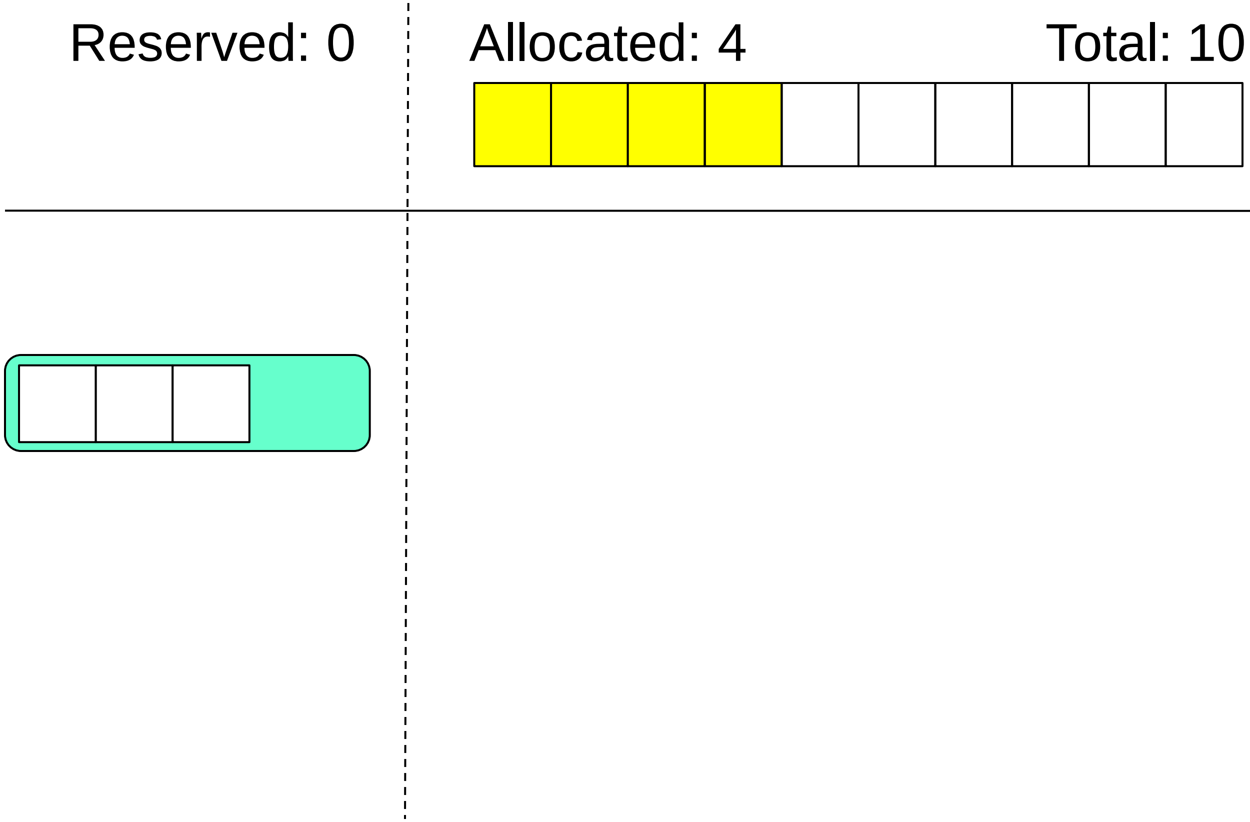
Event actions

Event 1 created
Event 1 admitted
Event 1 allocates 4
Event 1 completes

event's reservations are released

may leave behind allocated resources

Resource Management



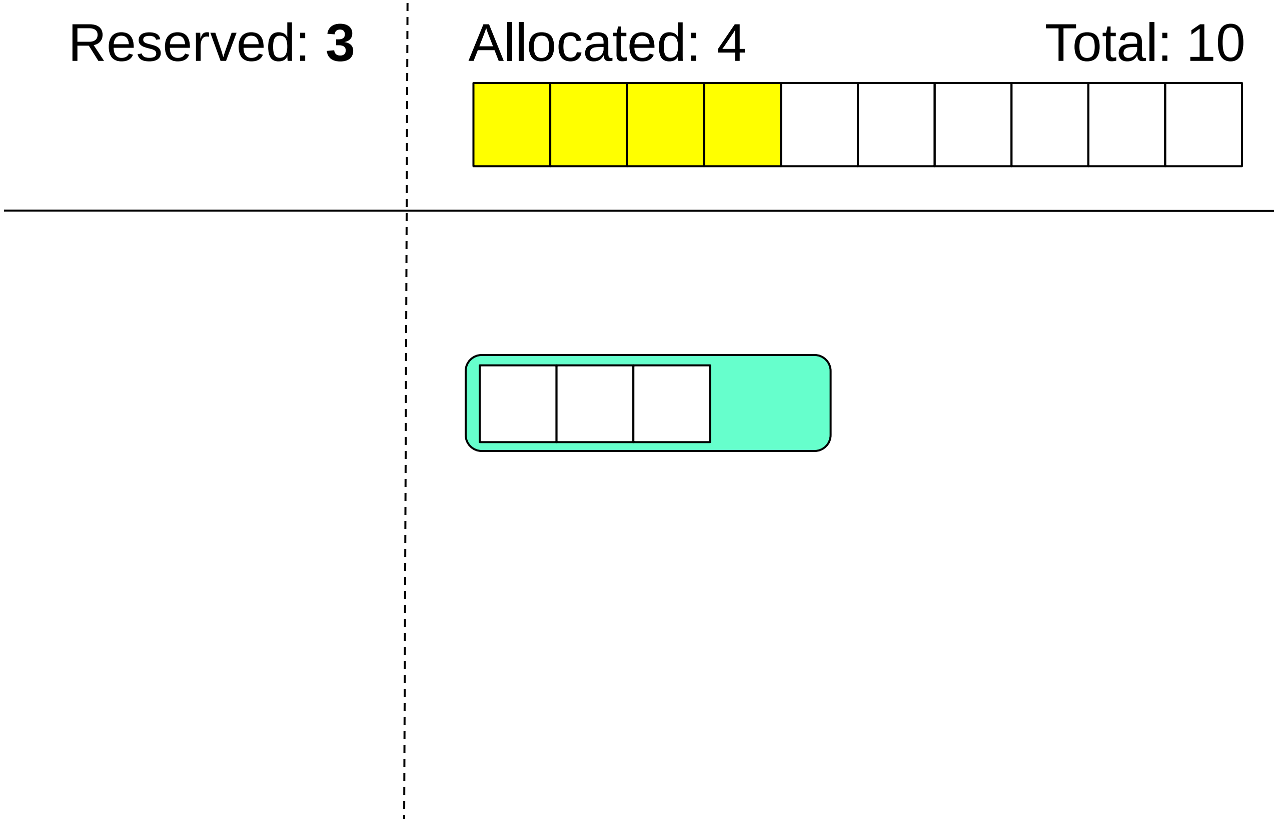
Resource

Event actions

Event 1 created
Event 1 admitted
Event 1 allocates 4
Event 1 completes
Event 2 created

