How to Copy Files?

Shared by Yiduo Wang, Daniel Shao

2020/05/08

When we need *Copy*?

Typical Scenarios:

- Backup, Snapshots
- Create new VM images
- Docker, Container
- ……

When we need *Copy*?

Typical Scenarios:

- Backup, Snapshots
- Create new VM images
- Docker, Container
- ……

Copy is Imporant & Ubiquitous!

Different kinds of copy

Physical Copy: cp –r
Copy Data&Metadata Modifying source file will **not** change copy

Soft Link: ln –s
A file of path pointer Modifying source file **will** change copy

Hard Link: ln
Pointer of inode Modifying source file **will** change copy

What is logical Copy?

USTC, CHINA ADSLAB

Physical Copy: cp –r
Copy Data&Metadata Modifying source file will **not** change copy

Soft Link: ln –s
A file of path pointer Modifying source file **will** change copy

Hard Link: ln
Pointer of inode Modifying source file **will** change copy

Logical Copy: cp --reflink
Copy Metadata Modifying source file will **not** change copy

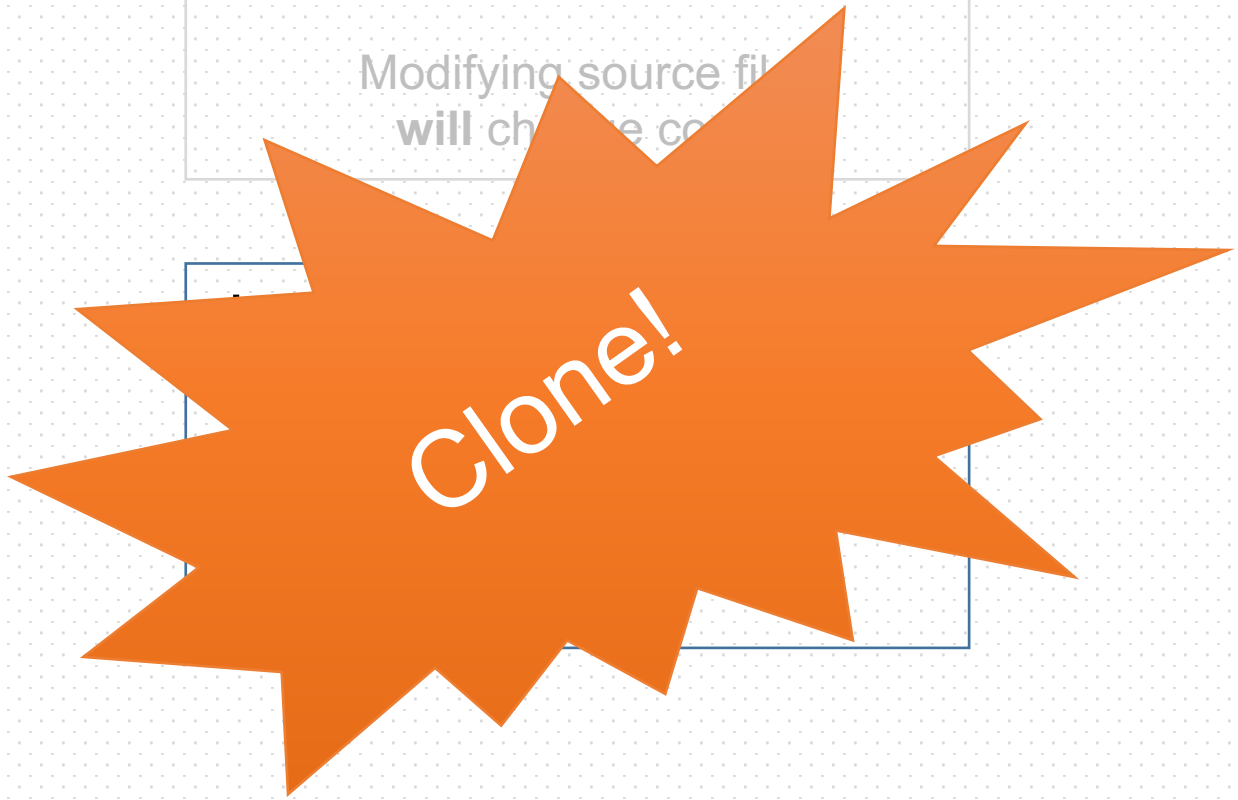
What is logical Copy?

USTC, CHINA ADSLAB

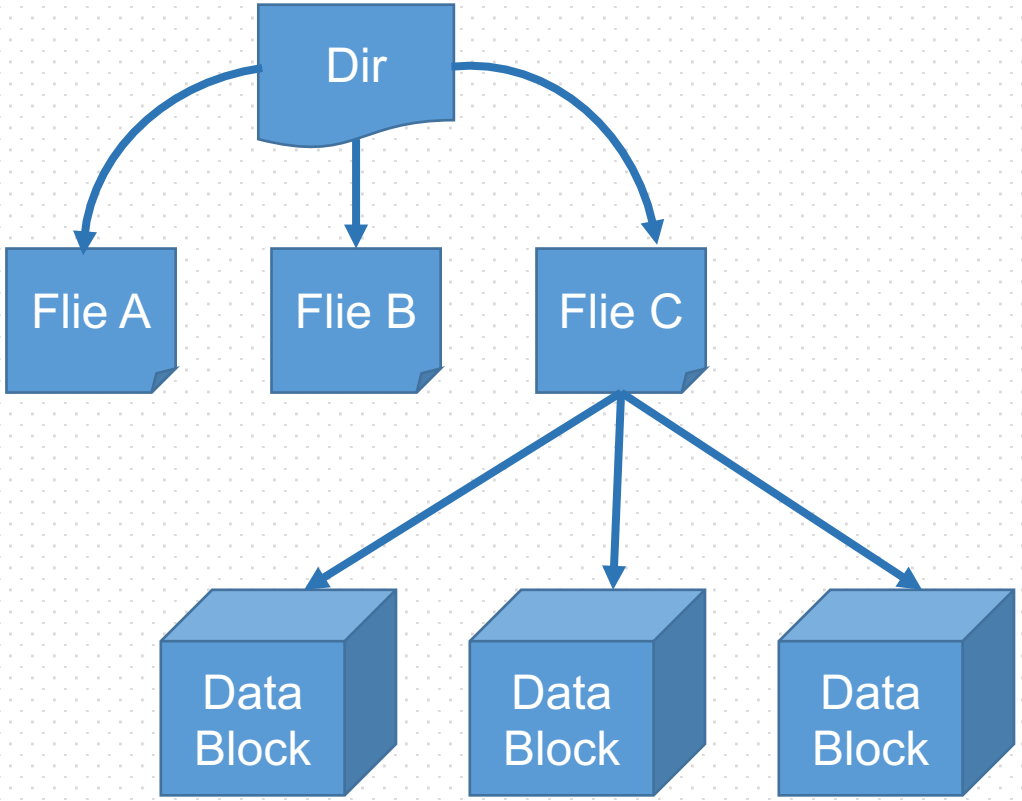
Physical Copy: cp –r
Copy Data&Metadata Modifying source file will **not** change copy

Soft Link: ln –s
A file of path pointer Modifying source fil… **will** ch…e co…

Hard Link: ln
Pointer of inode Modifying source file **will** change copy

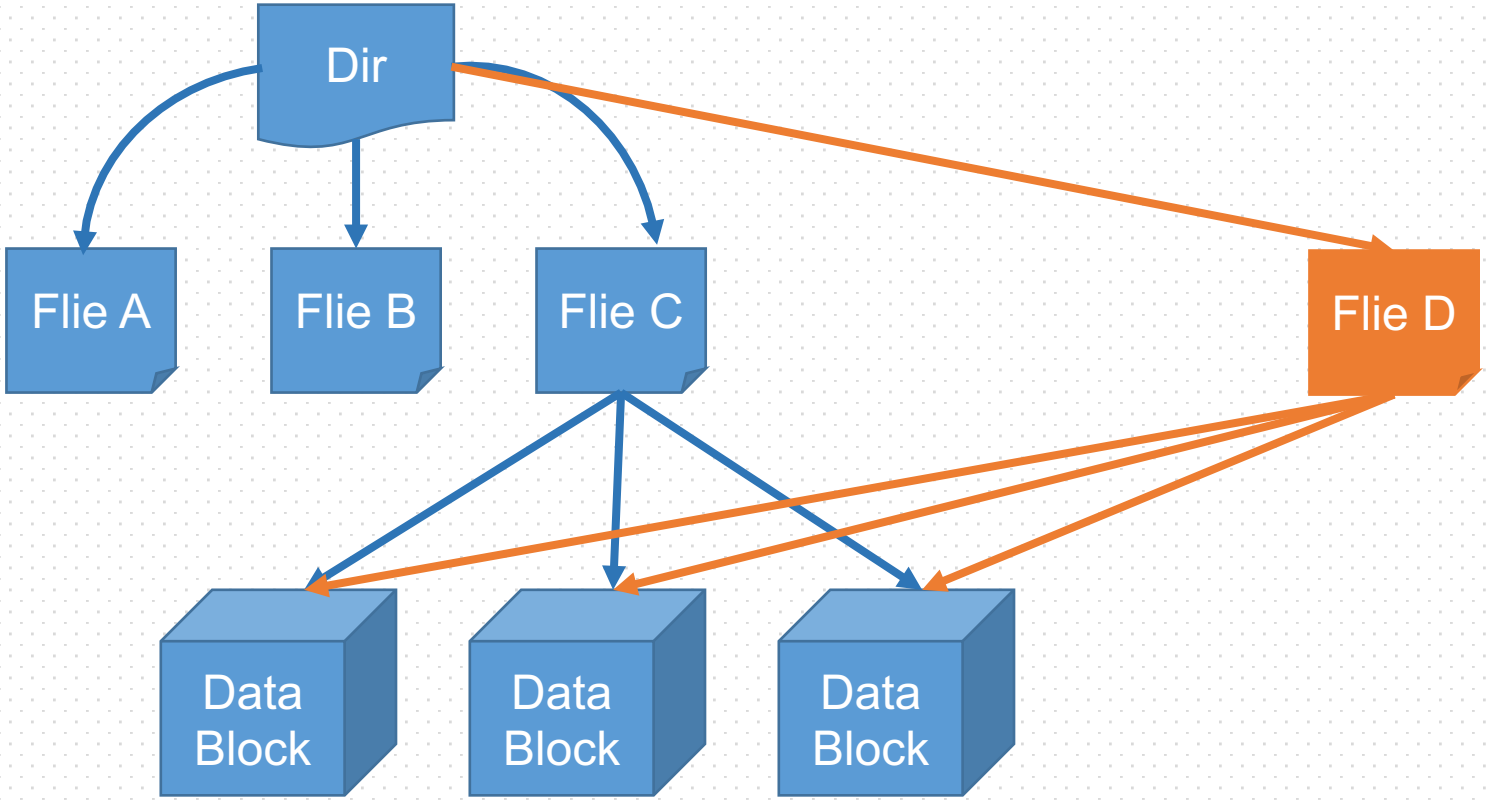


Logical copy process



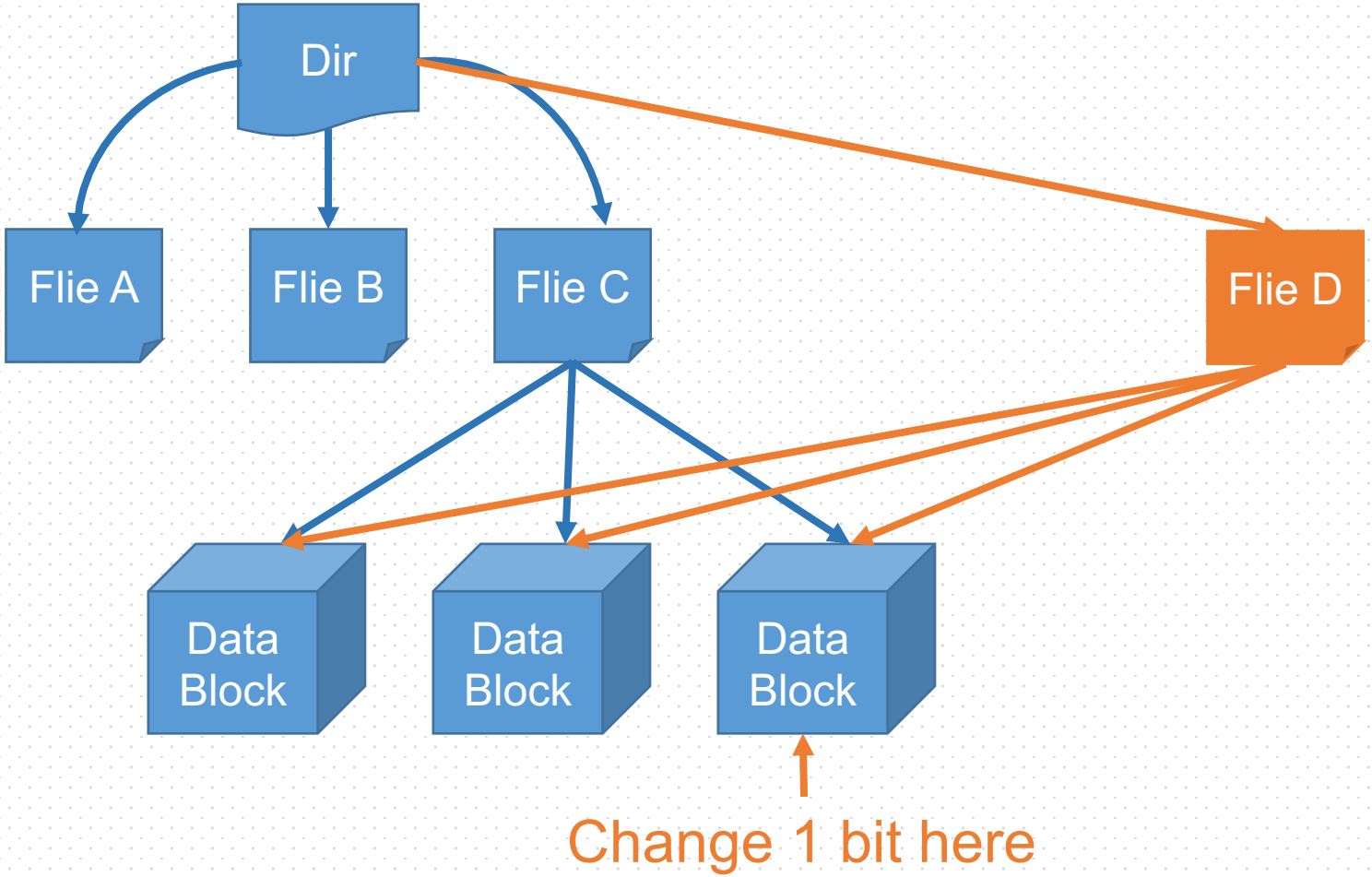
Logical copy process

Copy FileC to FileD



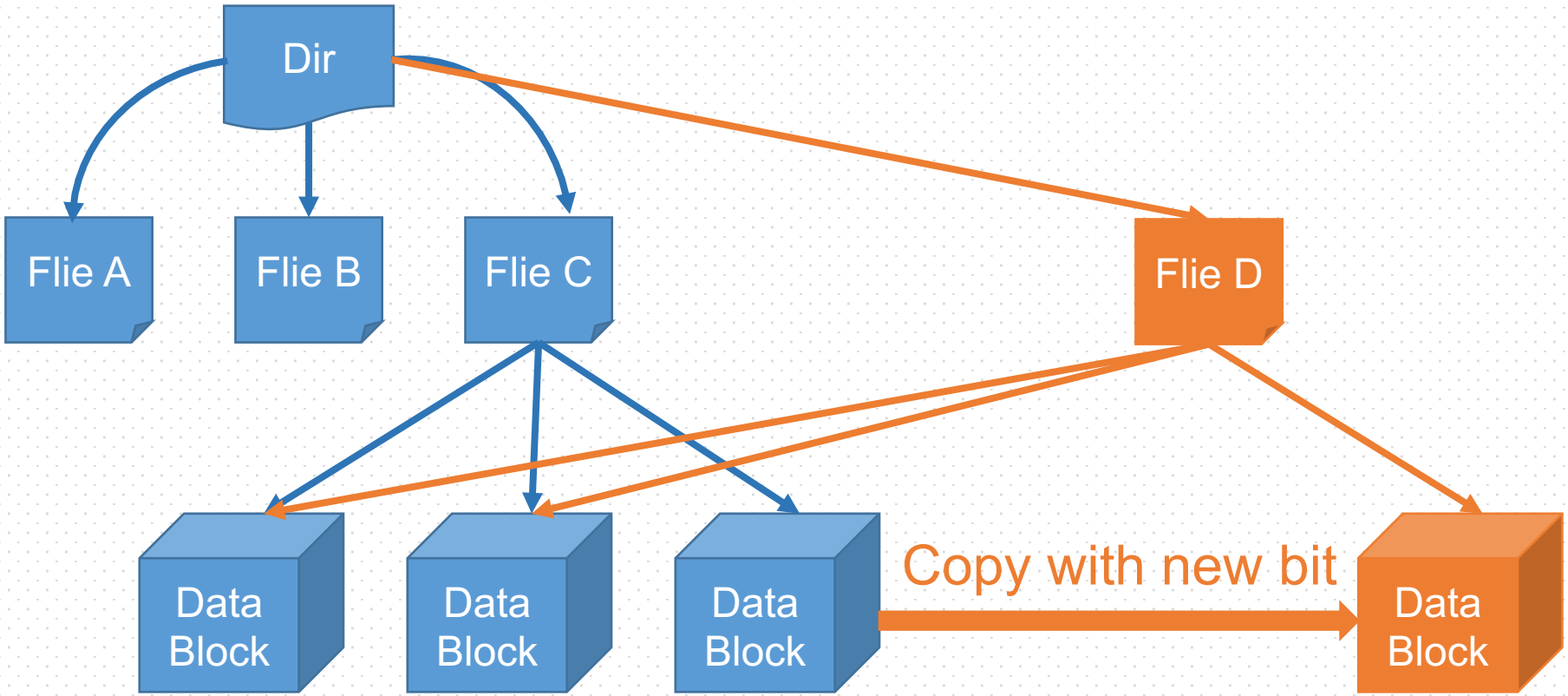
Logical copy process

Modify FileD



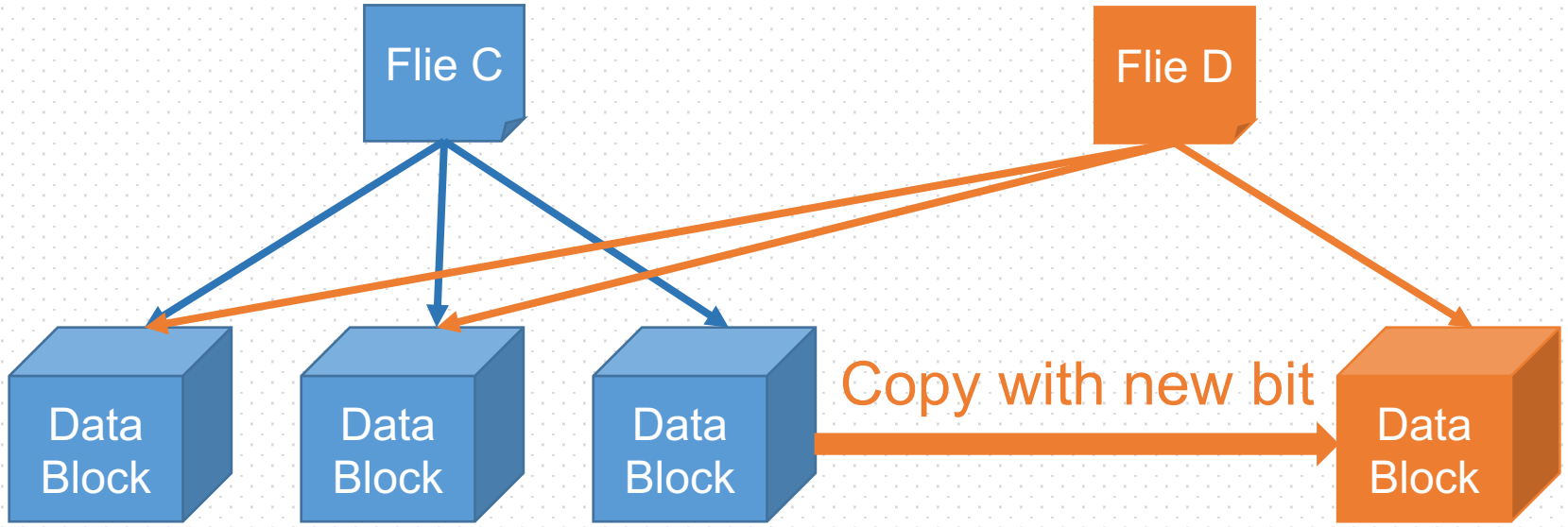
Logical copy process

Modify FileD



Logical copy process

Trade-off



1: Modification will cost at least 1block (No matter how small it is)
2: Fragmentation of data blocks (Especially when the block size is small)

Current Clone Implementation

BtrFS: cp --reflink

- Leverages the underlying copy-on-write B-tree to implement

XFS: cp --reflink

- Uses an update-in-place B-tree but supports sharing data blocks with copy-on-write

ZFS: zfs clone

- Implements a limited version of copy-on-write

Let's test them!

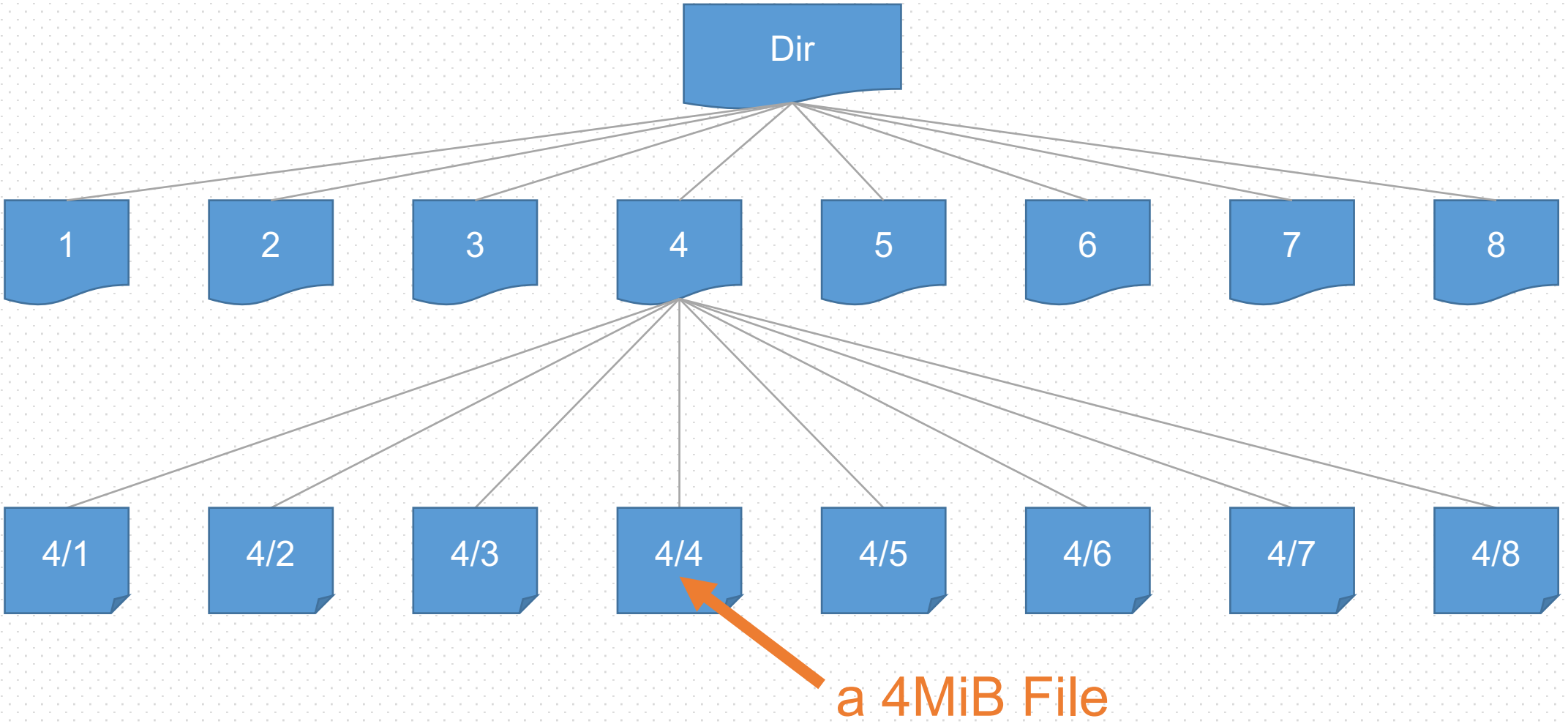
Machine Configuration

- Dell Optiplex 790
- 4-core 3.40 GHz Intel Core i7 CPU
- 500GB, 7200 RPM SATA disk
- 4096-byte block size
- 64-bit Ubuntu 14.04.5

Let's test them!