3 + 0 + 4 ≤ 10

Resource Management



Reserved: **3**

Allocated: 4

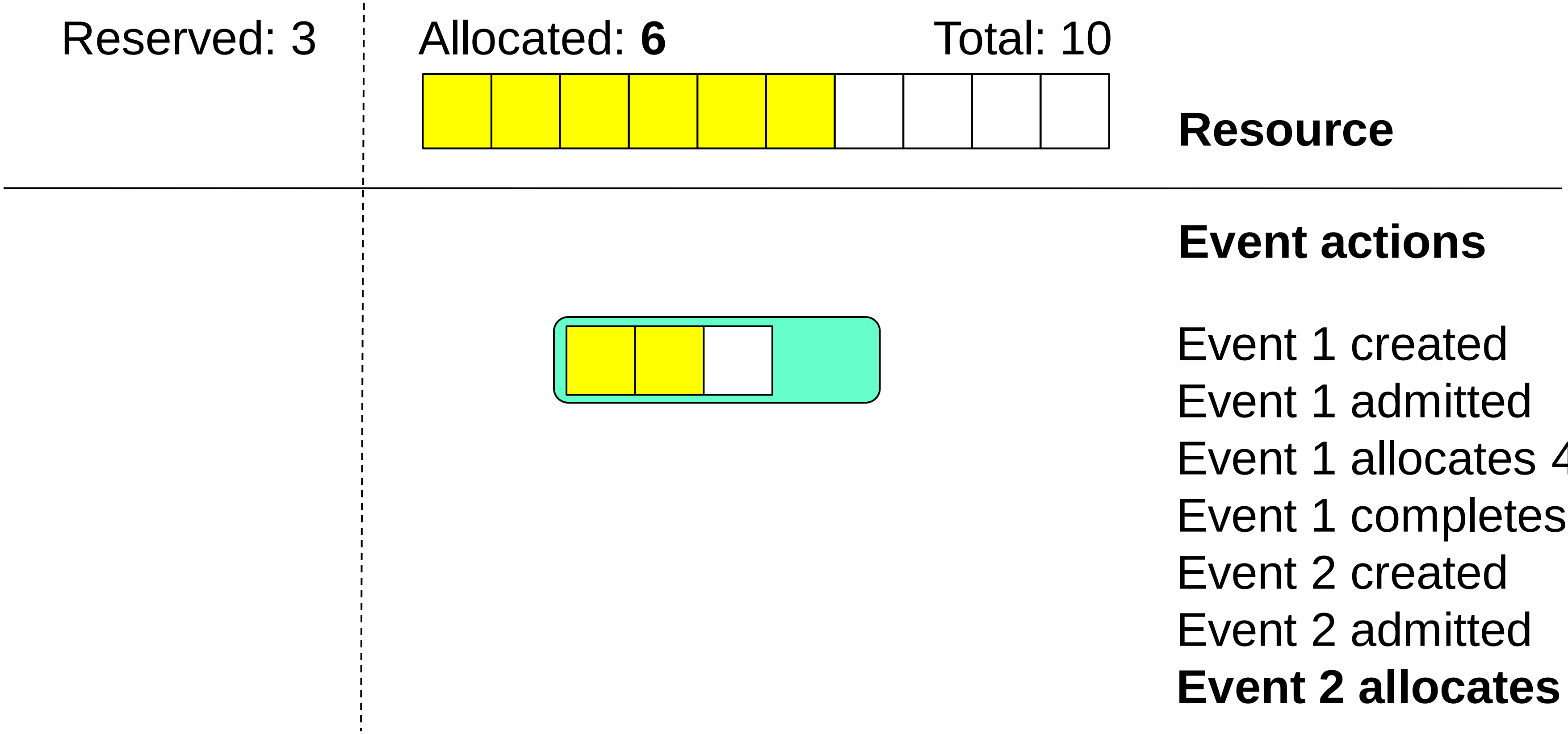
Total: 10

Resource

Event actions

Event 1 created
Event 1 admitted
Event 1 allocates 4
Event 1 completes
Event 2 created
Event 2 admitted