USTC, CHINA ADSLAB

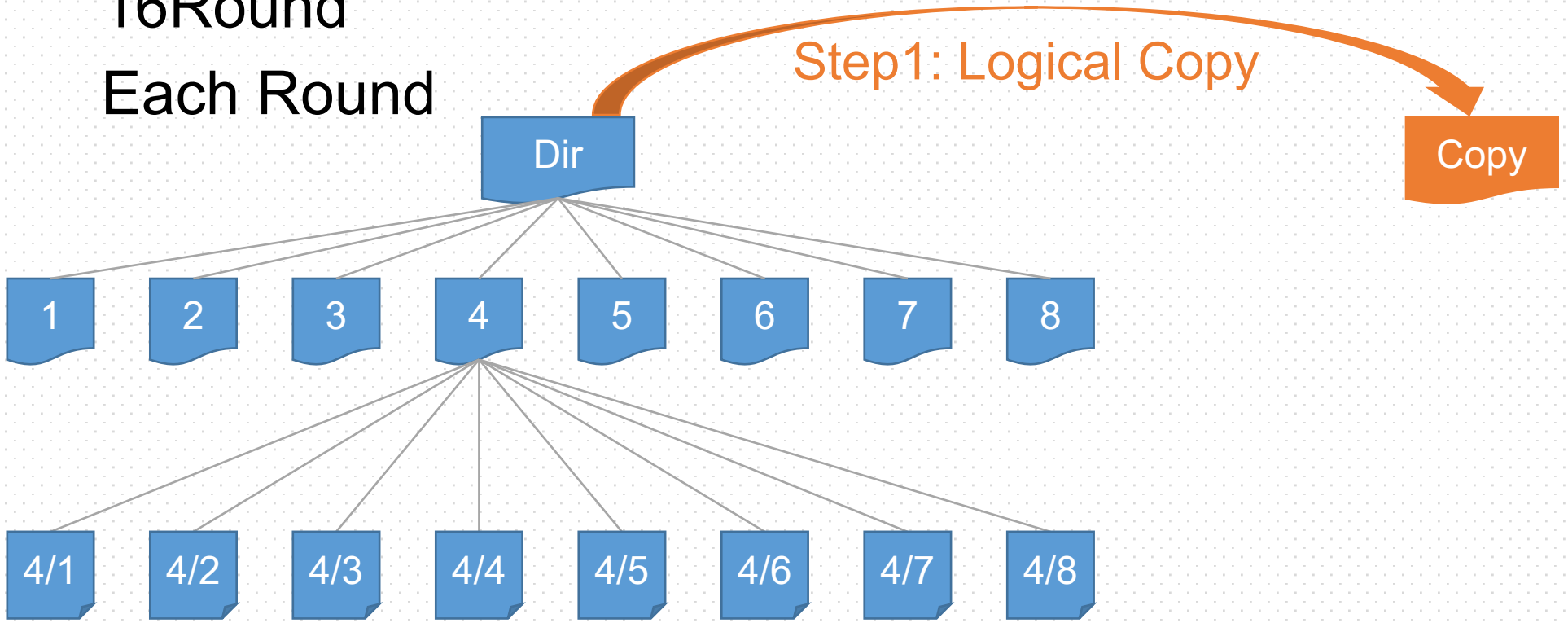
Workload



Let's test them!

16Round

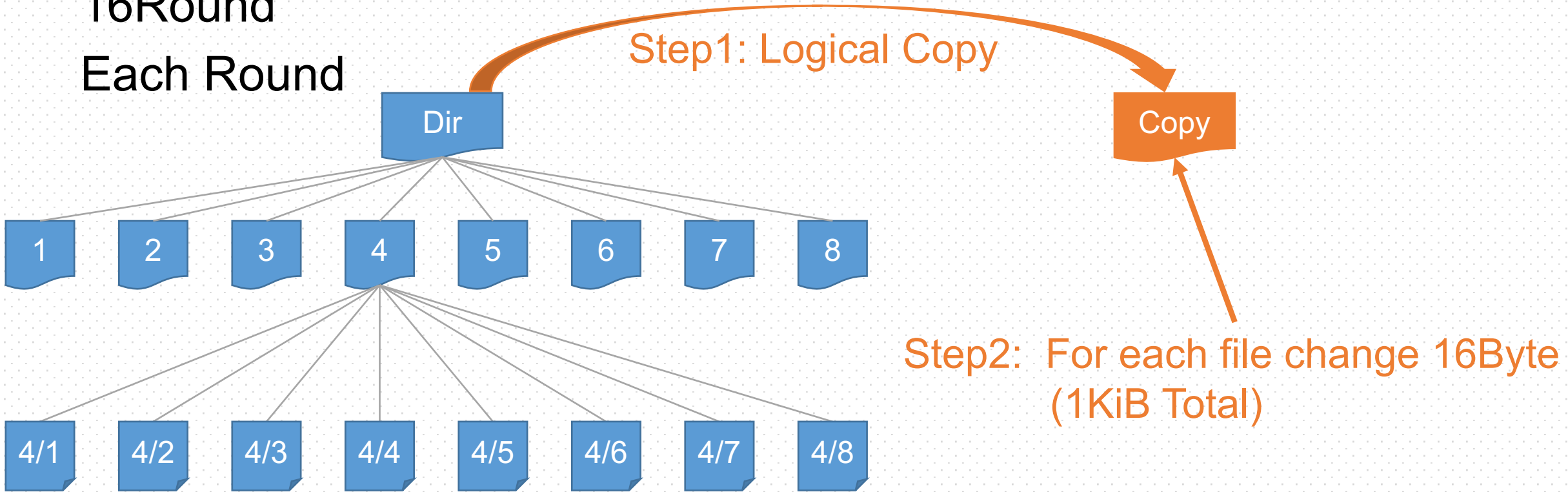
Each Round



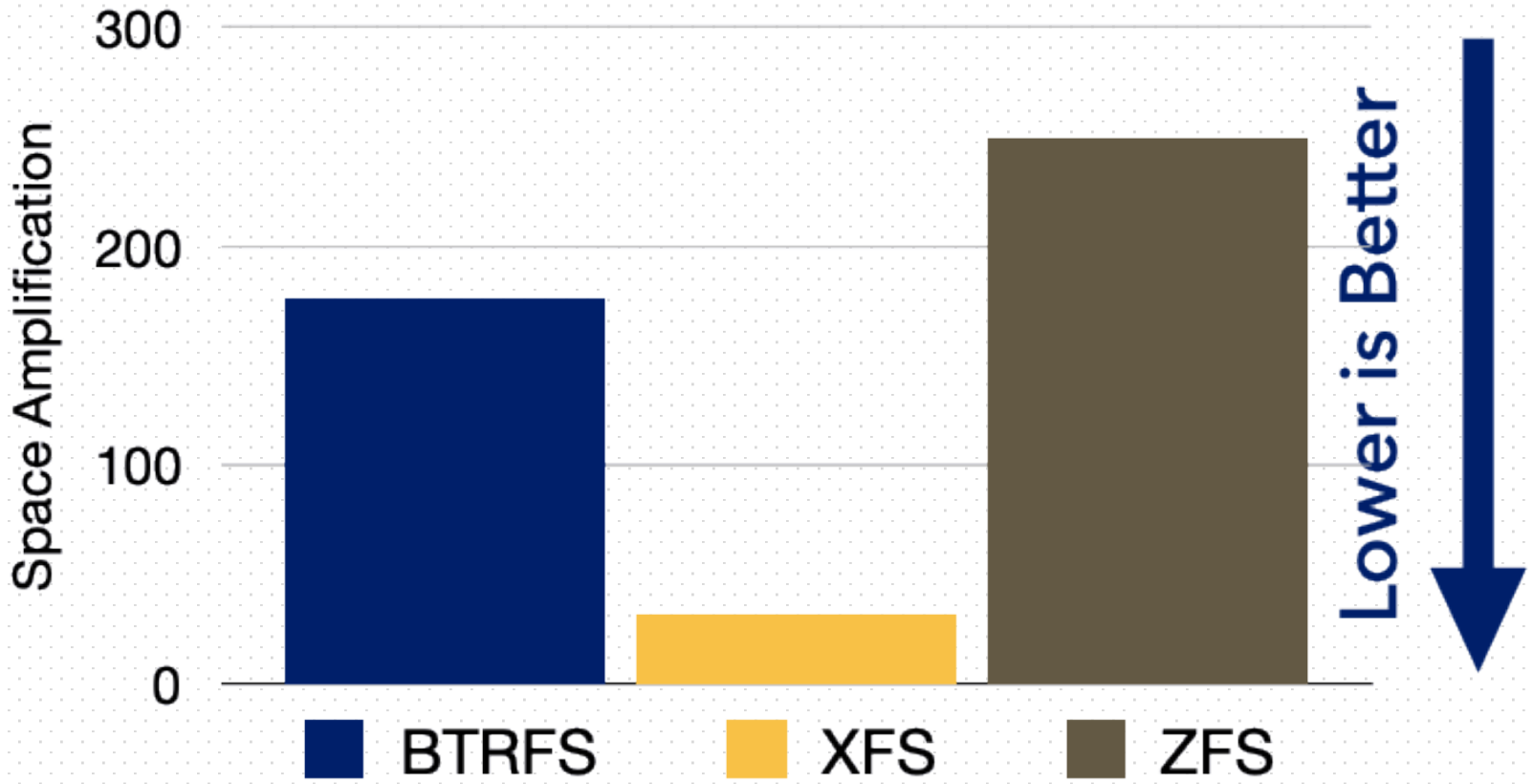
Let's test them!

16Round

Each Round

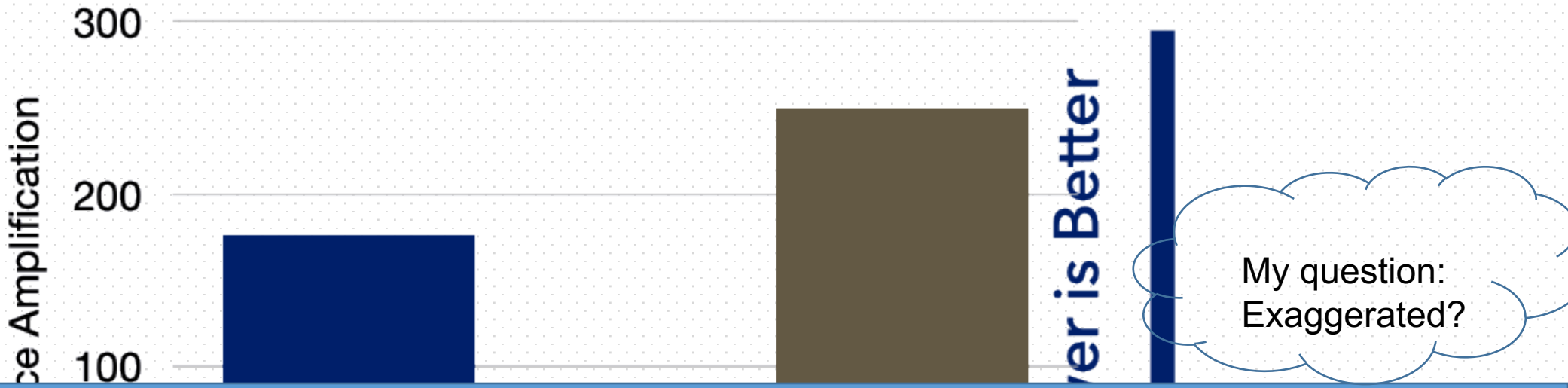


Write Amplification



File system space use / (16B*64)

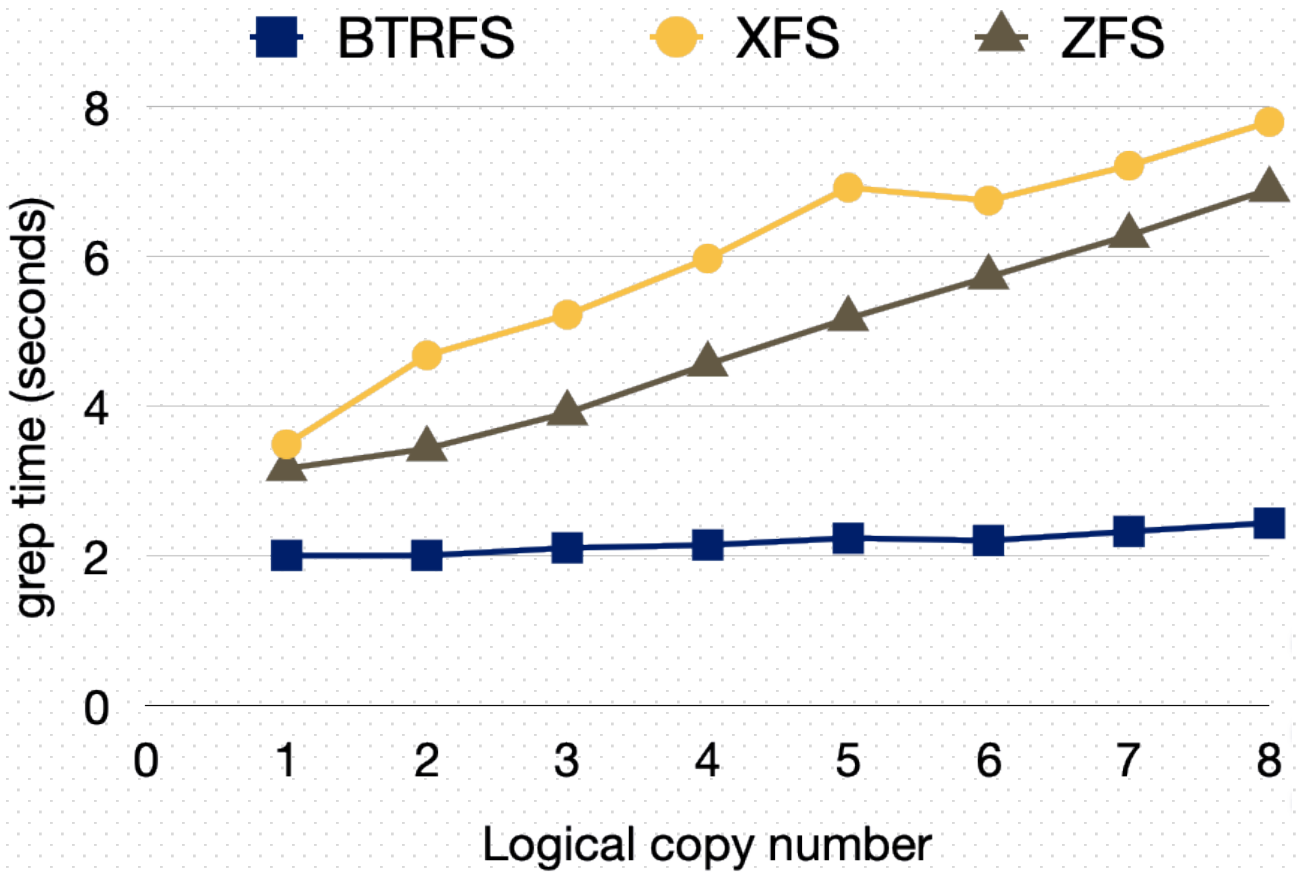
Write Amplification



Tens to hundreds of times Amplification
(XFS is best)

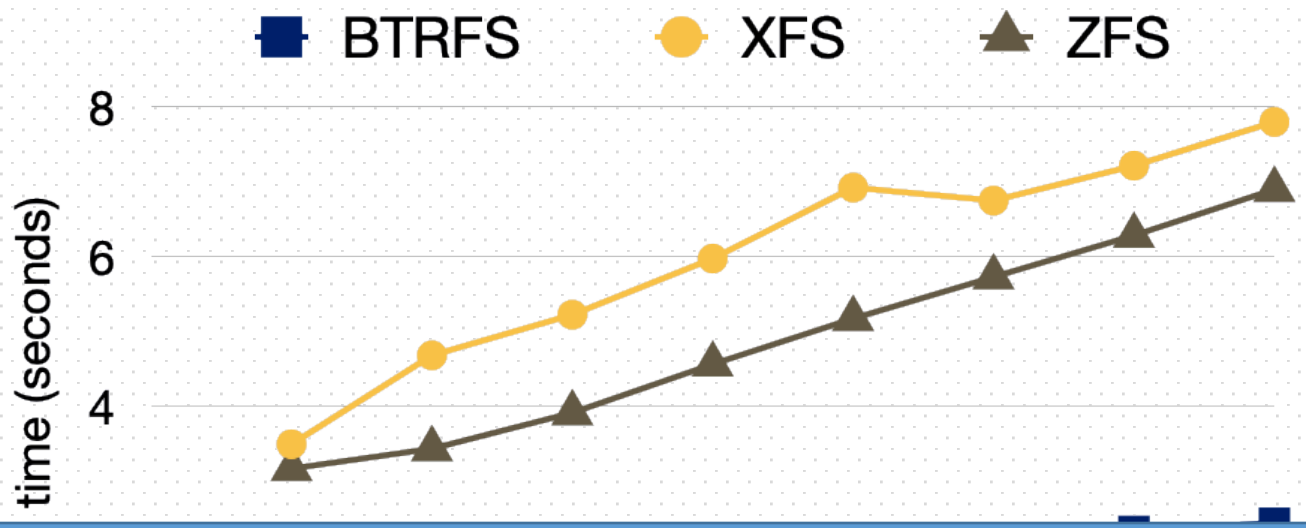
File system space use / (16B*64)

Fragmentation



Time cost: Grep over the lastest copy

Fragmentation



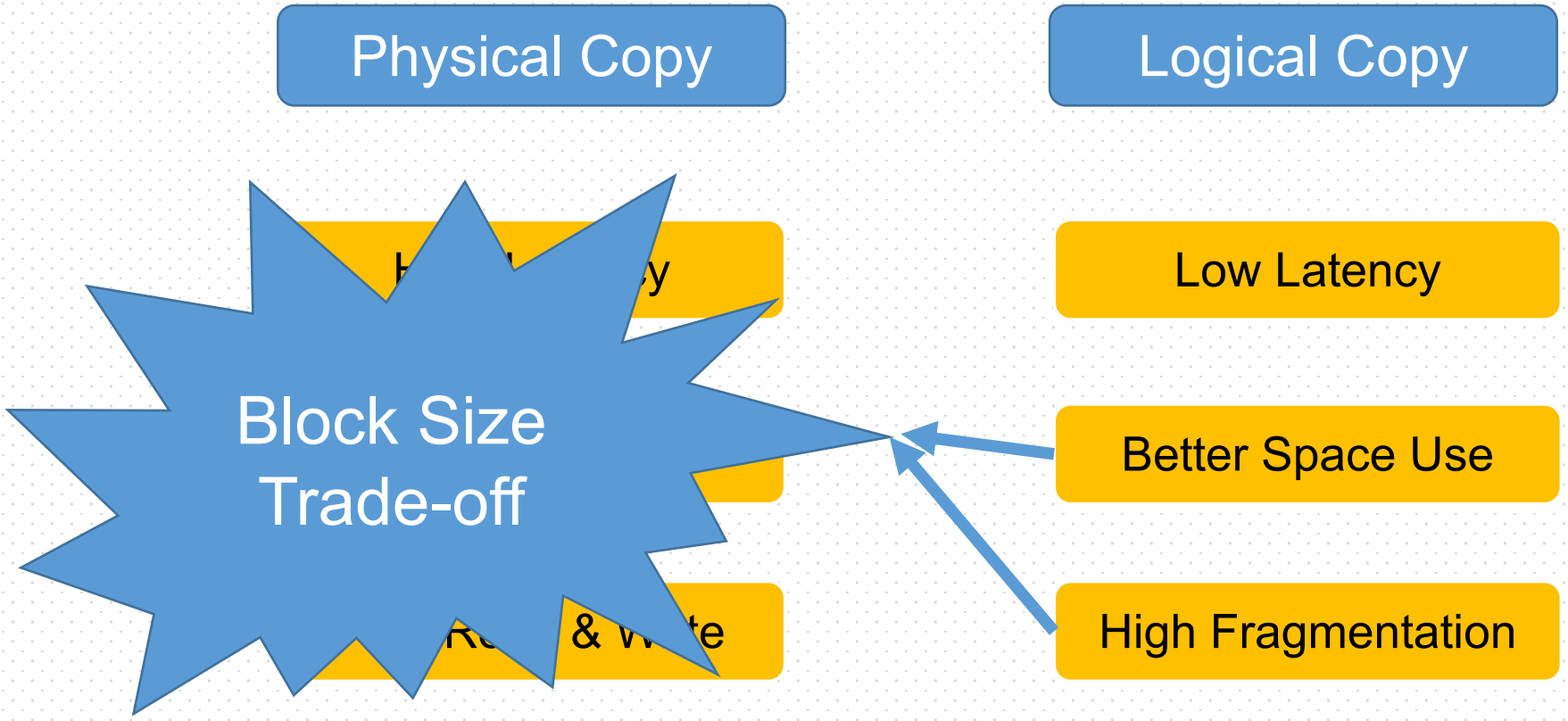
Different degrees of fragmentation
(XFS is Worst)

Time cost: Grep over the lastest copy

What's wrong with existing *Copy*?

Physical Copy	Logical Copy
High Latency	Low Latency
High Space Use	Better Space Use
Fast Read & Write	High Fragmentation

What's wrong with existing *Copy*?



What we want: Nimble Clone!

USTC, CHINA ADSLAB

1. Fast Creation

2. Excellent read locality
   (Even after modification)

3. Fast write (Original & Clone)

4. Conserve space

Contribution

A logical copy specification: ***Clone***

A set of performance criteria that a ***nimble*** clone must satisfy

The ***design*** for a file system and nimble clone implementation that meets all of these criteria.

Implementation

- Extend BetrFS 0.4:
  - an open-source, full-path-indexed, write-optimized file system
- BetrFS 0.5 policy:
  - ***Copy-on-Abundant-Write***
- Transforming **Bε-tree** to **Bε-DAG**

Implementation

- Extend **BetrFS** 0.4:
  - an open-source, full-path-indexed, write-optimized file system
- BetrFS 0.5 policy:
  - ***Copy-on- Abundant-Write***
- Transforming **Bε-tree** to **Bε-DAG**

Background of BetrFS and Bε-tree

What is BetrFS?

- BetrFS[1]:
  - an in-kernel, local file system
  - built on a KV-store substrate(Bε-tree)

- Two level KV-store in BetrFS:
  - Metadata to Full-path
  - Data(full path+block number) to 4KiB blocks

[1] BetrFS: A right- optimized write-optimized file system

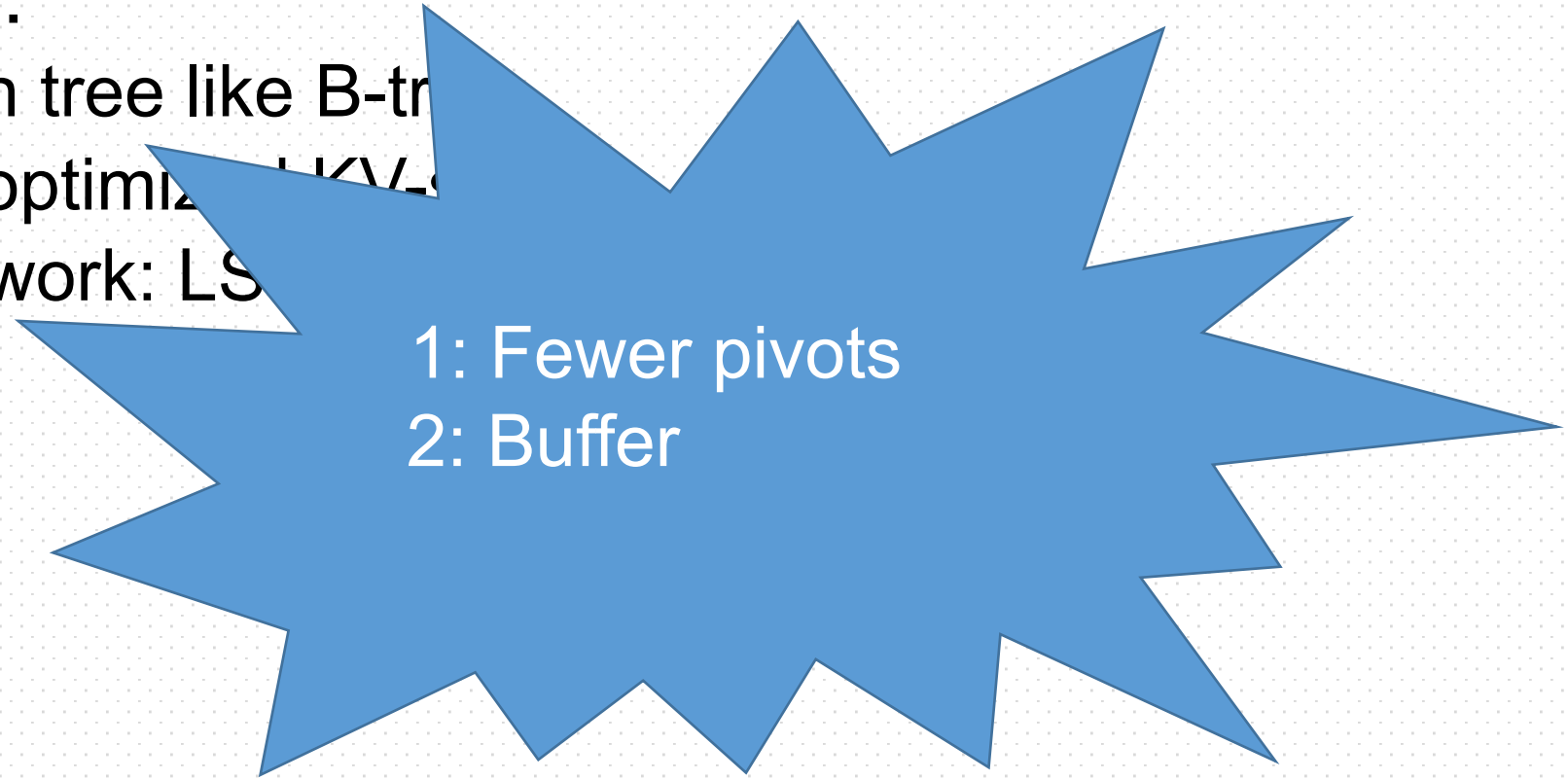
What is Bε-tree?

- Bε-tree[2]:
  - a search tree like B-tree
  - a write-optimized KV-store
  - related work: LSM-tree

[2] An Introduction to Bε-trees and Write-Optimization

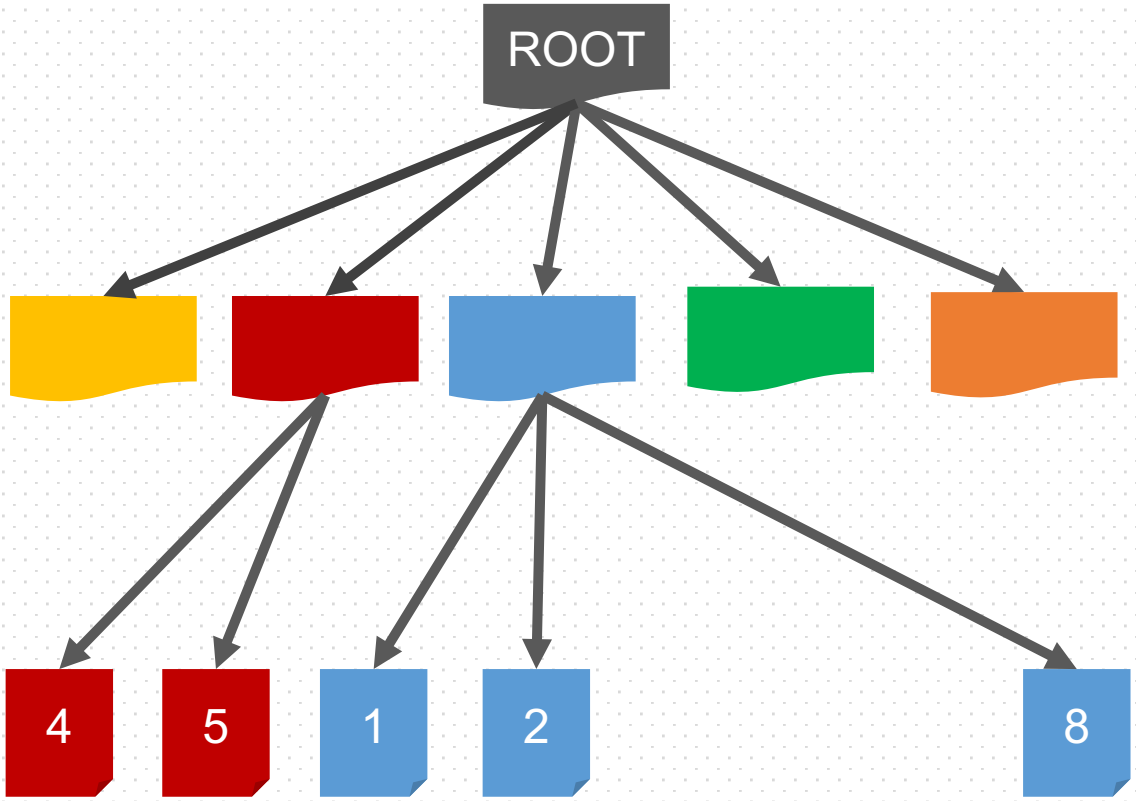
What is Bε-tree?

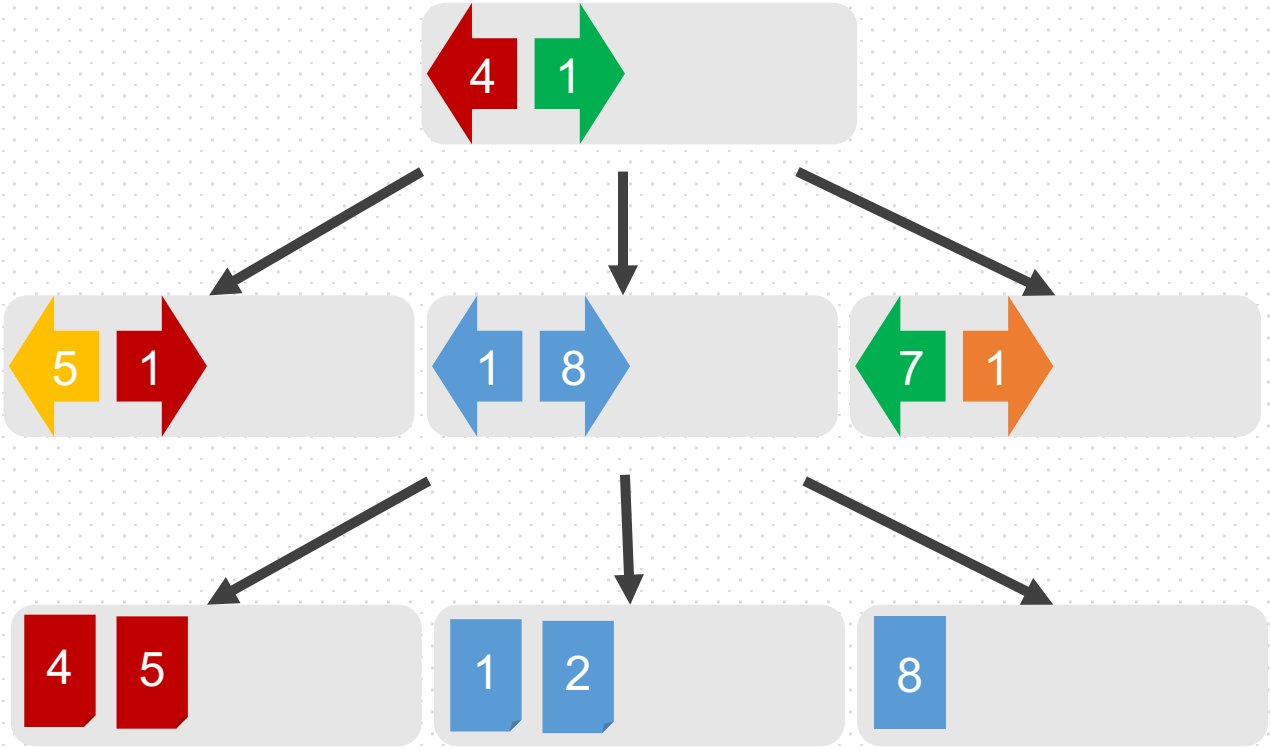
- Bε-tree[2]:
  - a search tree like B-tr
  - a write-optimi
  - related work: LS



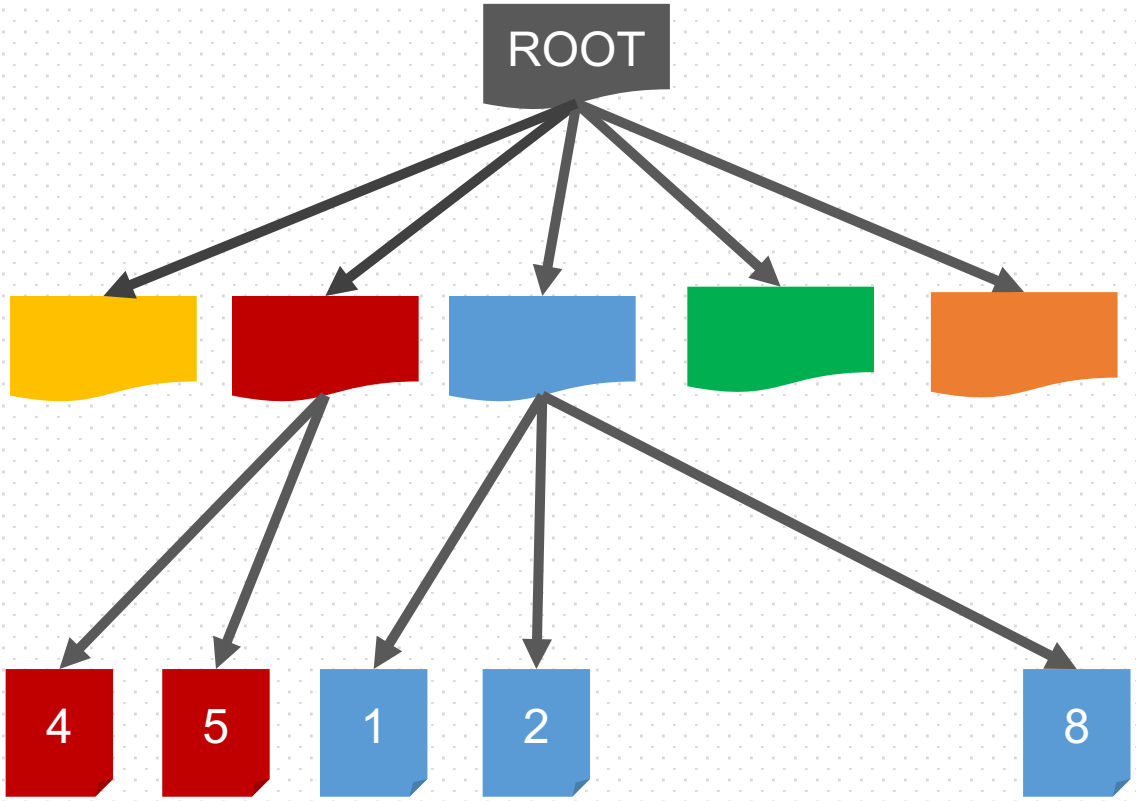
[2] An Introduction to Bε-trees and Write-Optimization