Resource Management

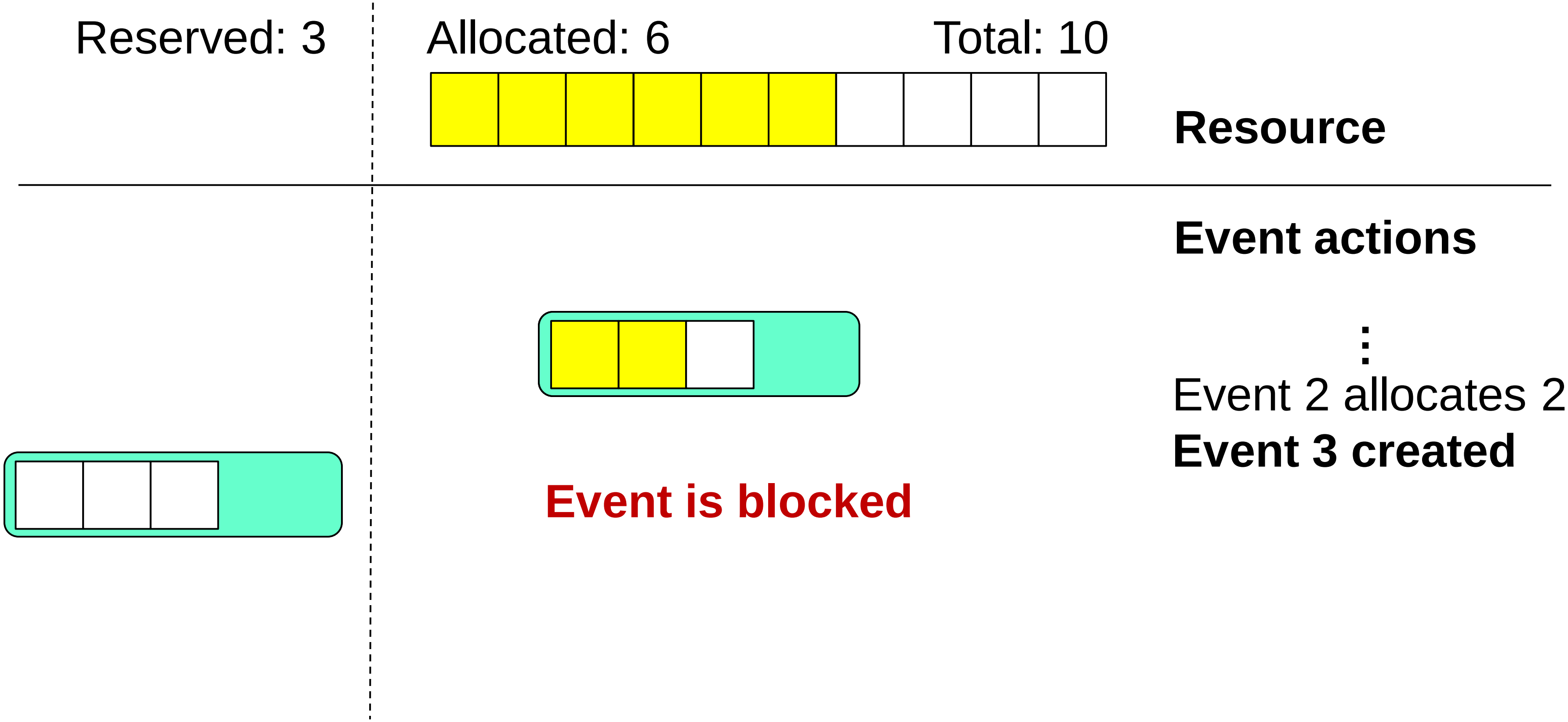


Resource

Event actions

Event 1 created
Event 1 admitted
Event 1 allocates 4
Event 1 completes
Event 2 created
Event 2 admitted
Event 2 allocates 2