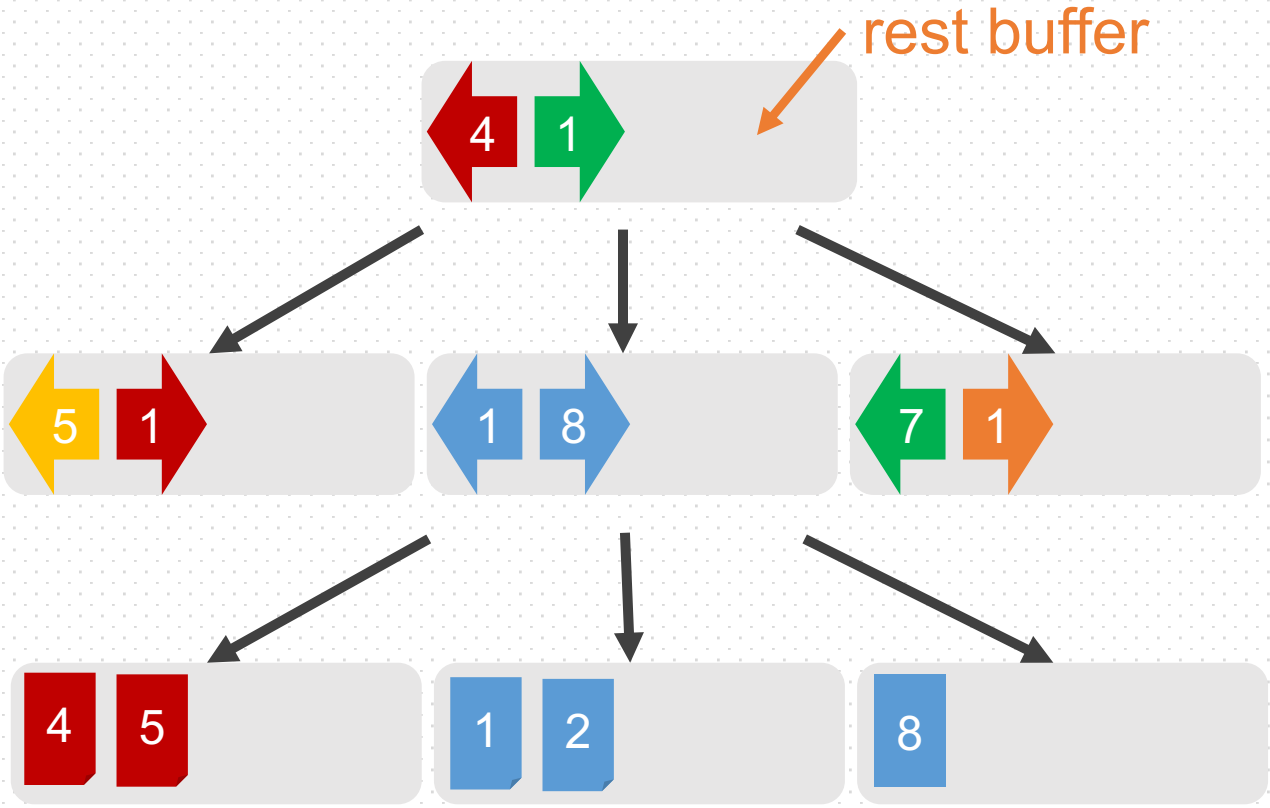
Bε-tree





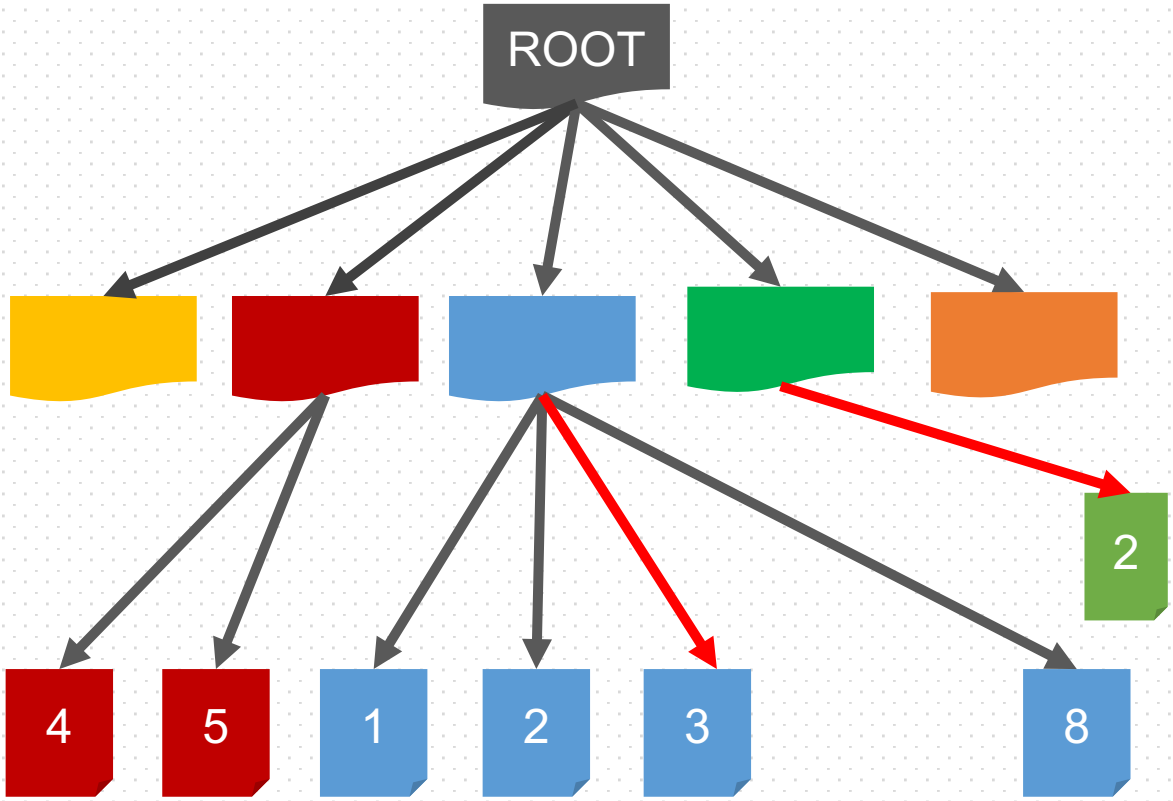
Bε-tree

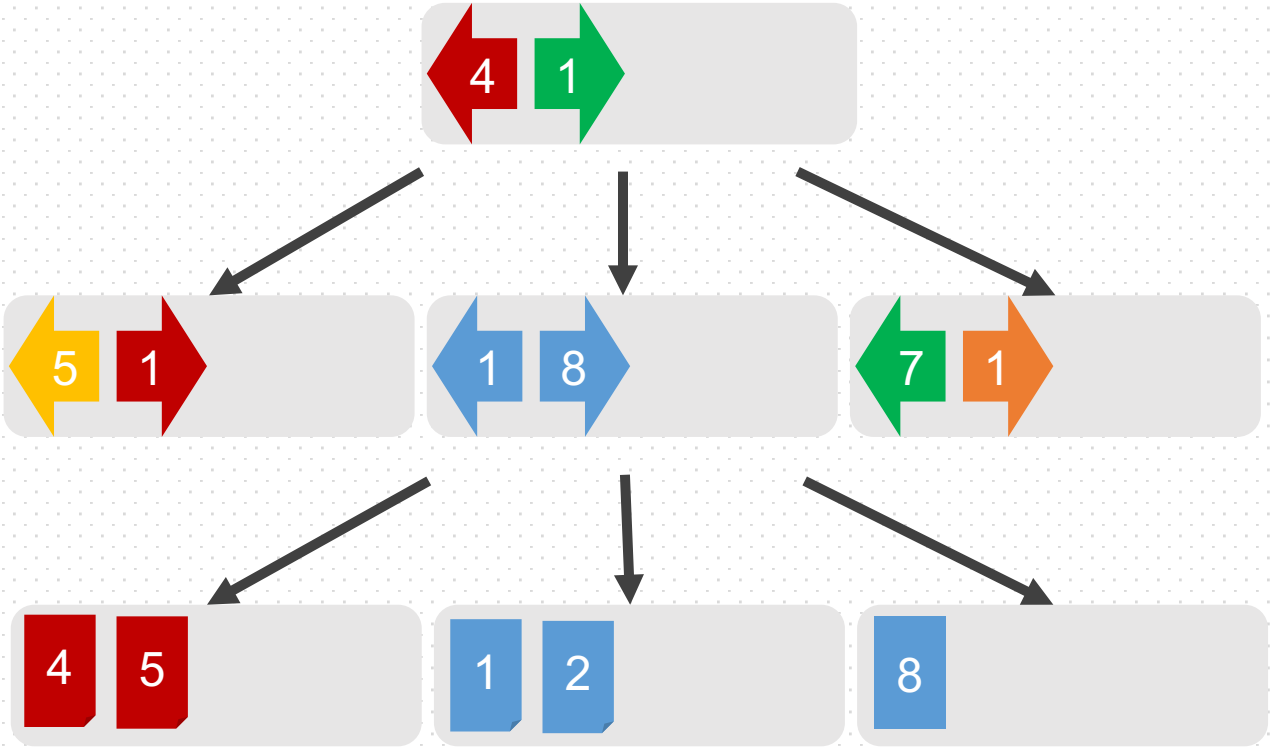




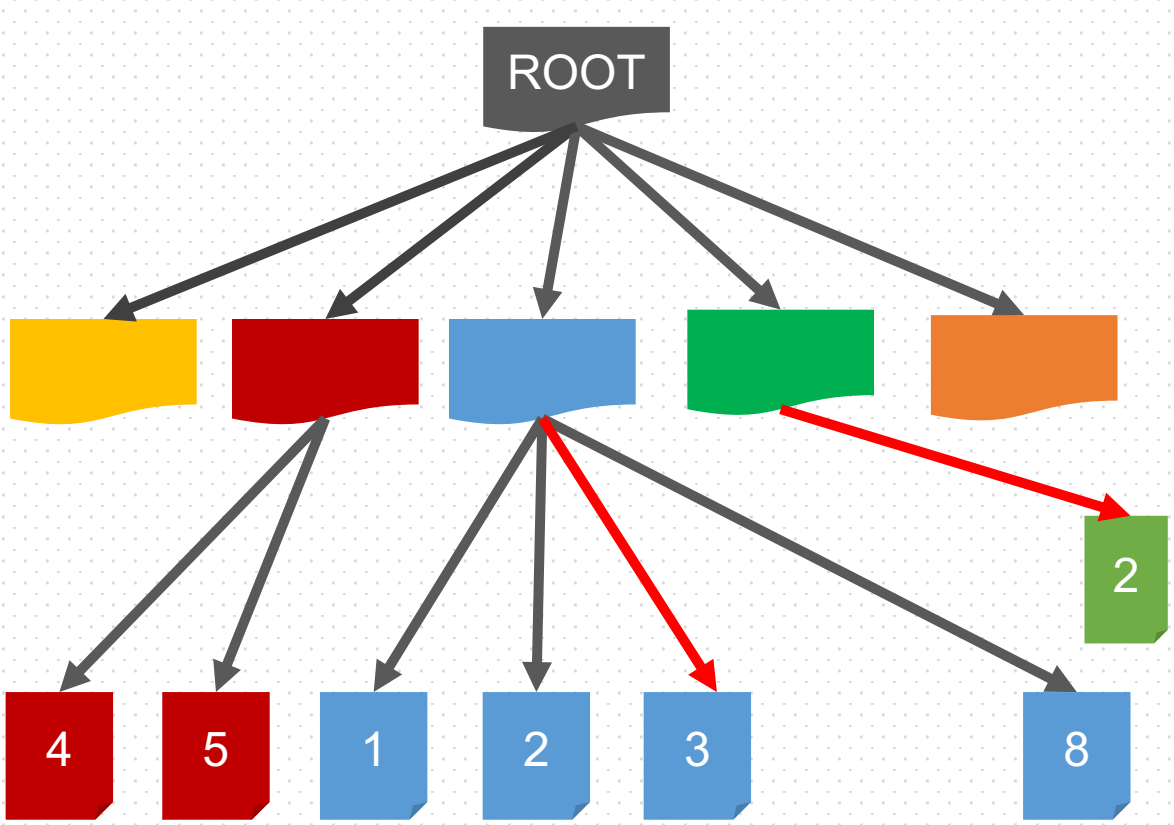
Pivot

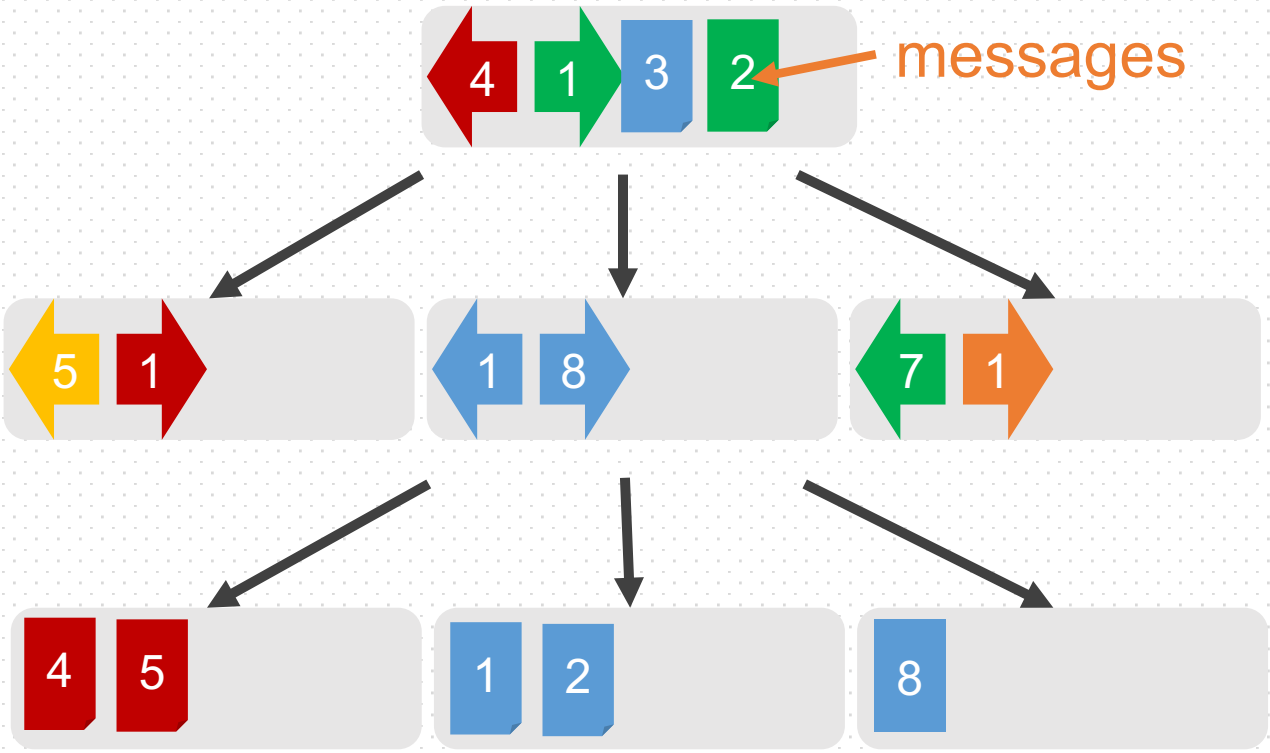
Bε-tree



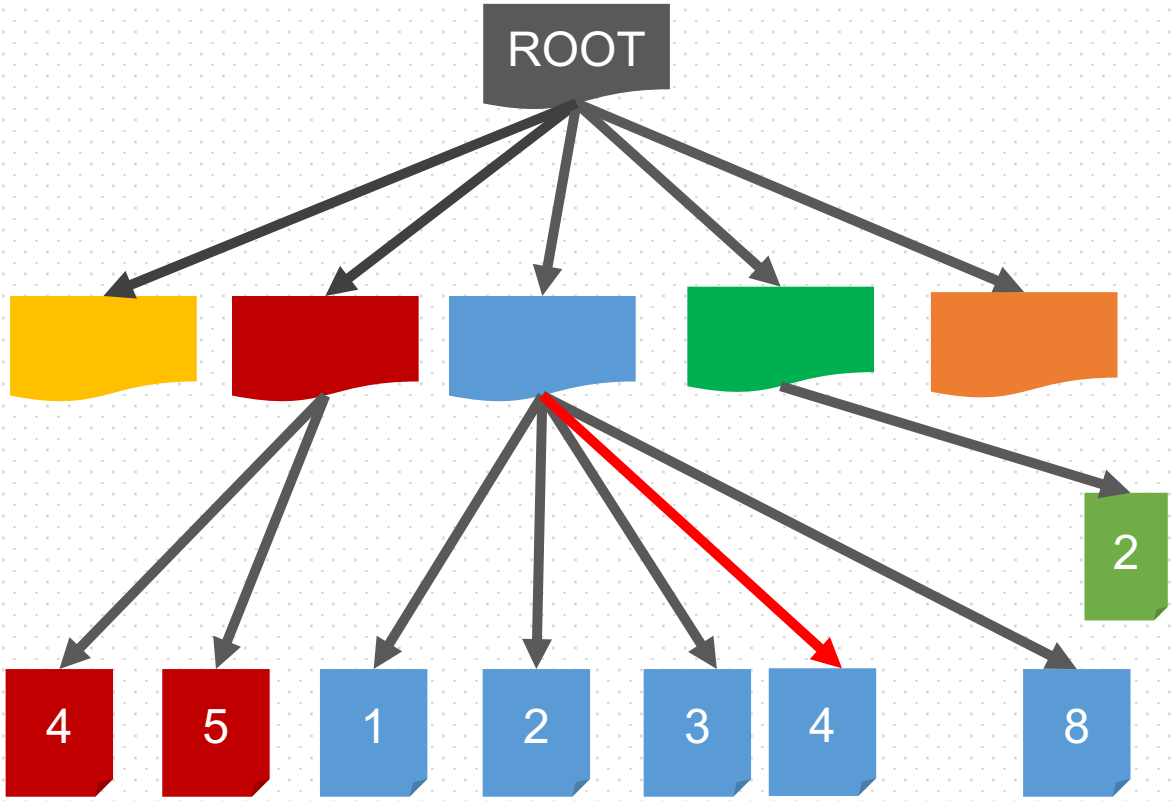


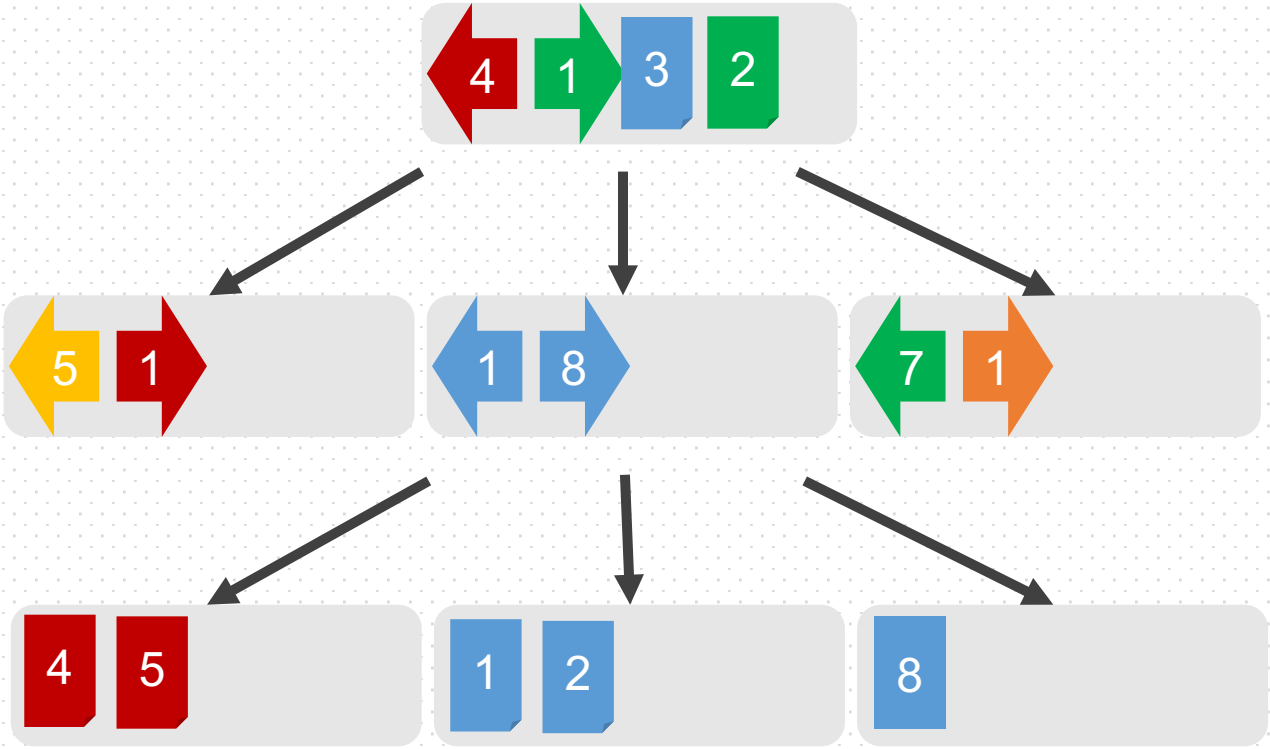
Bε-tree



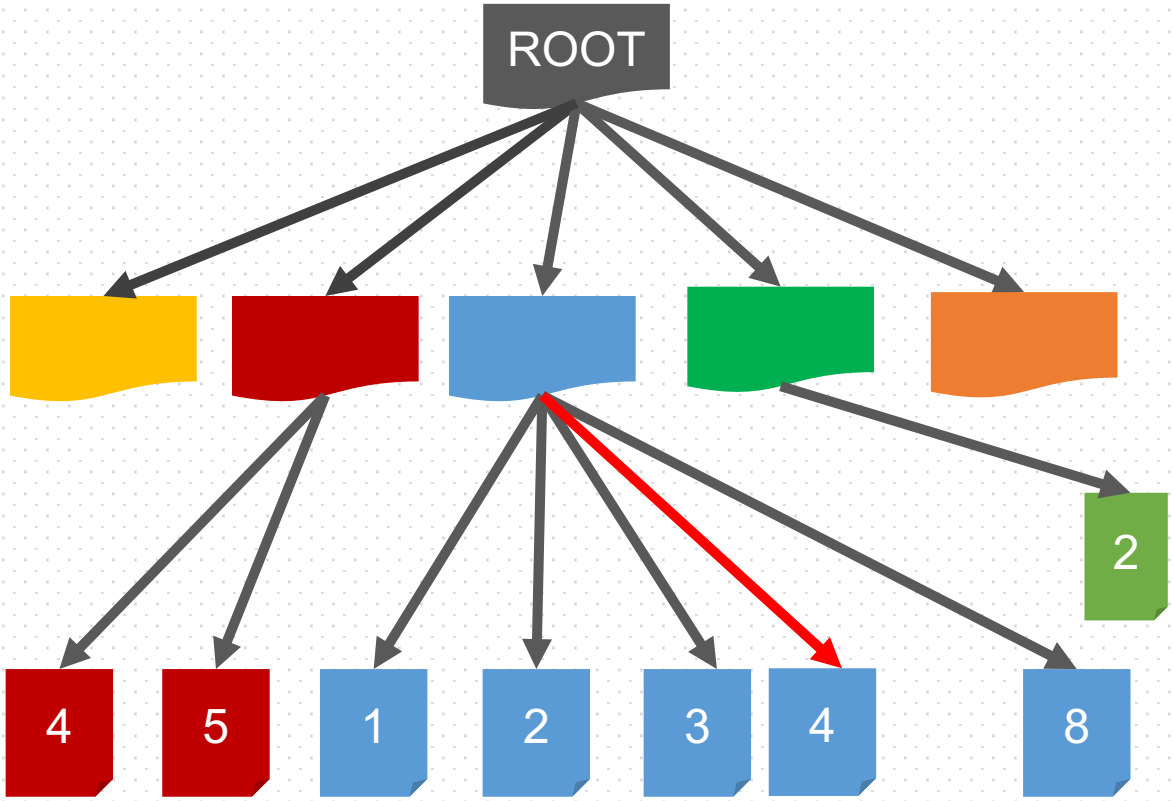


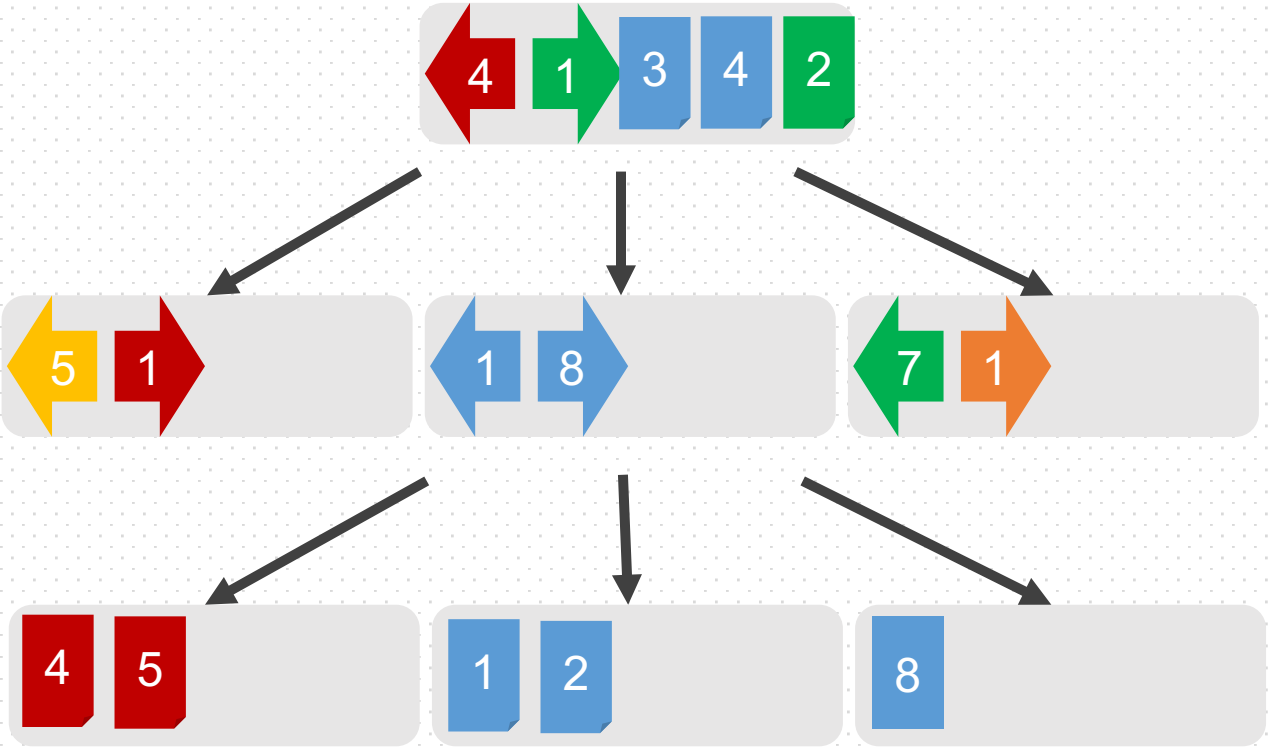
Bε-tree



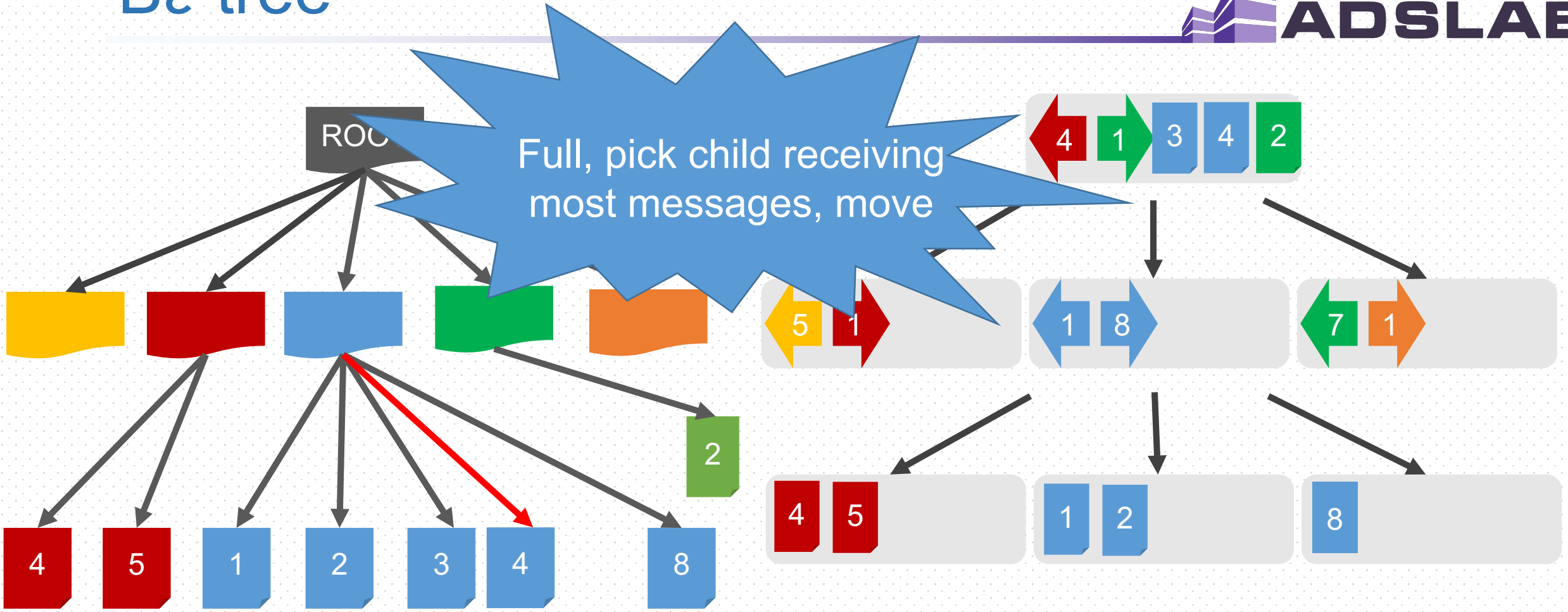


Bε-tree

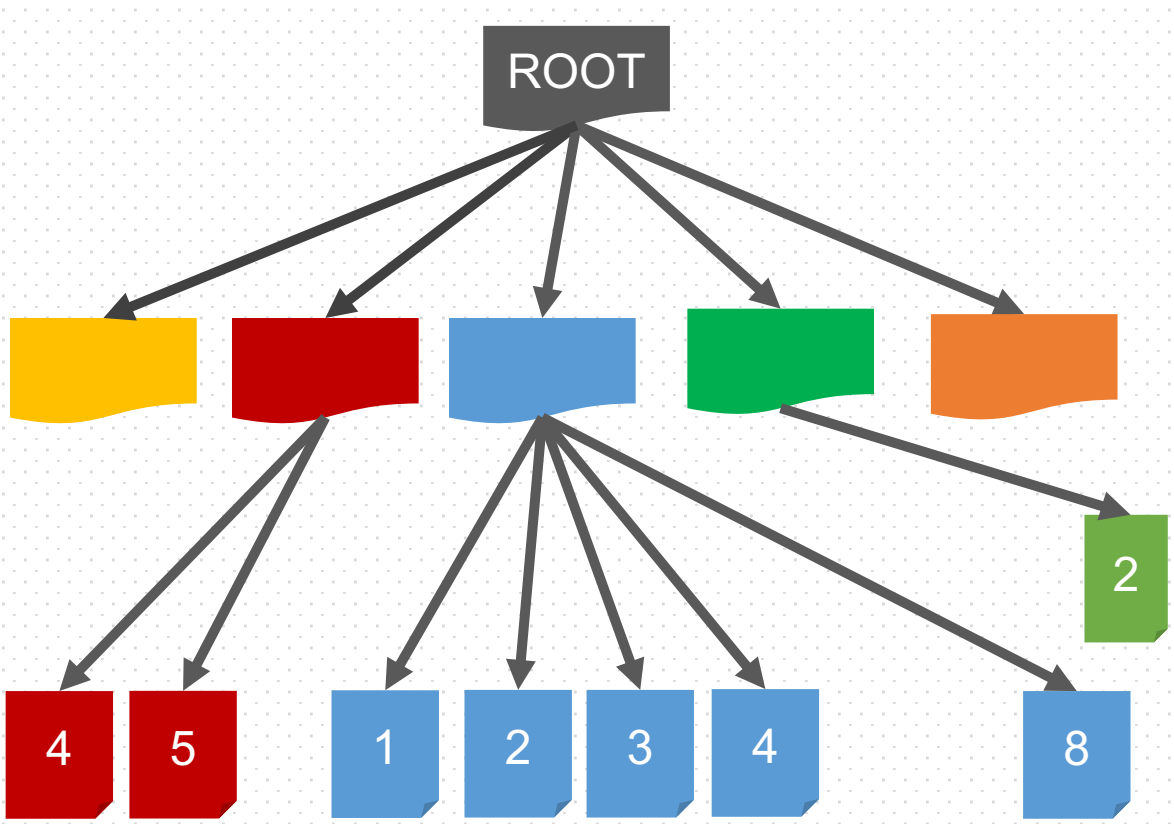


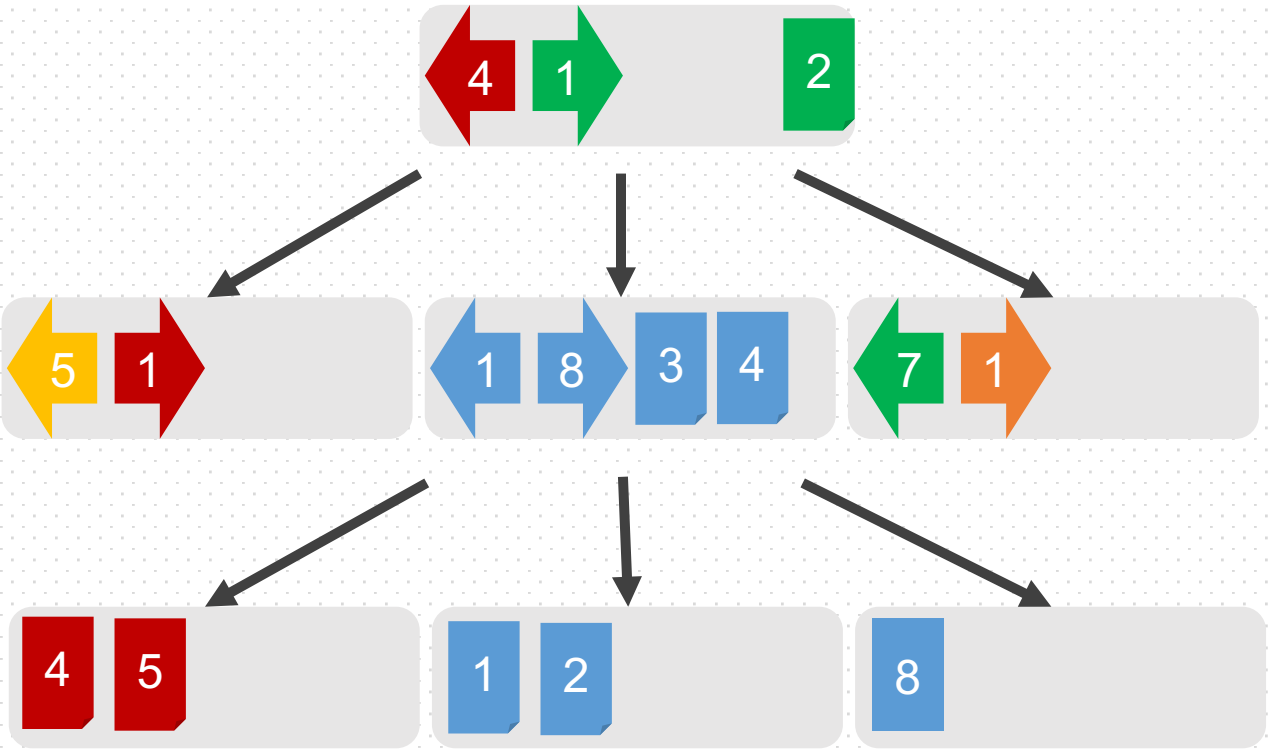


Bε-tree

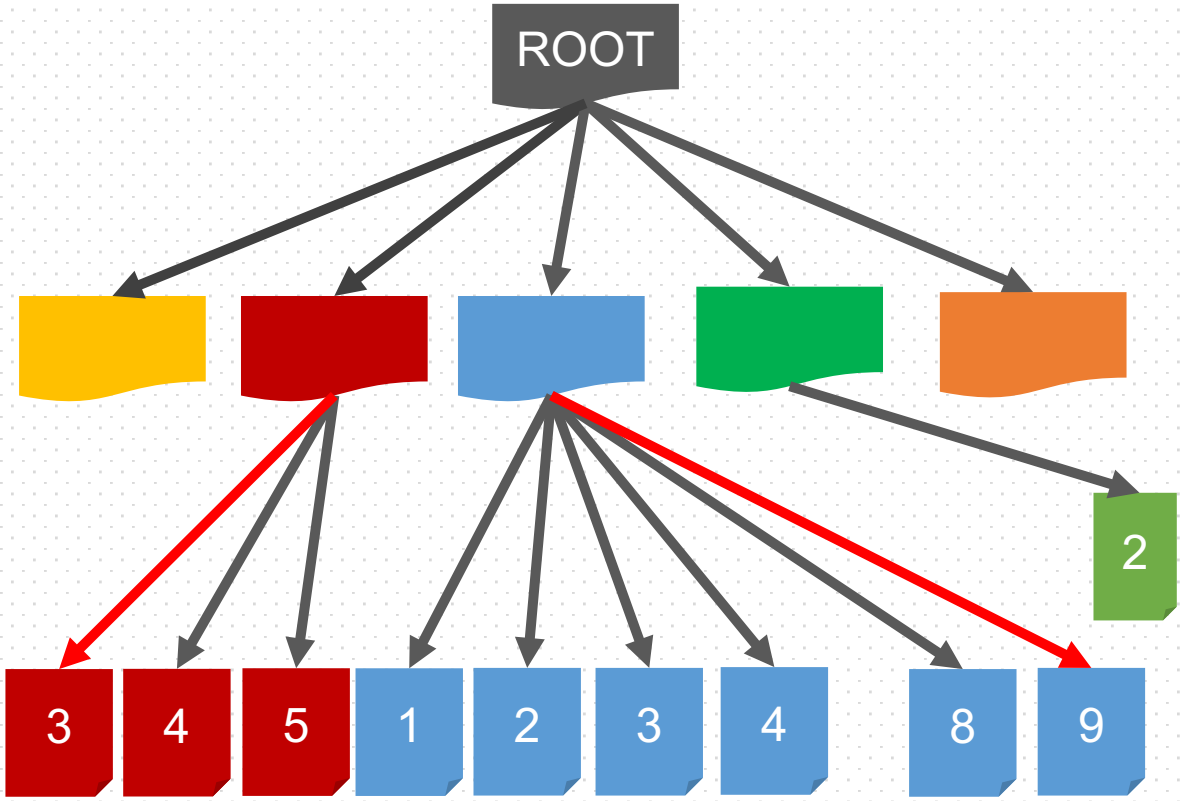

Full, pick child receiving most messages, move
4 1 3 4 2
5 1
1 8
7 1
2
4 5 1 2 3 4 8