Resource Management



Reserved: 3

Allocated: 6

Total: 10

Resource

Event actions

⋮

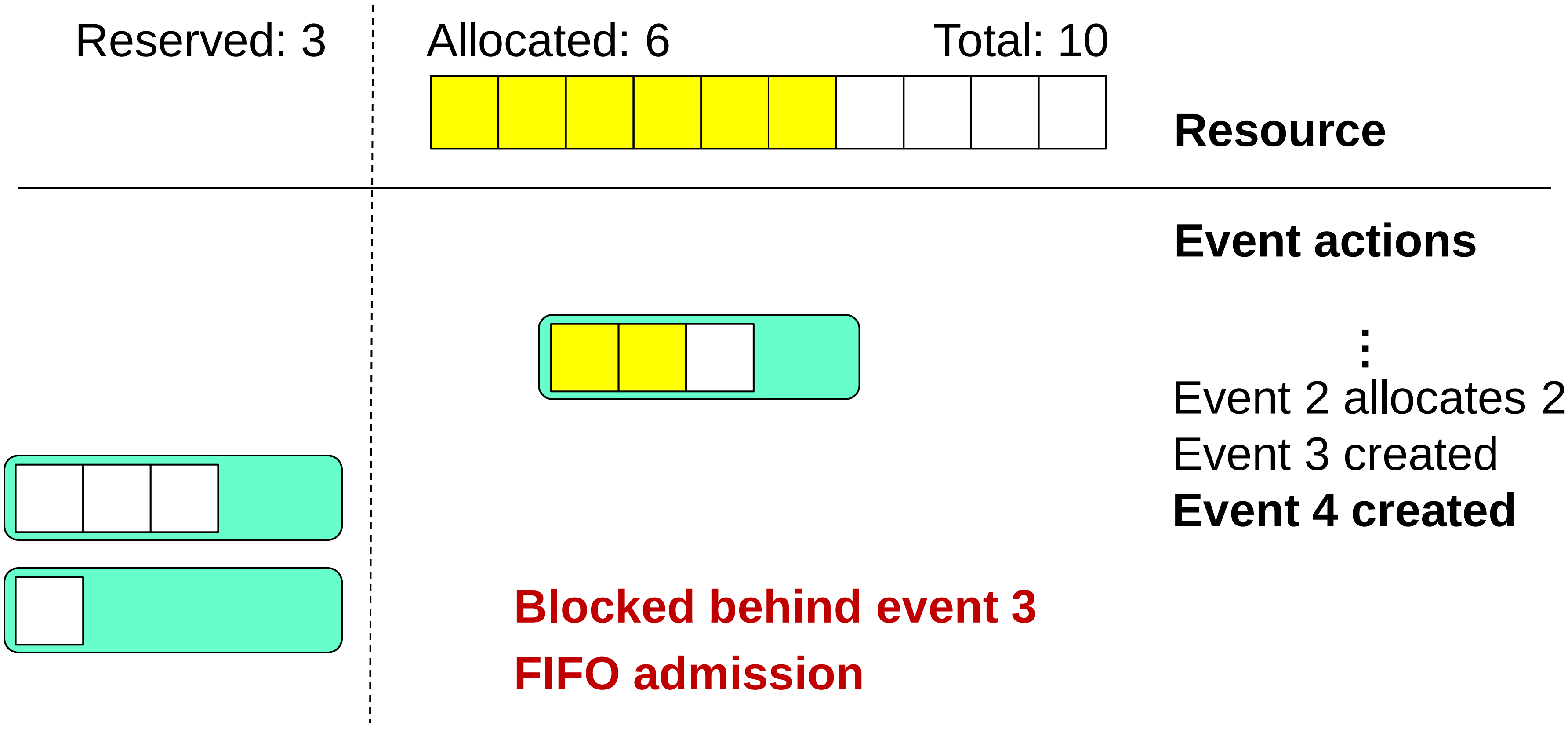
Event 2 allocates 2

Event 3 created

Event is blocked

3 + 3 + 6 > 10

Resource Management



Reserved: 3

Allocated: 6

Total: 10

Resource

Event actions

⋮

Event 2 allocates 2

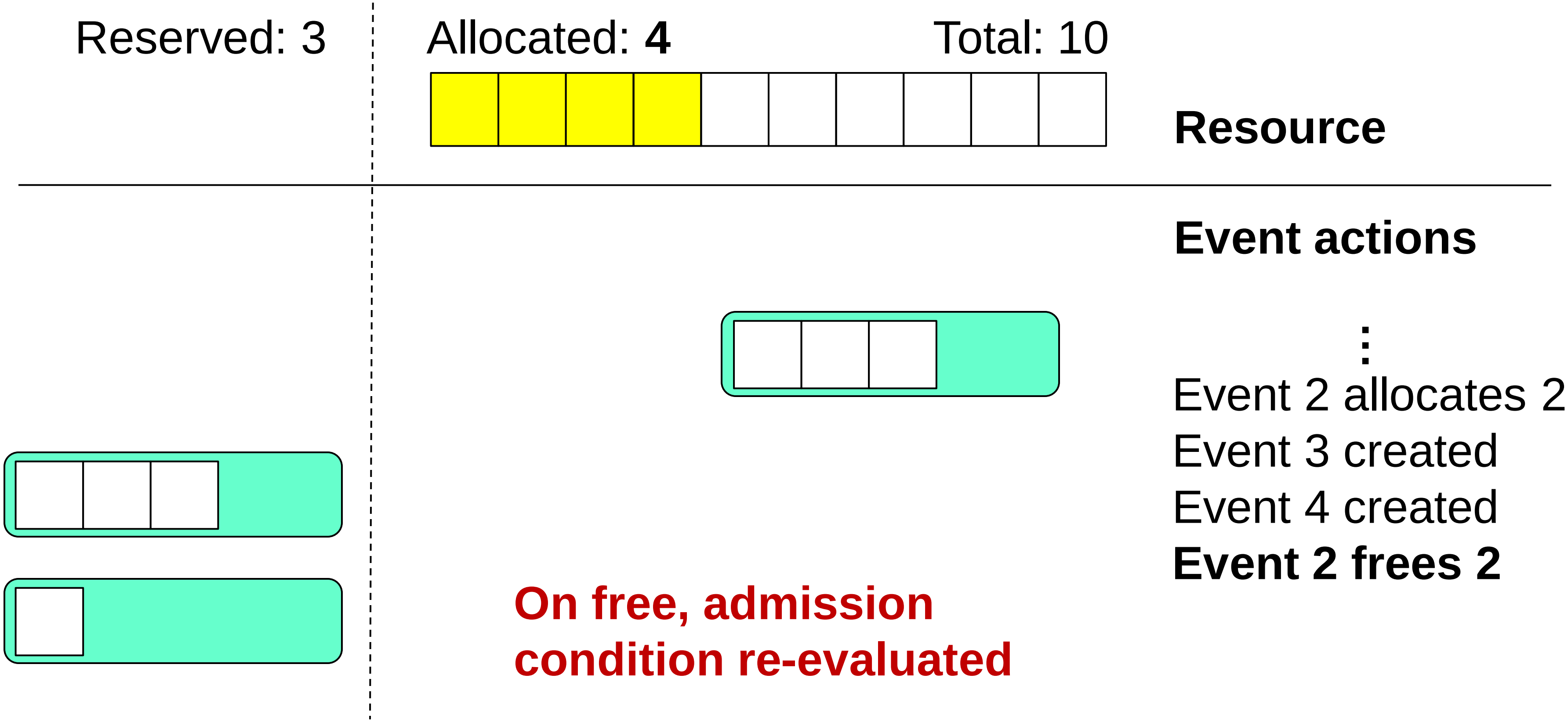
Event 3 created

Event 4 created

Blocked behind event 3

FIFO admission

Resource Management