4 5
1 2
8

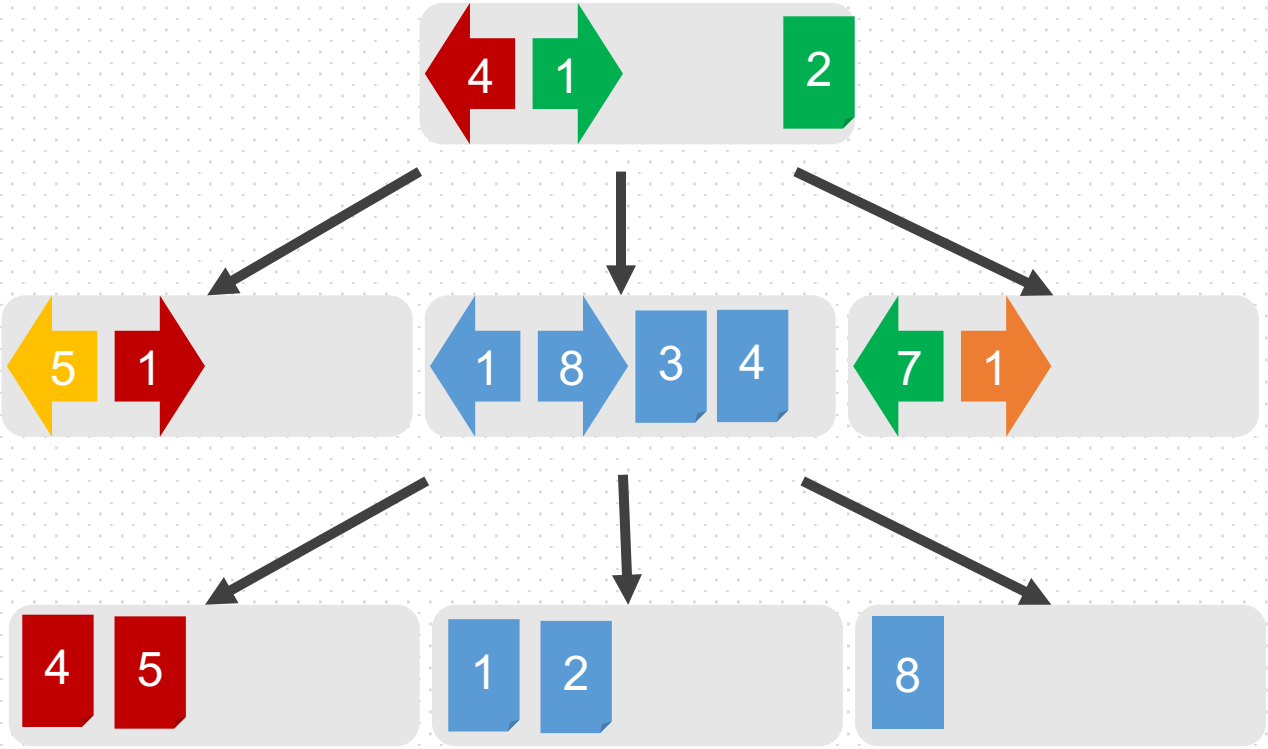
Bε-tree



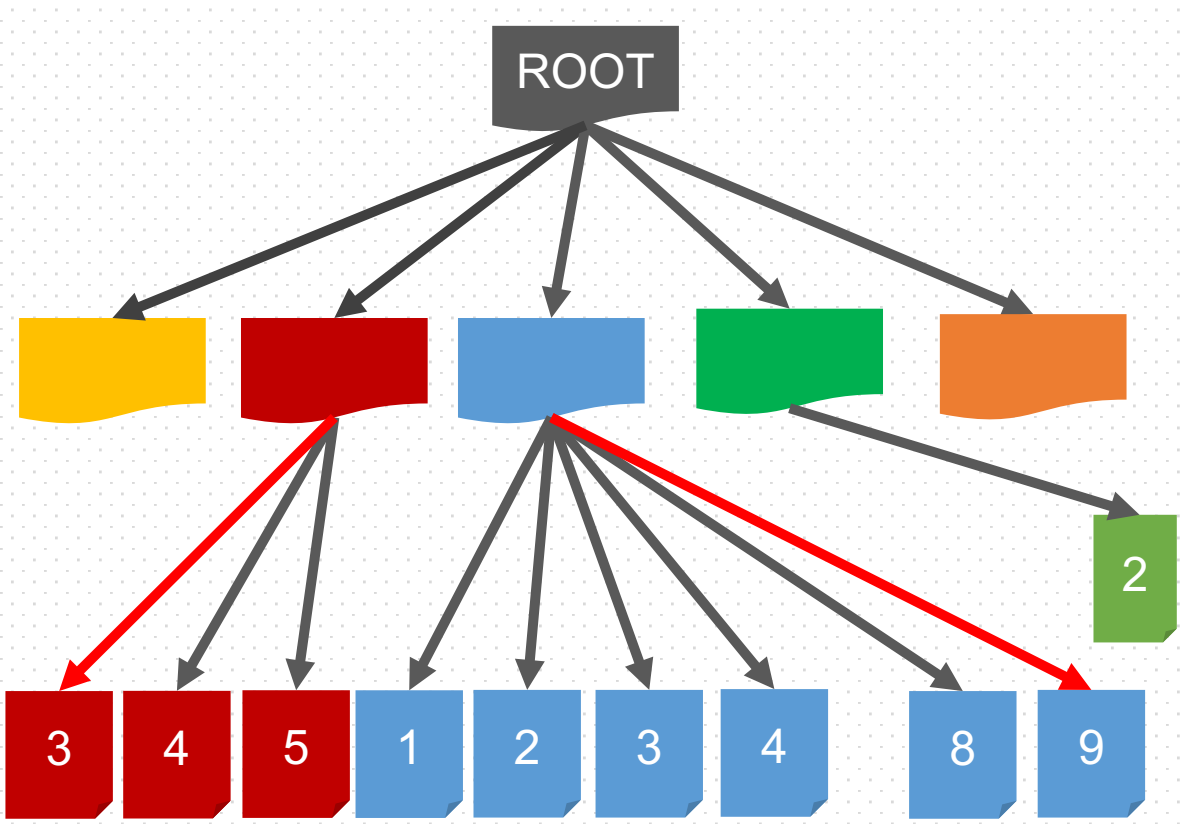


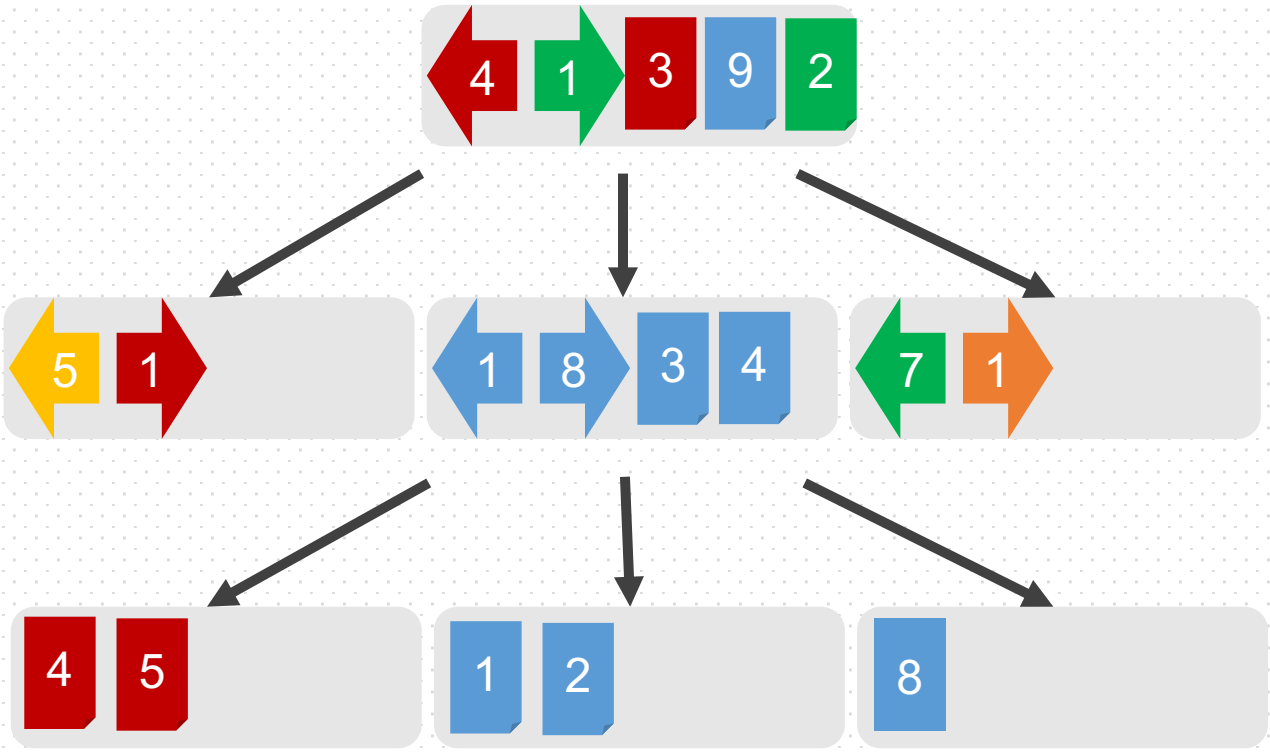
Bε-tree



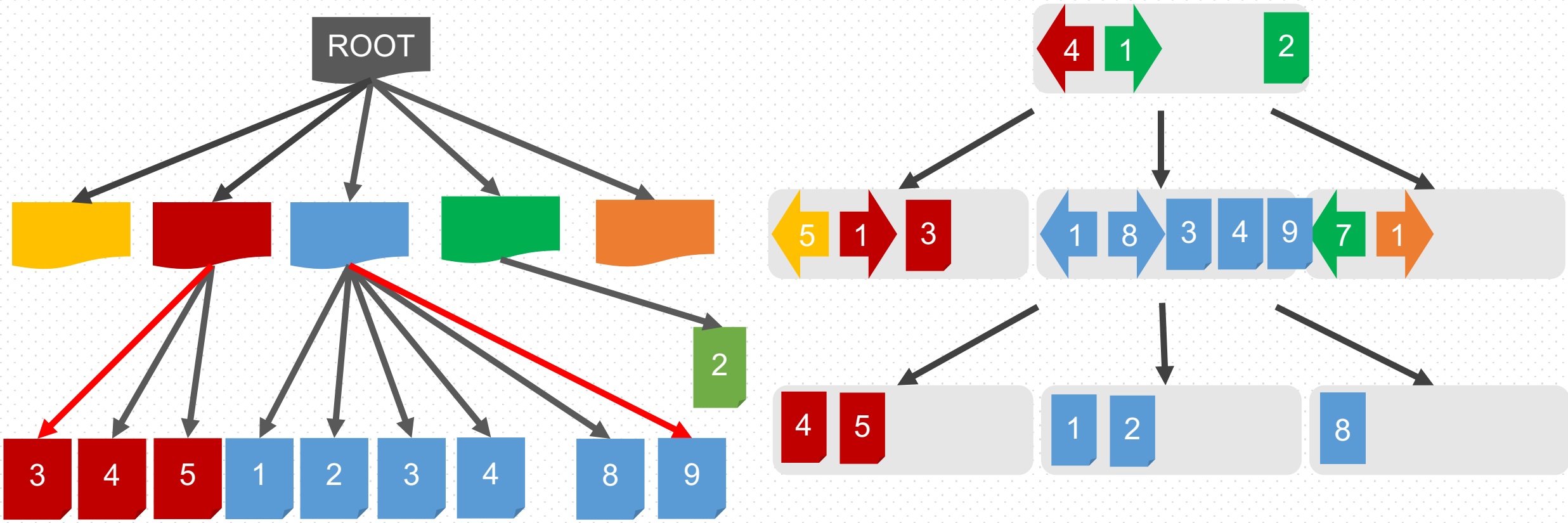


Bε-tree

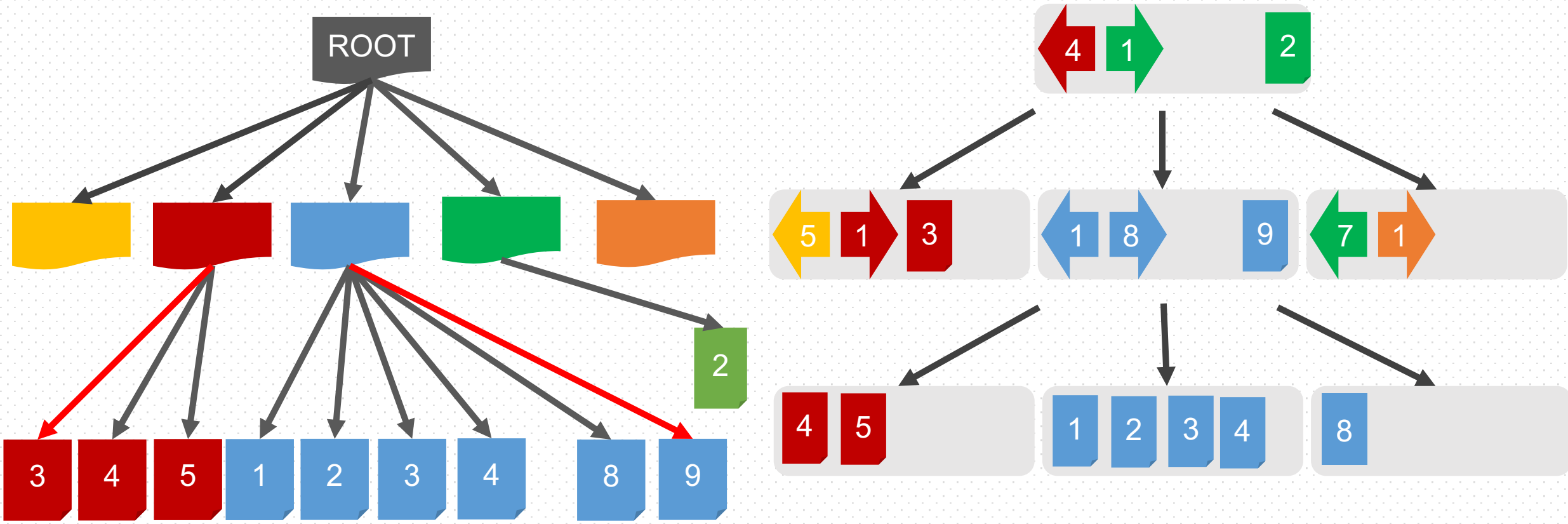




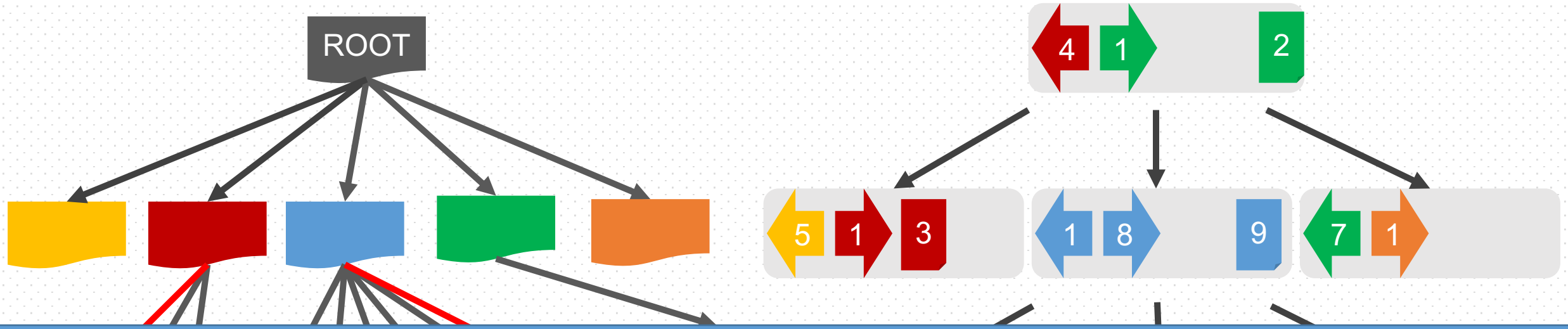
Bε-tree



Bε-tree



Bε-tree



Each flush applies manys small changes

write-optimized message-batching model

Cloning in BetrFS 0.5

Cloning Operation Semantics

- When we say clone, we mean copying files or directories.
- However, files and directories are mapped into keys based on their full paths.

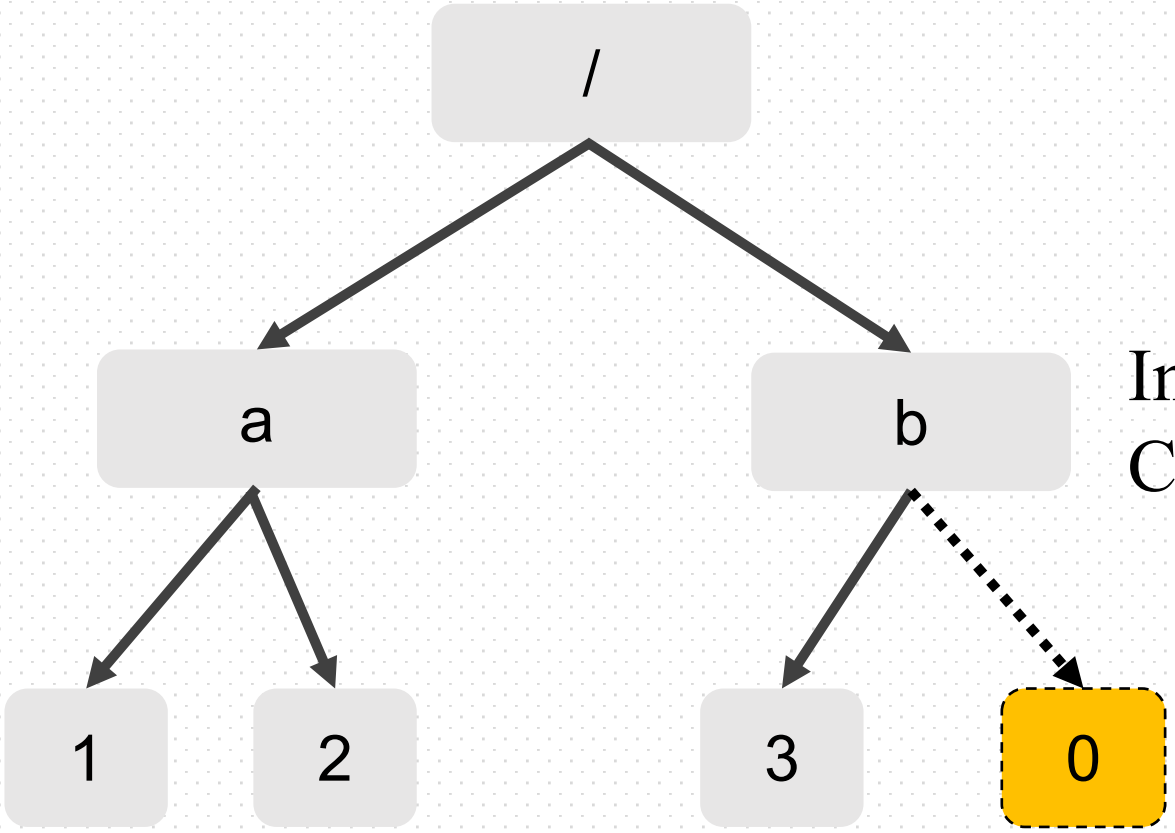
- So, should we define CLONE in which abstraction level?

Cloning Operation Semantics

- CLONE operation semantics:
- A CLONE operation takes as input two paths:
  - a source path—either a file or directory tree root
  - a destination path.

- Why does it work for both file system level and KV-store level?

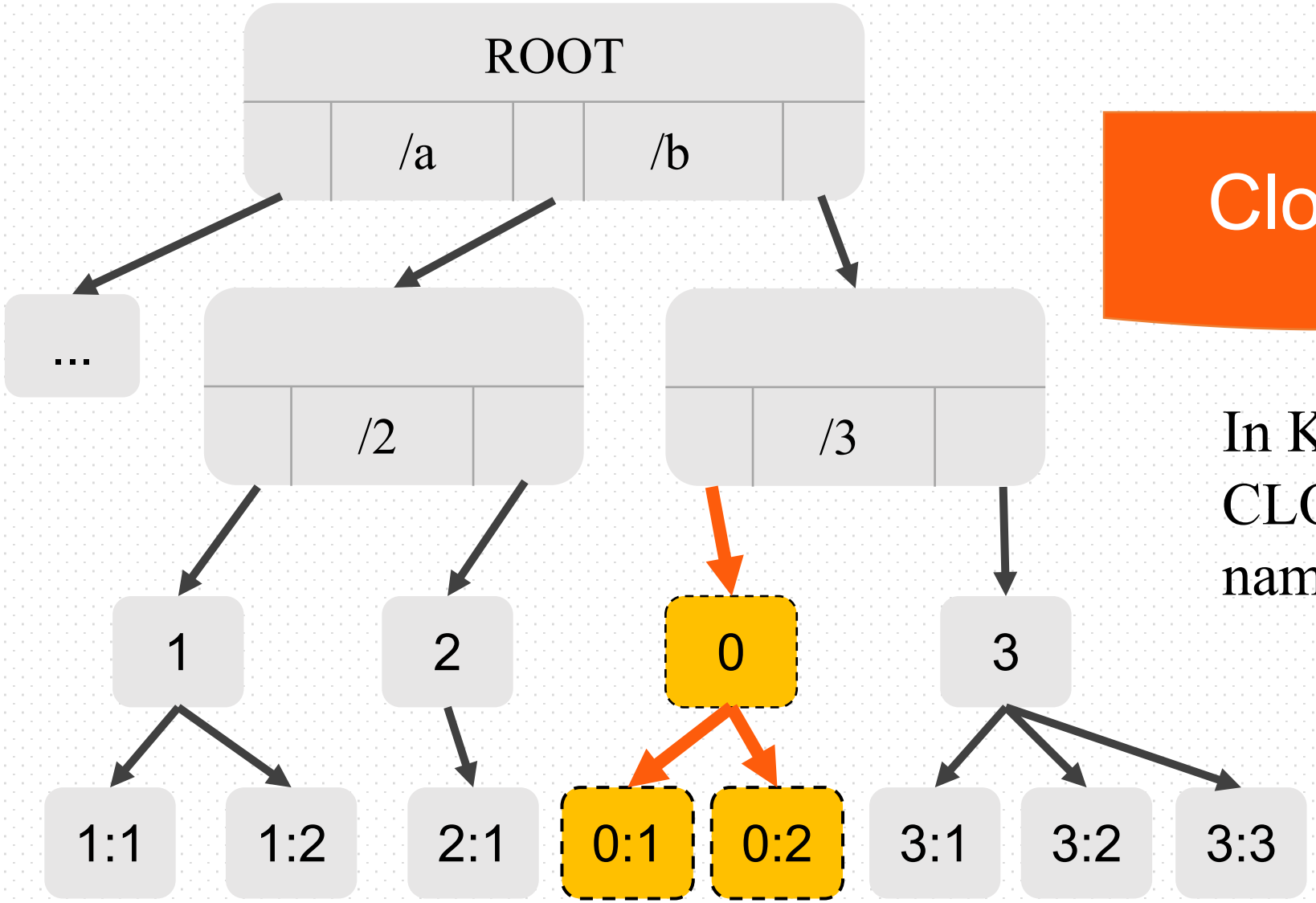
Cloning Operation Semantics



Clone: /a/1 to /b/0

In file system directory hierarchy:
CLONE means copying files (logically)

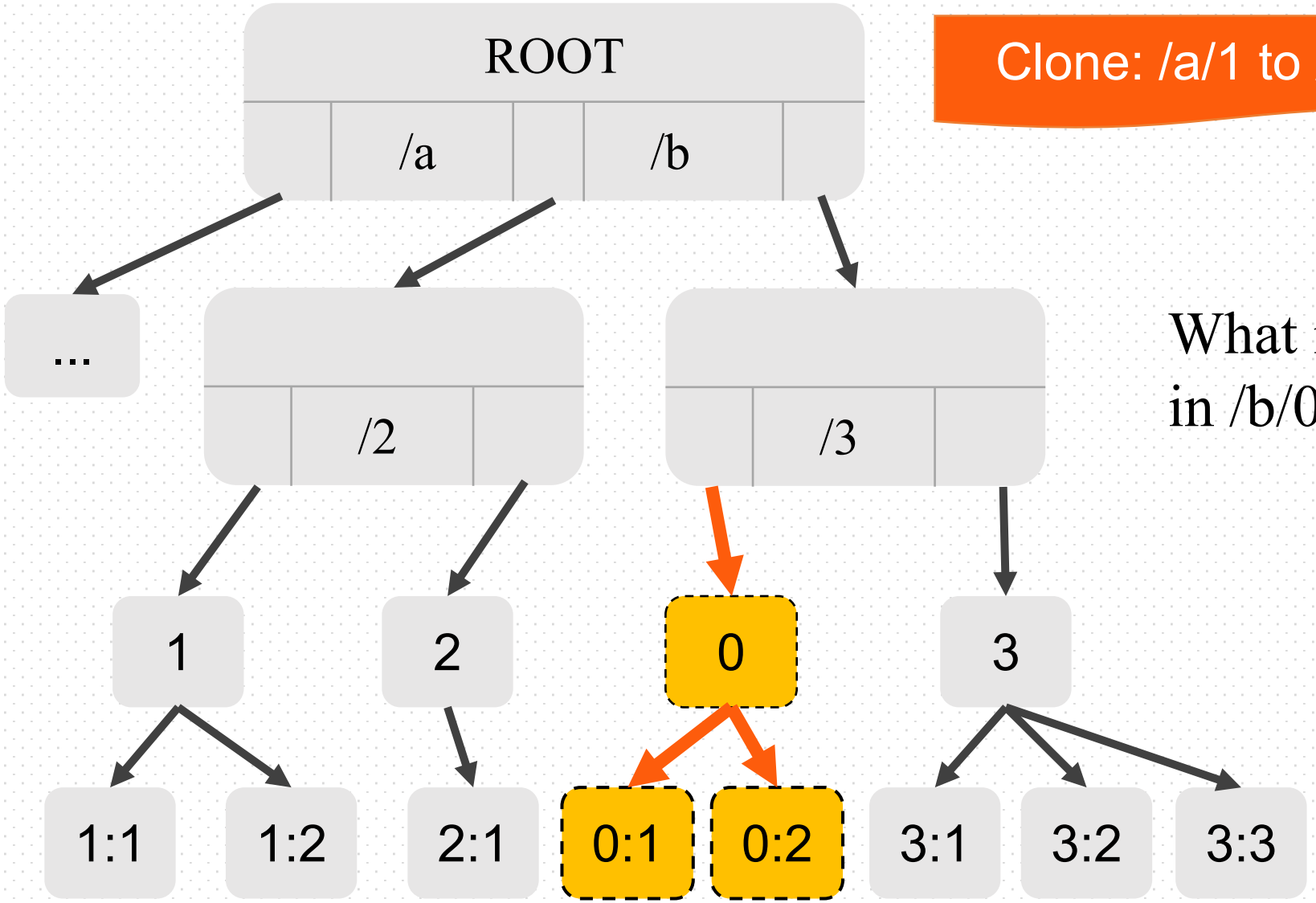
Cloning Operation Semantics



Clone: /a/1 to /b/0

In KV-store keyspace:
CLONE means copying a namespace

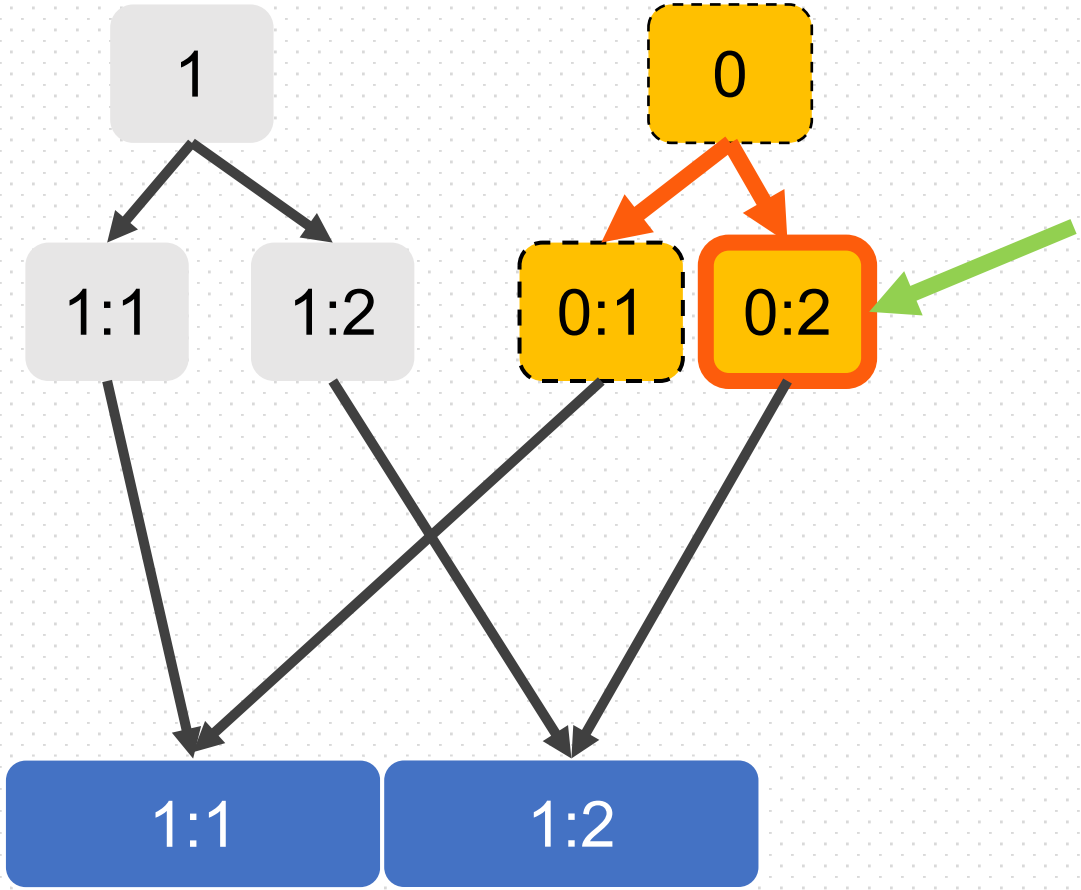
Problem: write amplification



Clone: /a/1 to /b/0

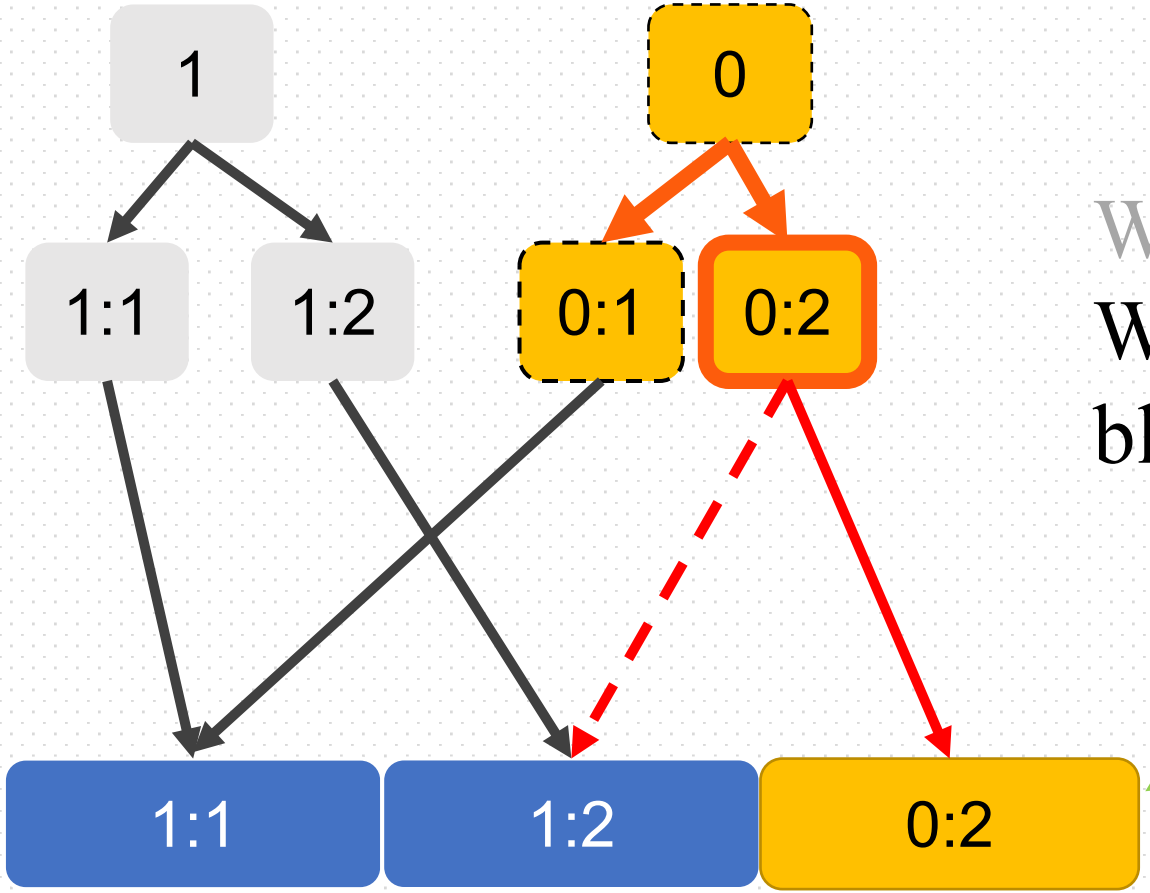
What if I modify the last byte in /b/0?

Problem: write amplification



What if we modify this?

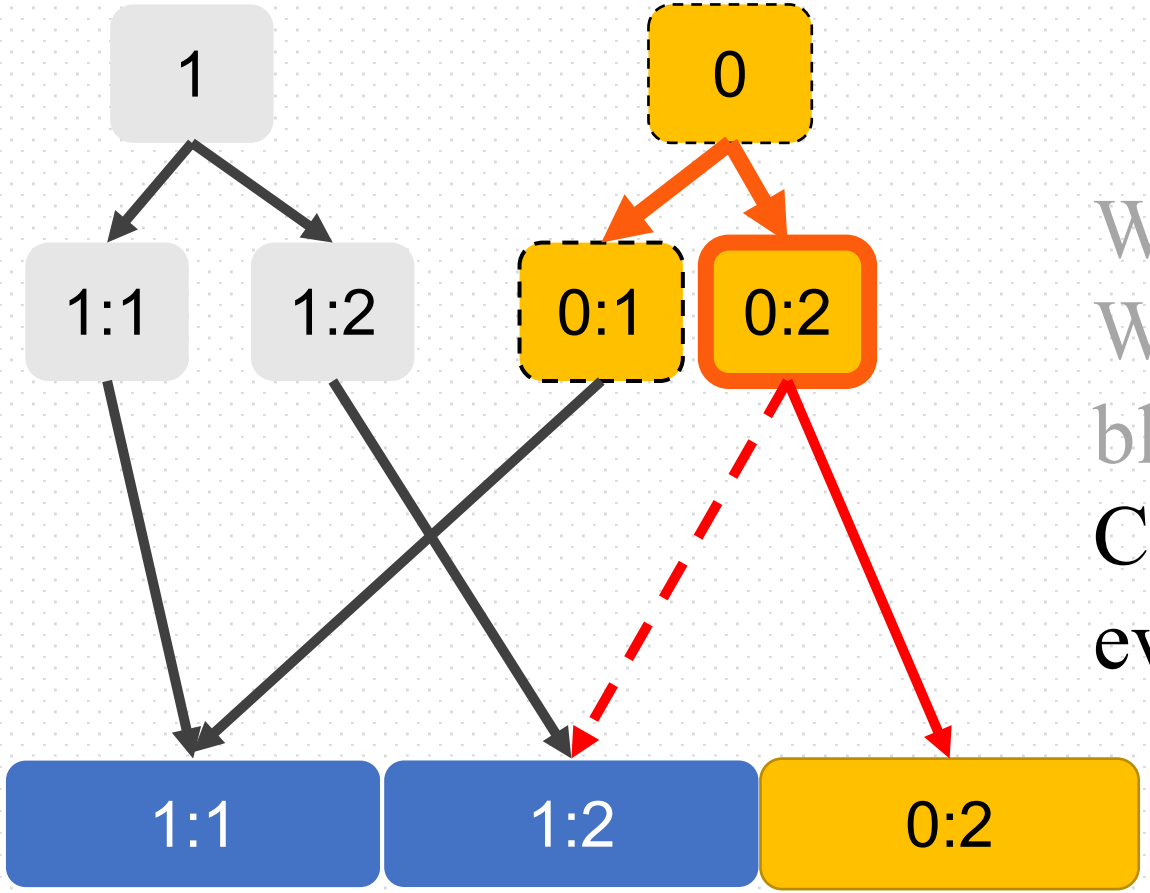
Problem: write amplification



What if we modify this?

We still need to create a data block immediately.

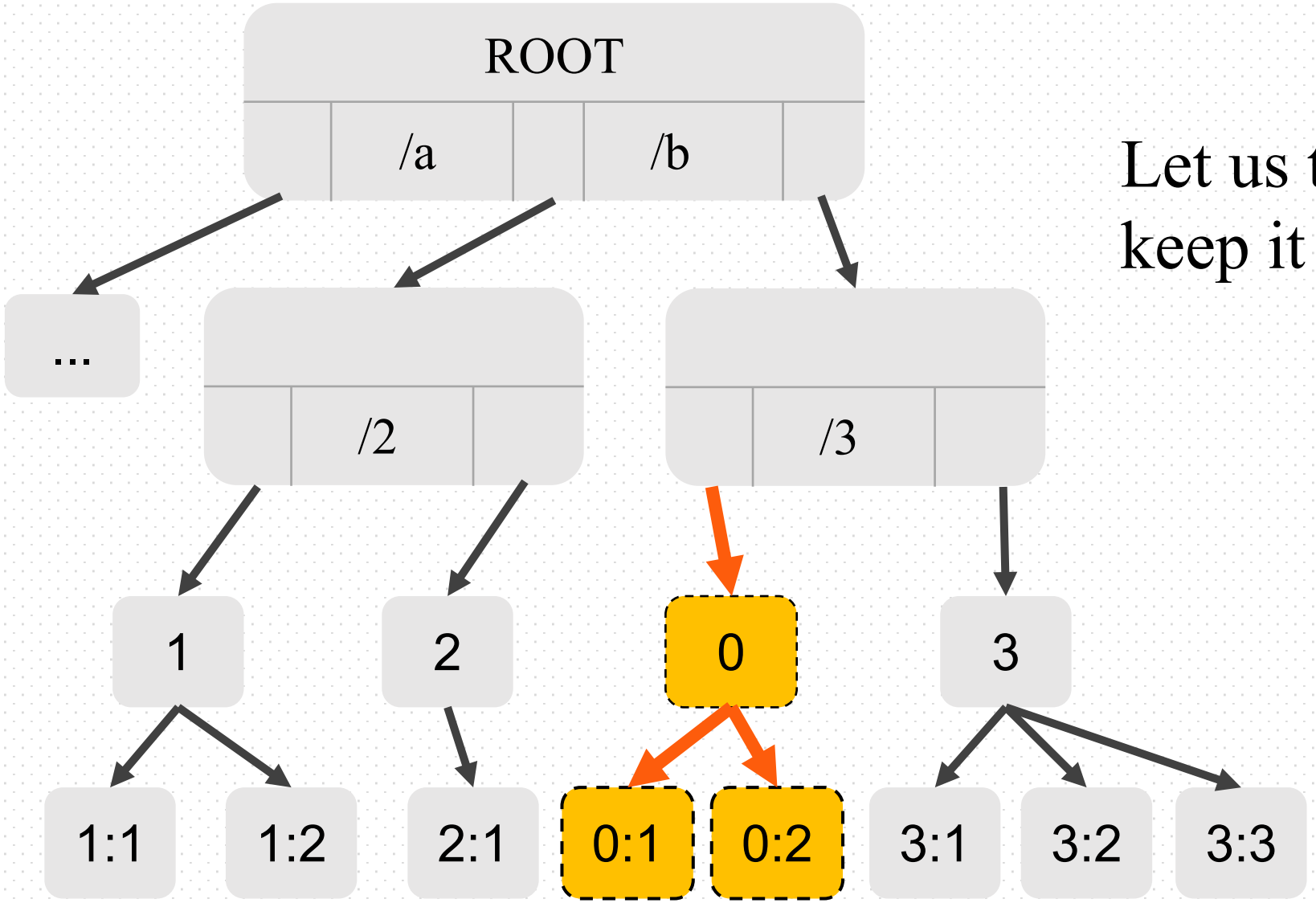
Problem: write amplification



What if we modify this?
We still need to create a data block immediately.

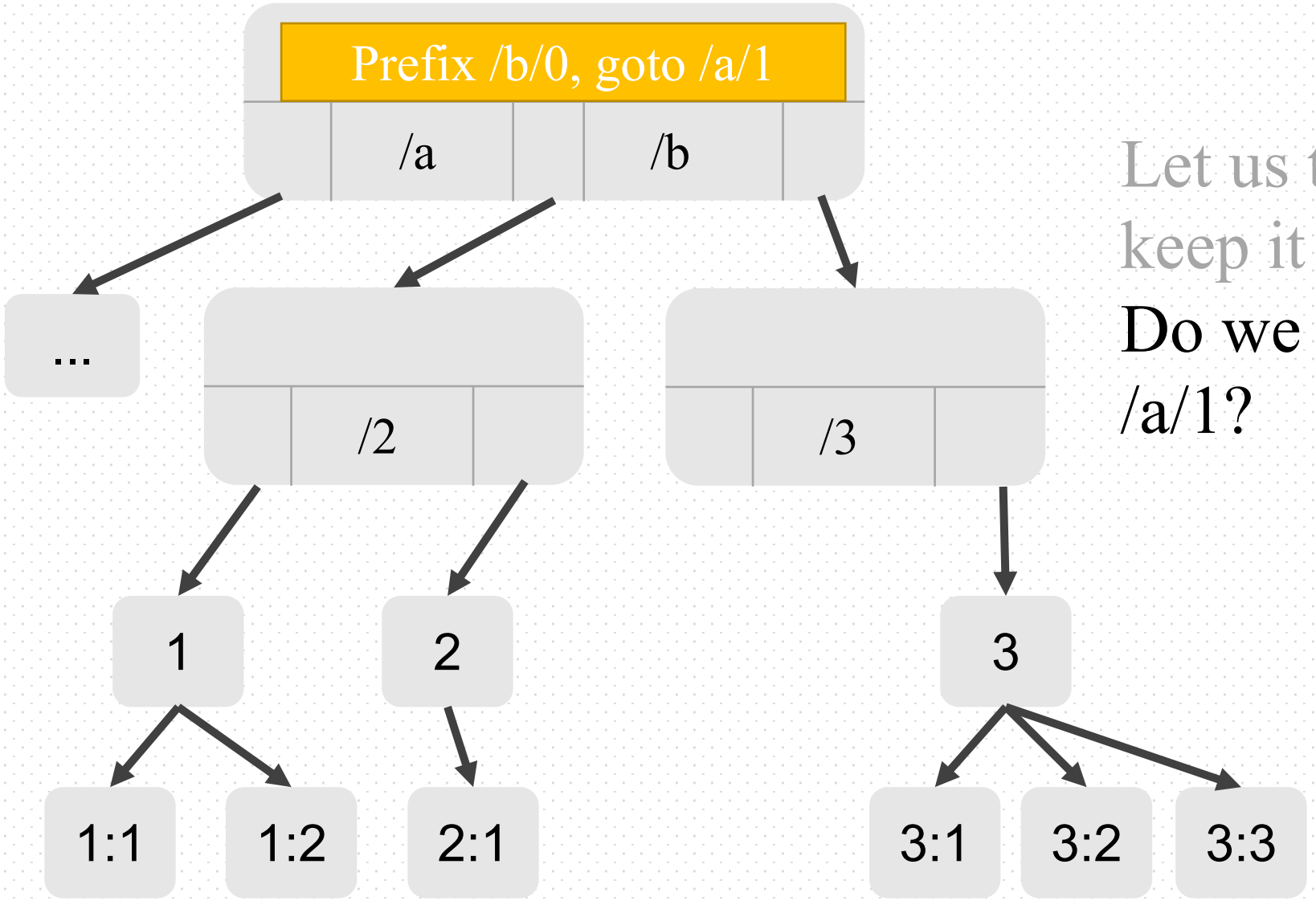
Could we even postpone this event?