Reserved: 3

Allocated: **4**

Total: 10

Resource

Event actions

⋮

Event 2 allocates 2

Event 3 created

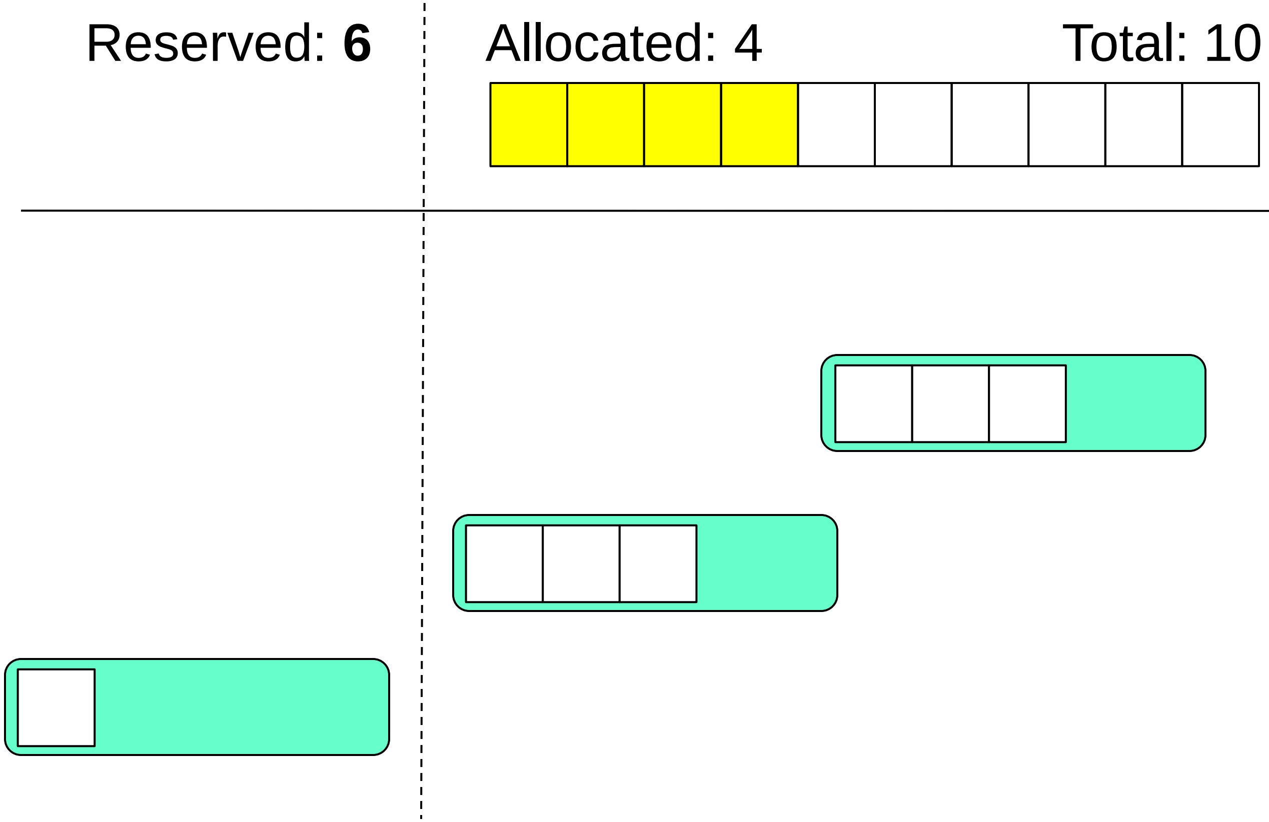
Event 4 created

Event 2 frees 2

On free, admission condition re-evaluated

3 + 3 + 4 ≤ 10

Resource Management



Reserved: **6**

Allocated: 4

Total: 10

Resource

Event actions

⋮

Event 2 allocates 2

Event 3 created

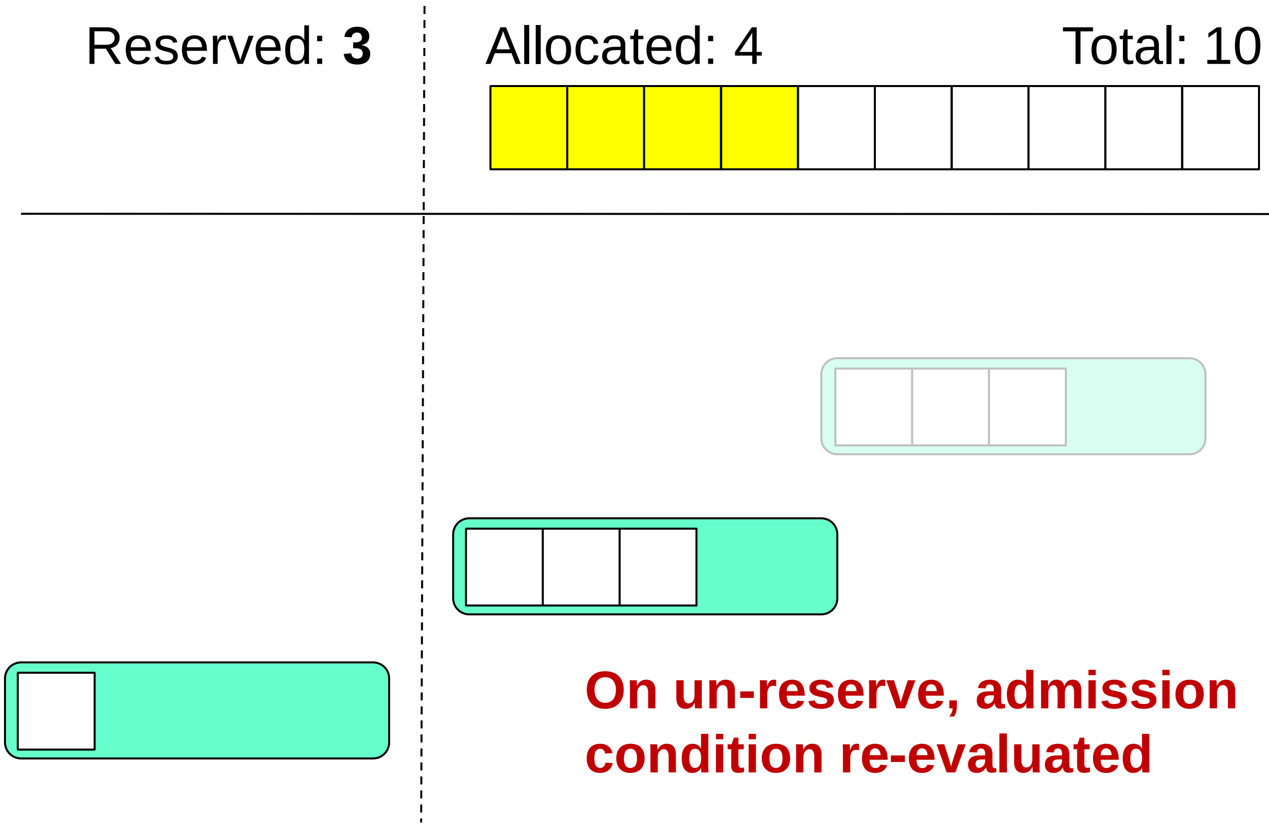
Event 4 created

Event 2 frees 2