GOTO messages



Let us teach the KV-store to keep it in mind rather than do it!

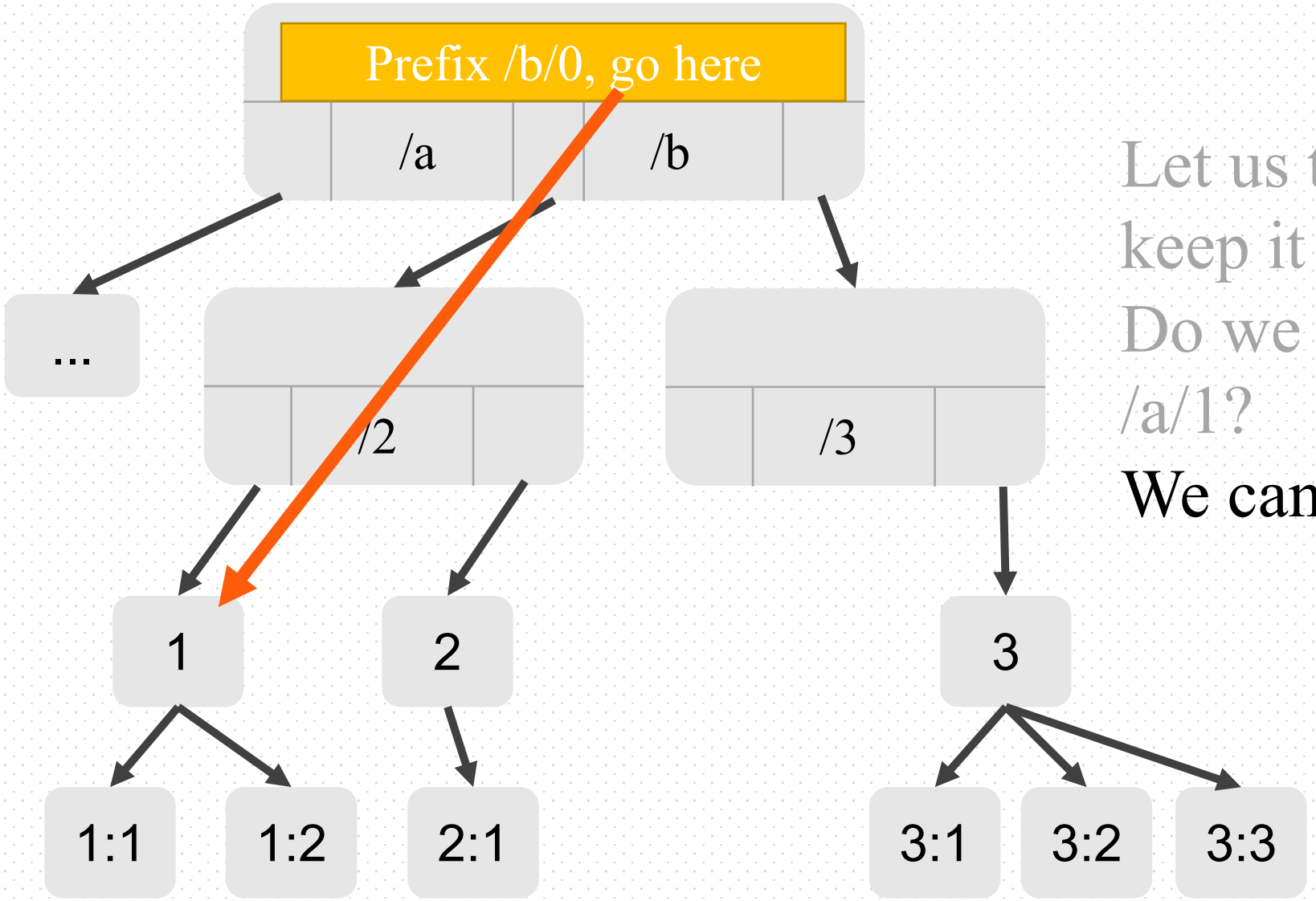
GOTO messages



Let us teach the KV-store to keep it in mind rather than do it!

Do we still needs to go through /a/1?

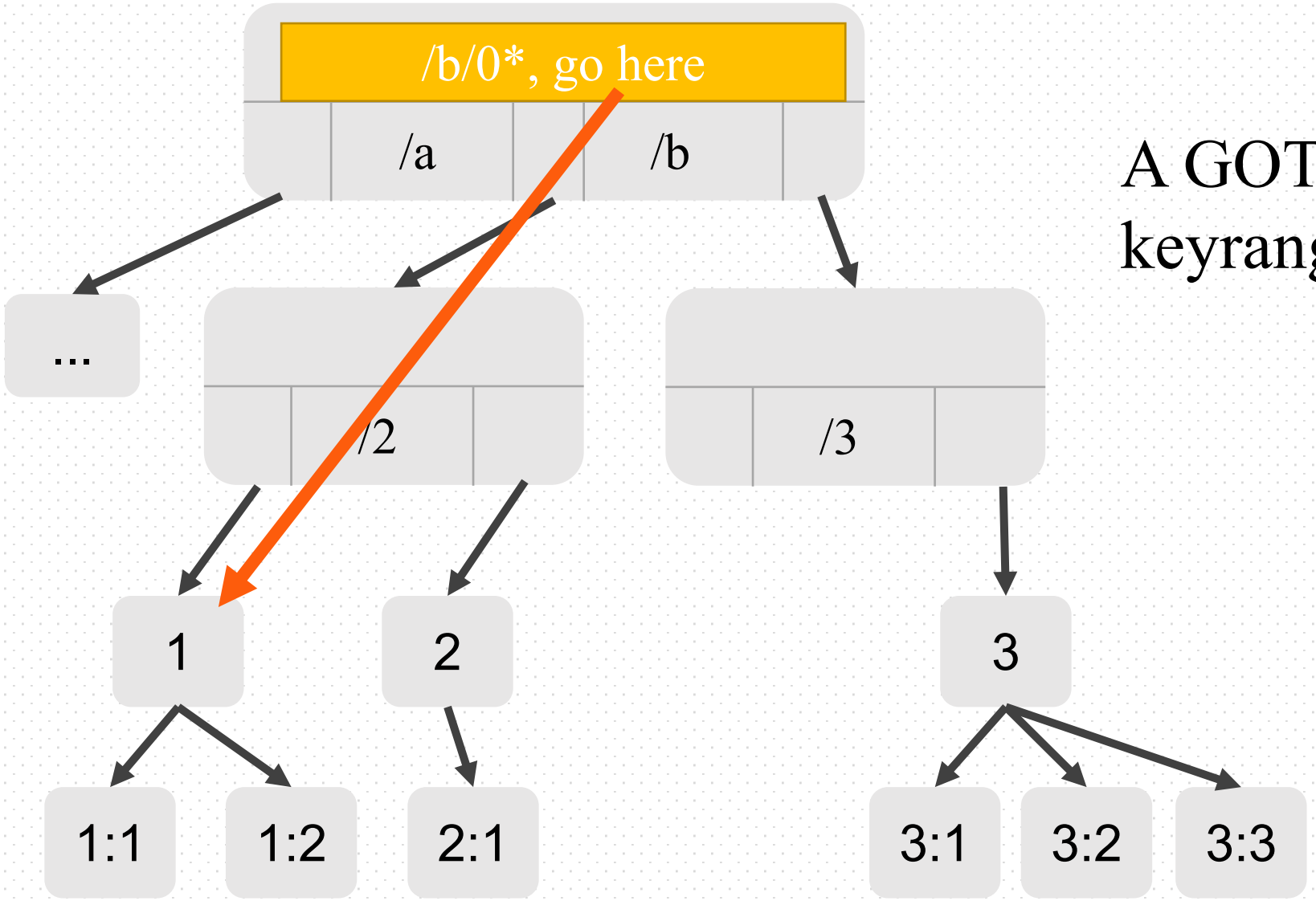
GOTO messages



Let us teach the KV-store to keep it in mind rather than do it! Do we still needs to go through /a/1?

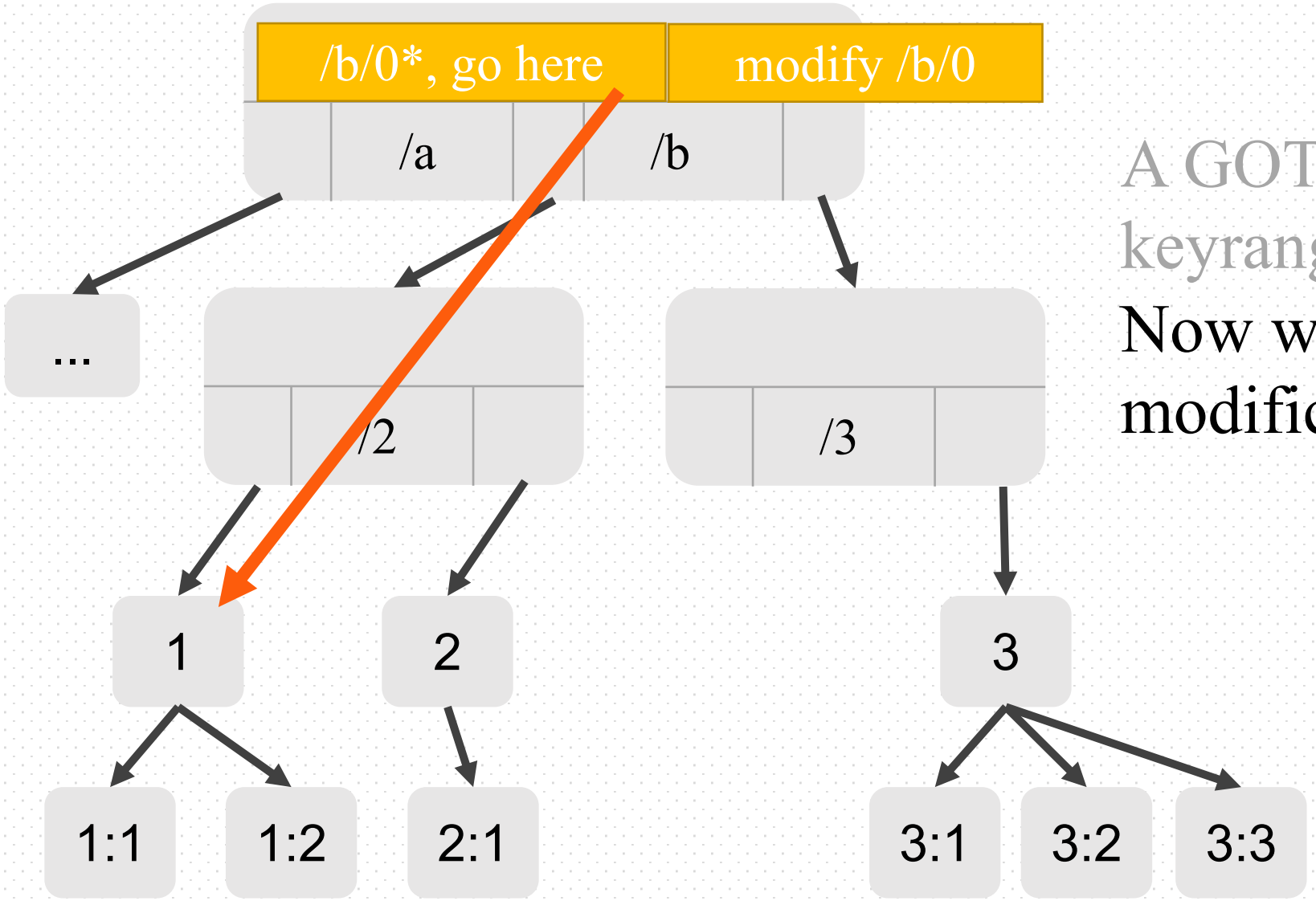
We can just get a pointer here.

GOTO messages



A GOTO message maps:
keyrange (a, b) -> a node

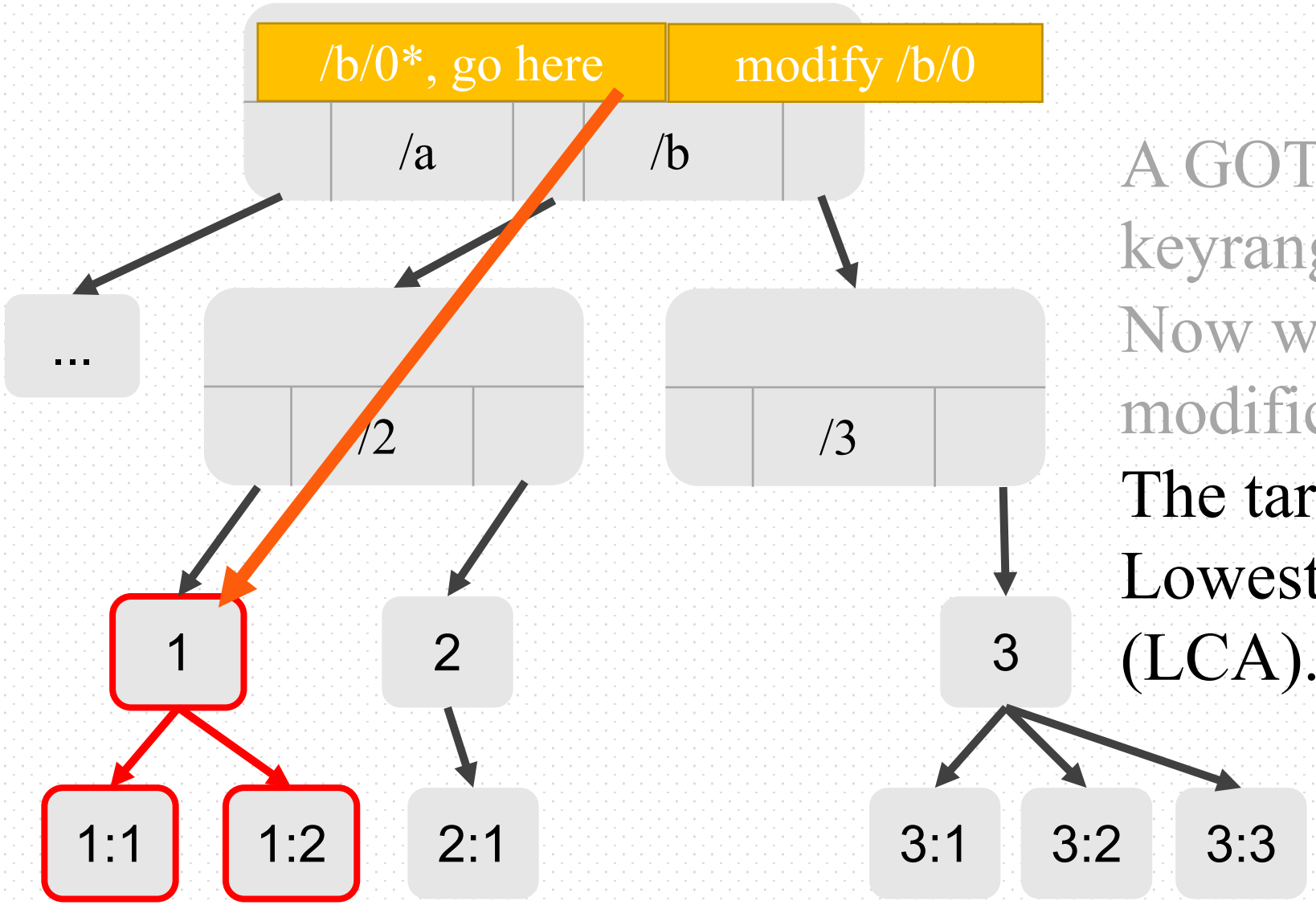
GOTO messages



A GOTO message maps: keyrange (a, b) -> a node

Now we can delay modifications.

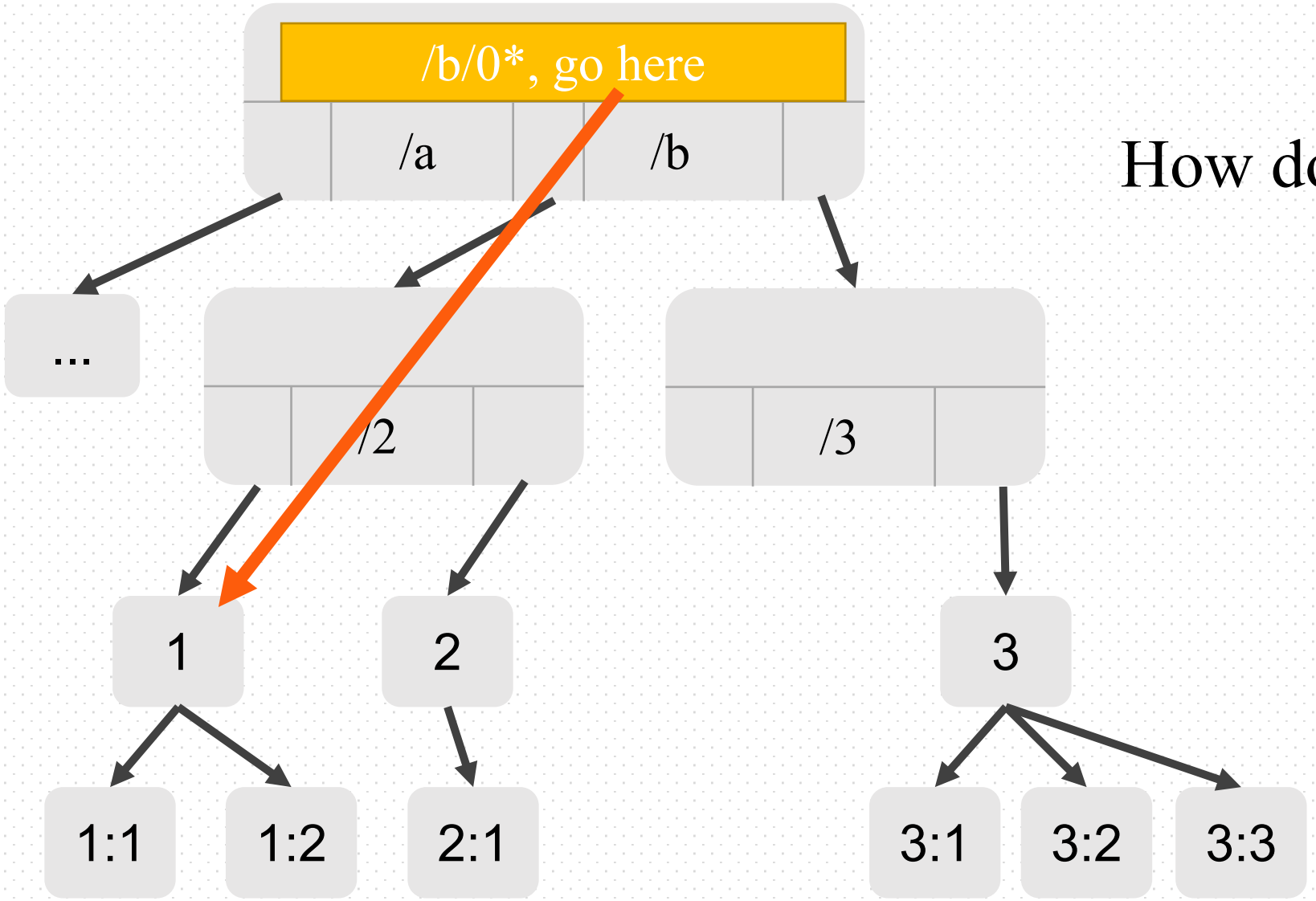
GOTO messages



A GOTO message maps: keyrange (a, b) -> a node
Now we can delay modifications.

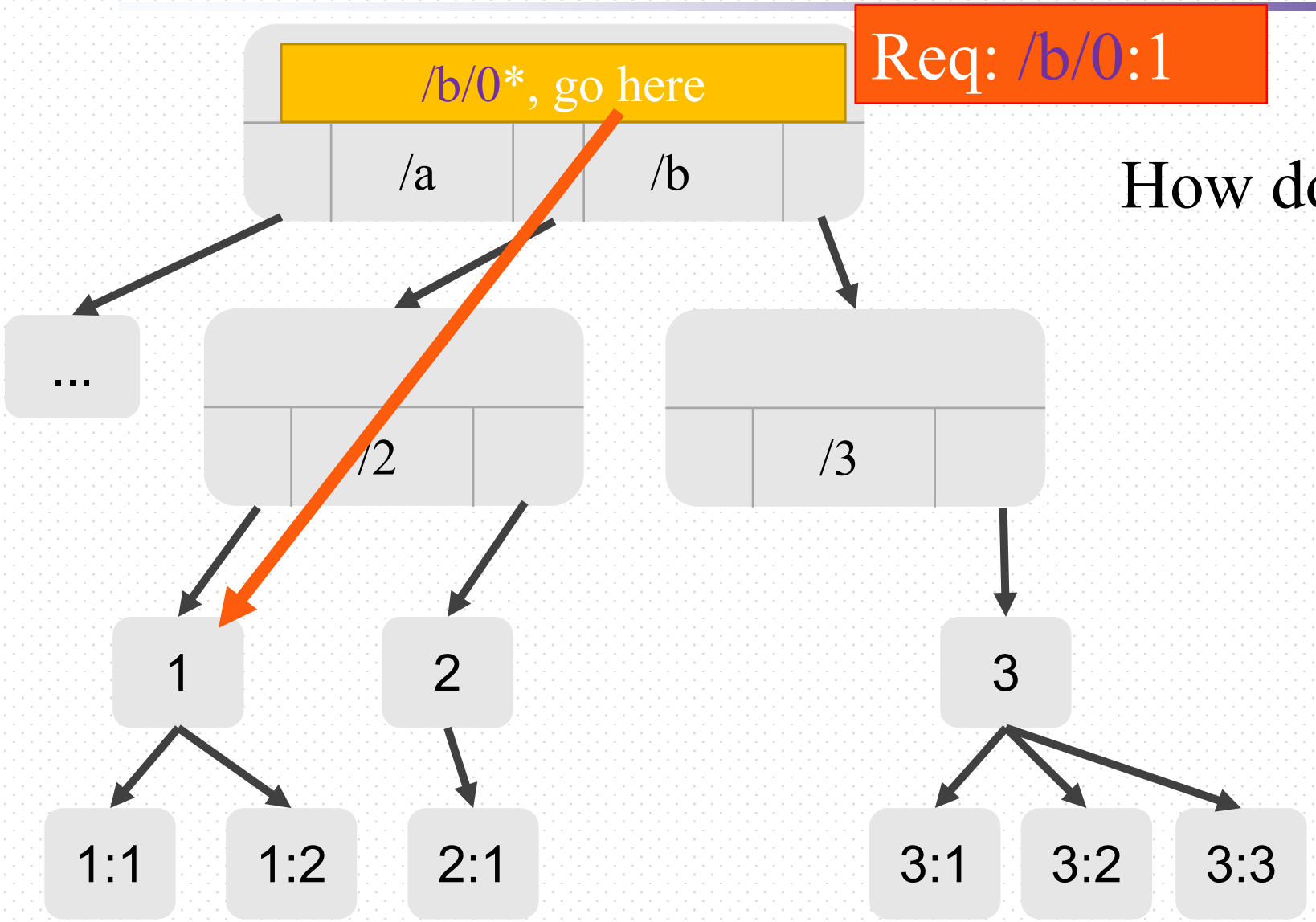
The target node should be the Lowest-Common Ancestor (LCA).