Event 3 admitted

3 + 6 + 4 > 10 **event 4 remains blocked**

Resource Management



Reserved: **3**

Allocated: 4

Total: 10

Resource

Event actions

⋮

Event 2 allocates 2

Event 3 created

Event 4 created

Event 2 frees 2

Event 3 admitted

Event 2 completes

On un-reserve, admission condition re-evaluated

1 + 3 + 4 ≤ 10

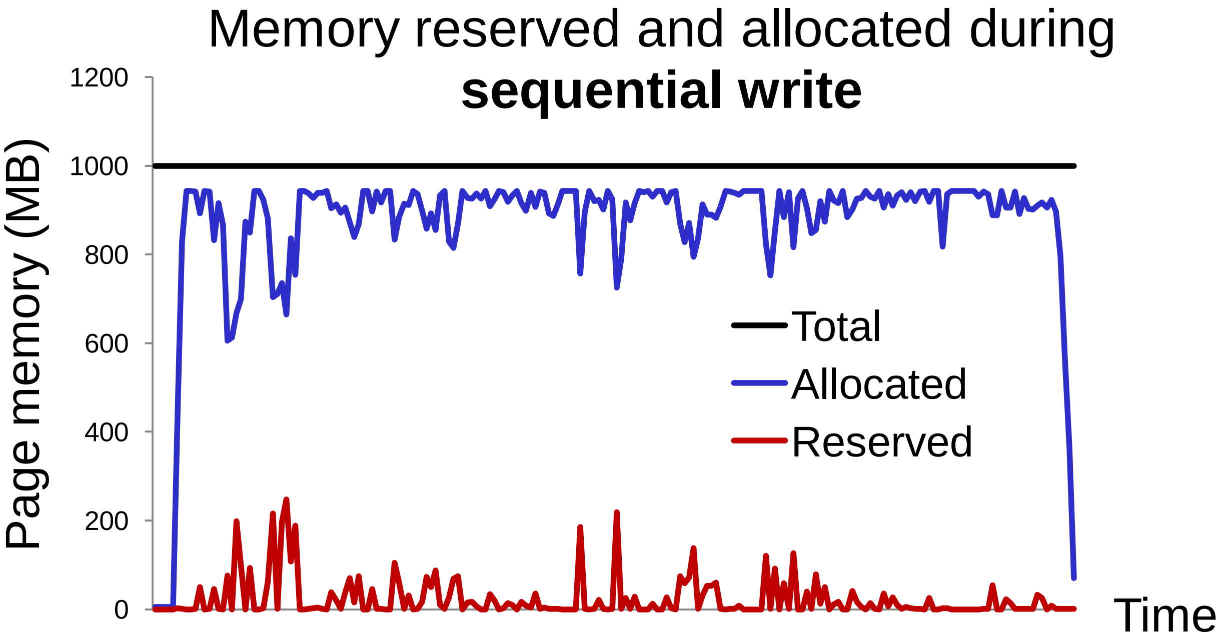
event 4 admitted …

Resource Management - Reclamation -

- Reclamation processing
  - First, free pages from clean cached blocks
  - If not sufficient, initiate **flush of dirty inodes**
  - Flush is an internal event with pre-reserved resources
- Reclamation initiated when
  - An event is blocked
  - A **threshold** is reached
- Threshold limit depends on resource type
  - Metadata modification records can only be cleaned through metadata update → start earlier
  - Others (pages, log blocks) can be cleaned quicker → start later

Resource Management

- Limits the amount of memory used (avoid swapping)
- Avoids handling allocation failures in the middle of event processing
- Avoids event starvation through FIFO processing
- Simple but effective (allows high utilization of resources)



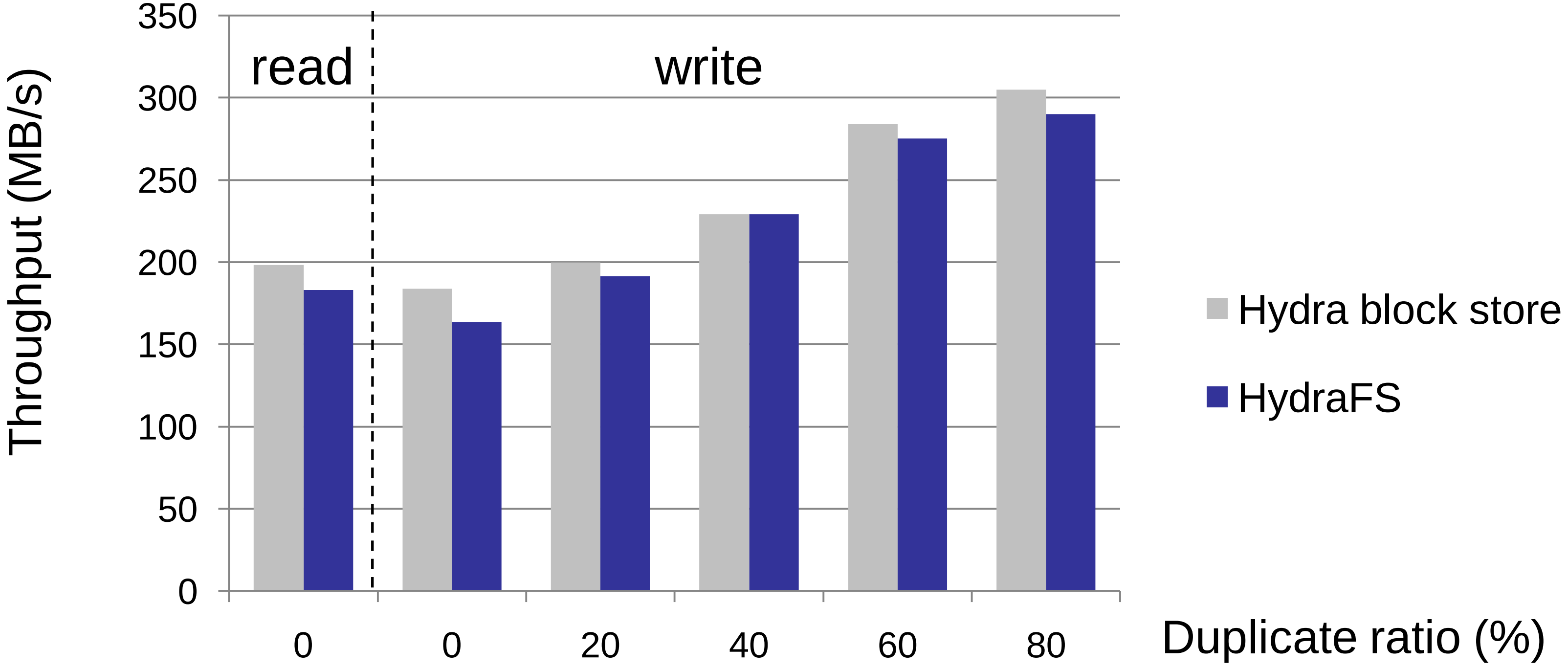
Experiments

Flow control API

- Requests can be rejected (“system busy”)
- Clients notified to resume submission

Tool

- Submit requests until busy, resumes as soon as notified
- Maximum concurrency; No parent-child structures
- Upper limit of performance



Conclusions and Future Work

Conclusions

- Building a filesystem for a content-addressable storage system with content-defined chunking poses interesting challenges
- A small number of techniques was sufficient to overcome them while keeping the system relatively simple and achieving high throughput

Future work

- Distribute the filesystem
- Use SSD to improve performance for metadata intensive workloads

Thank you!