GOTO messages



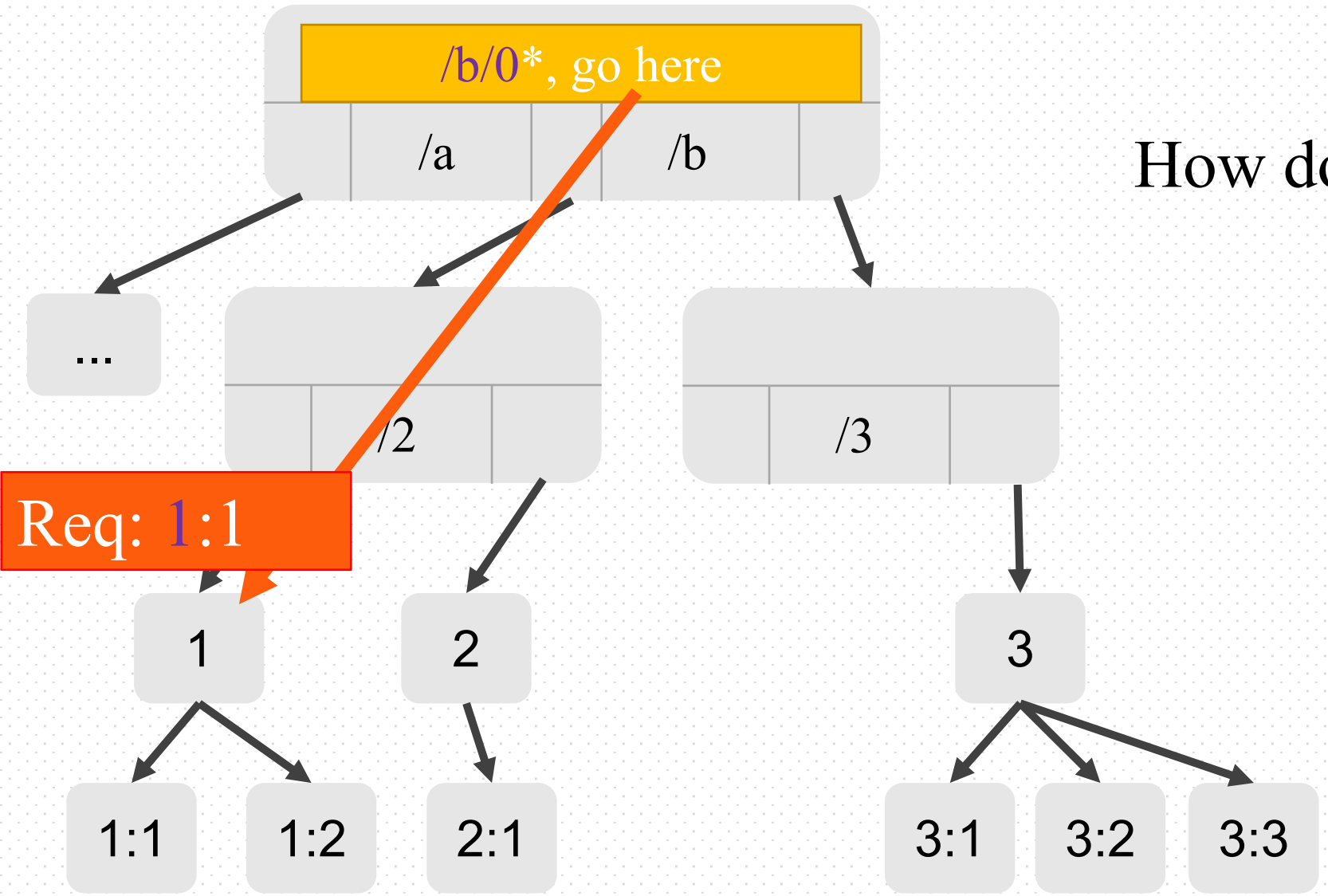
How do we translate requests?

GOTO messages



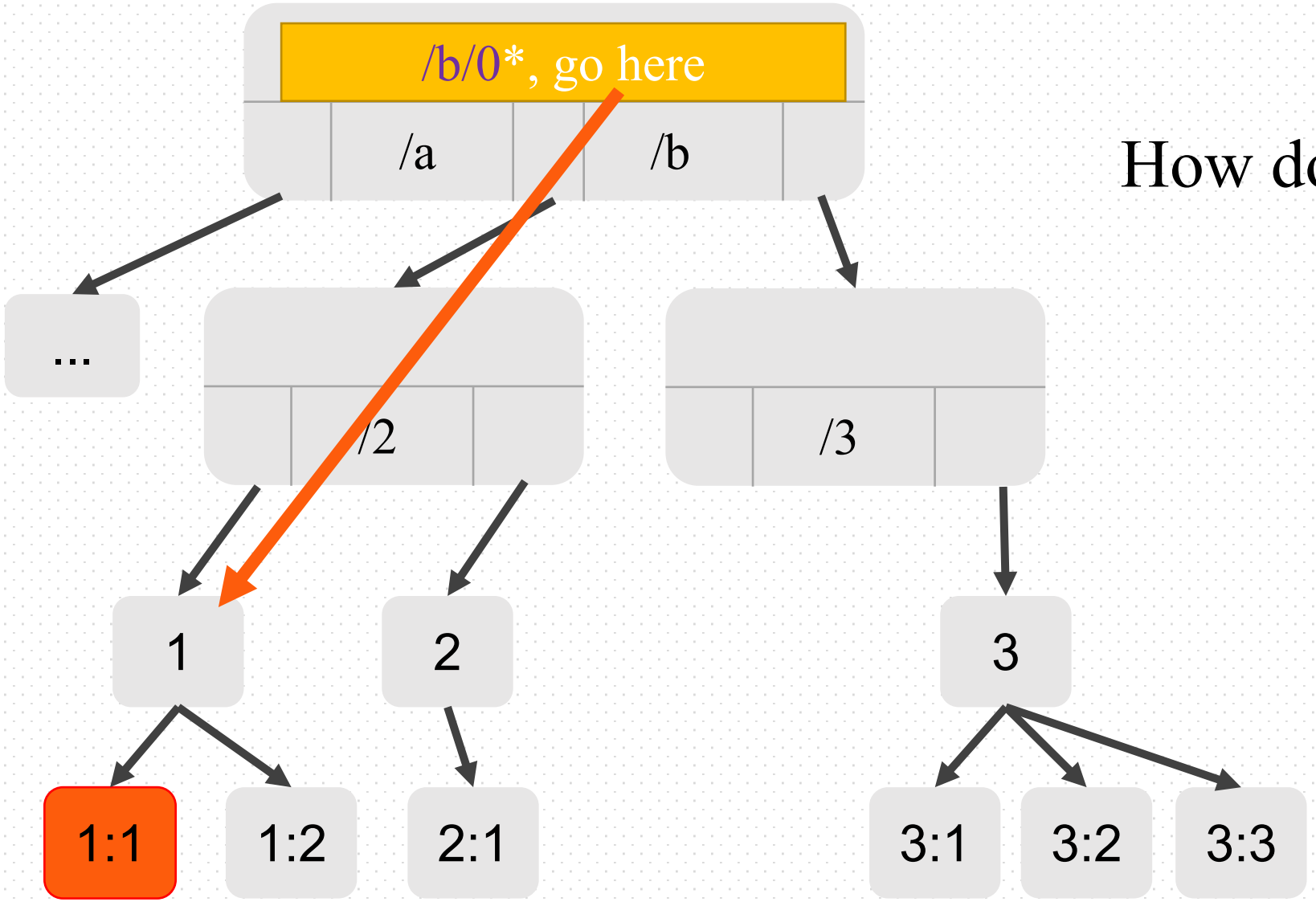
How do we translate requests?

GOTO messages



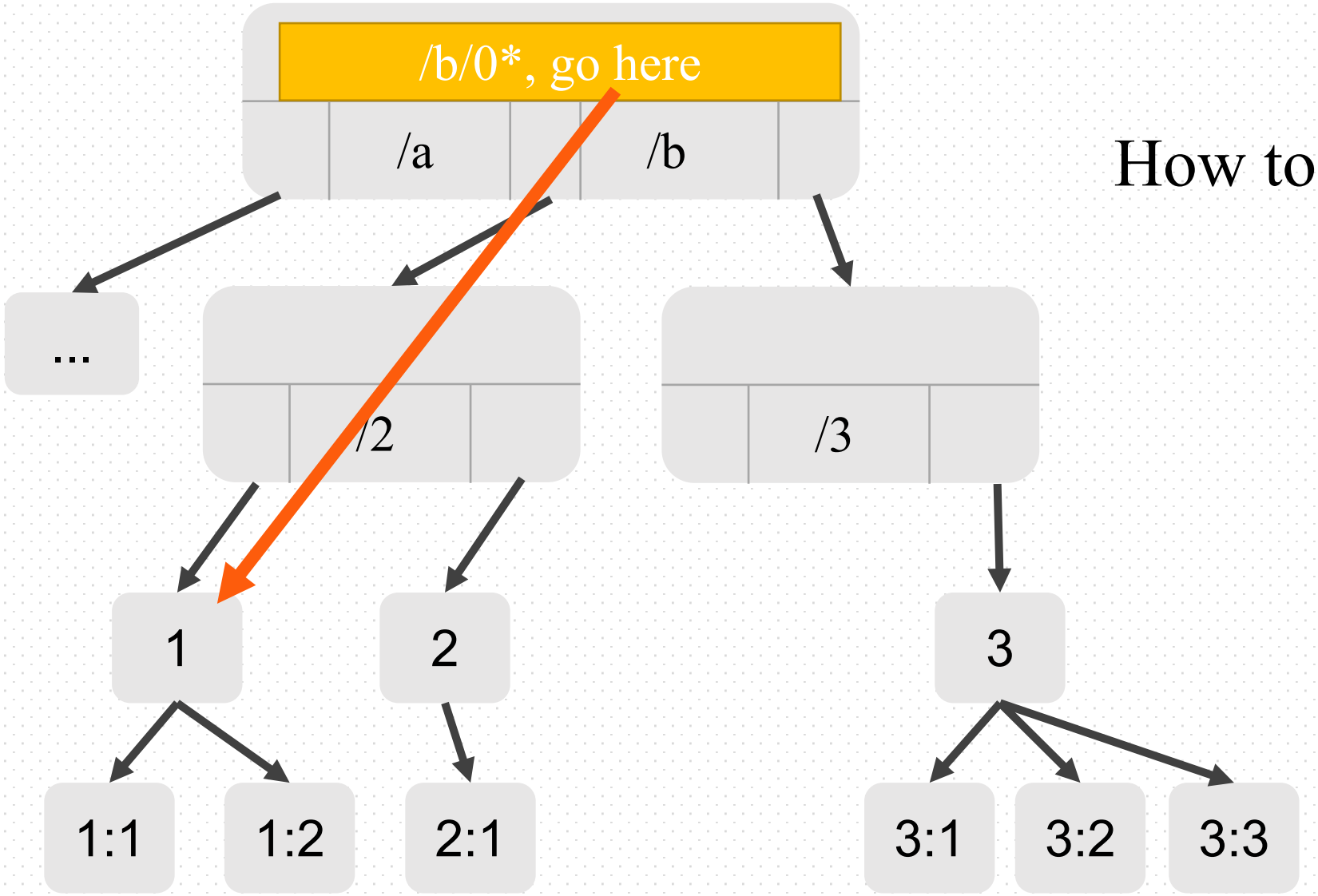
How do we translate requests?

GOTO messages



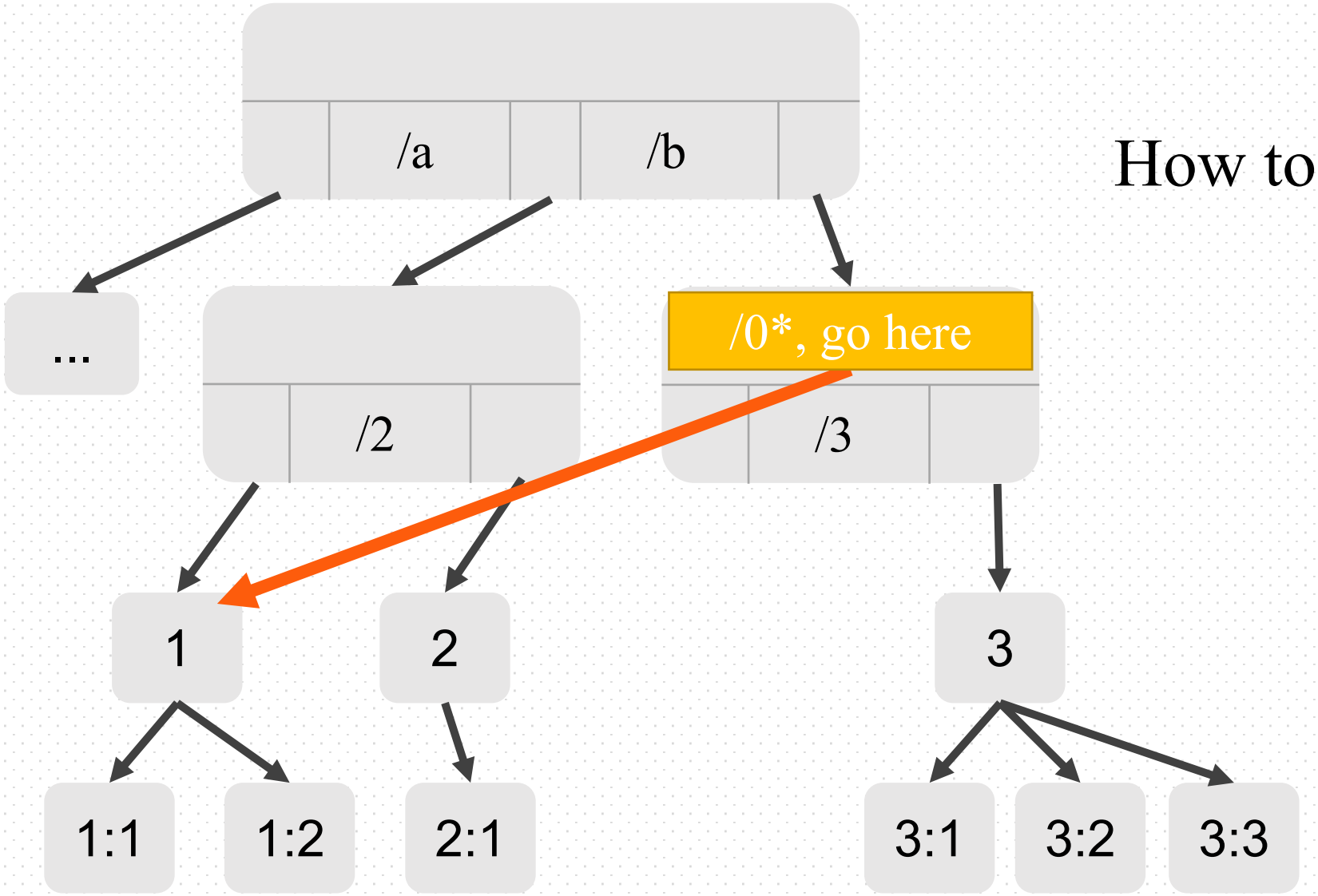
How do we translate requests?

GOTO messages



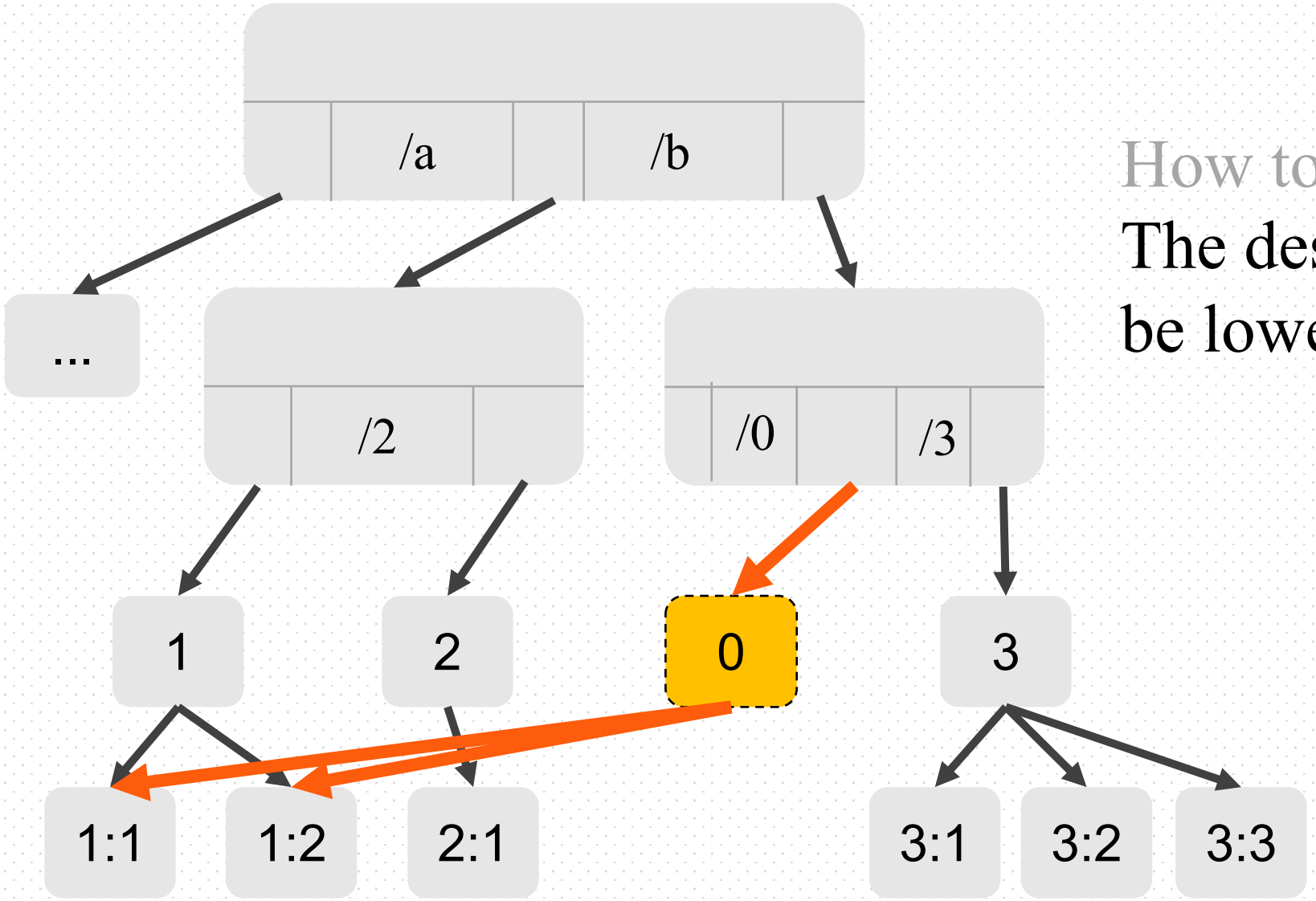
How to flush GOTO messages?

GOTO messages



How to flush GOTO messages?

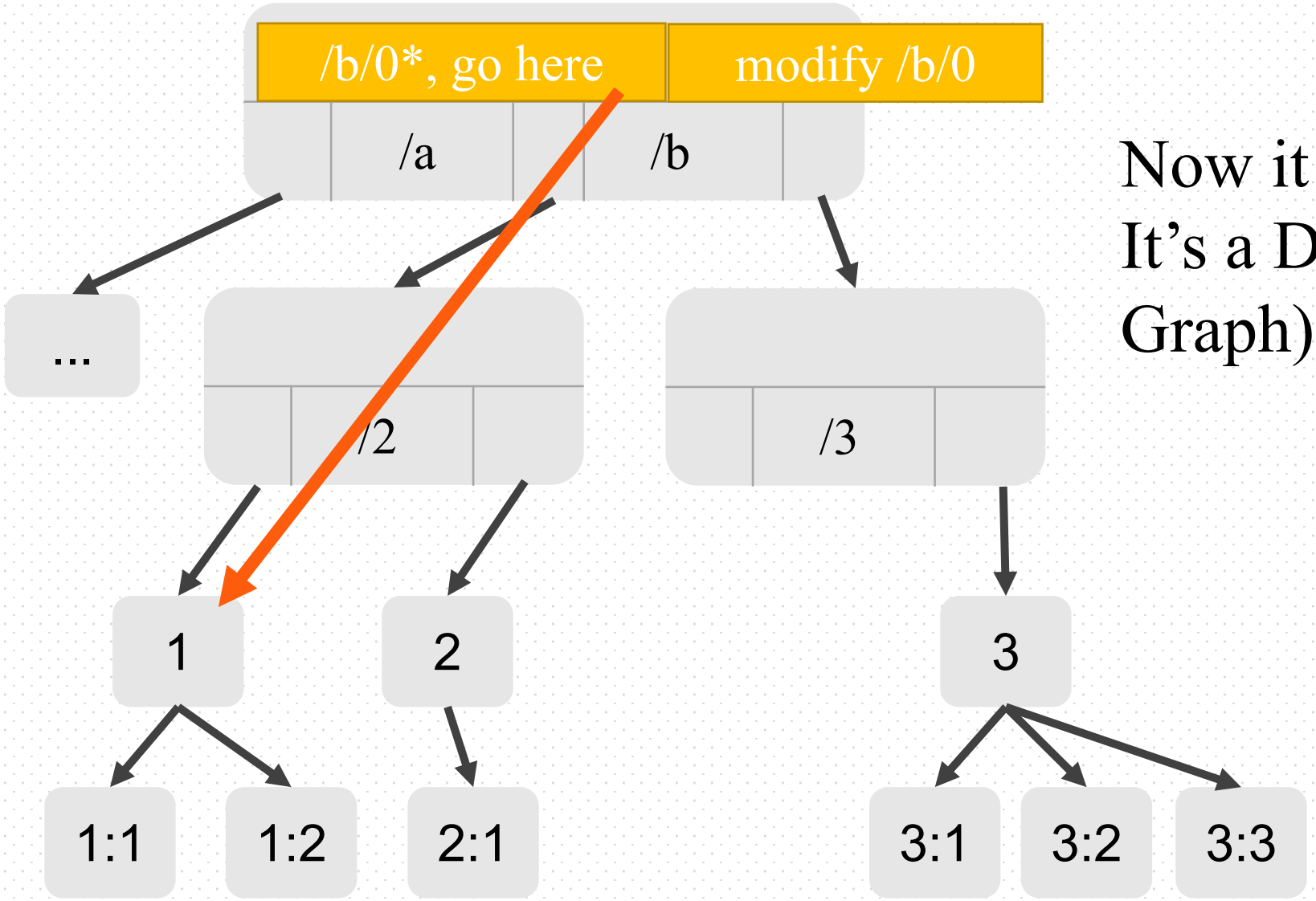
GOTO messages



How to flush GOTO messages?

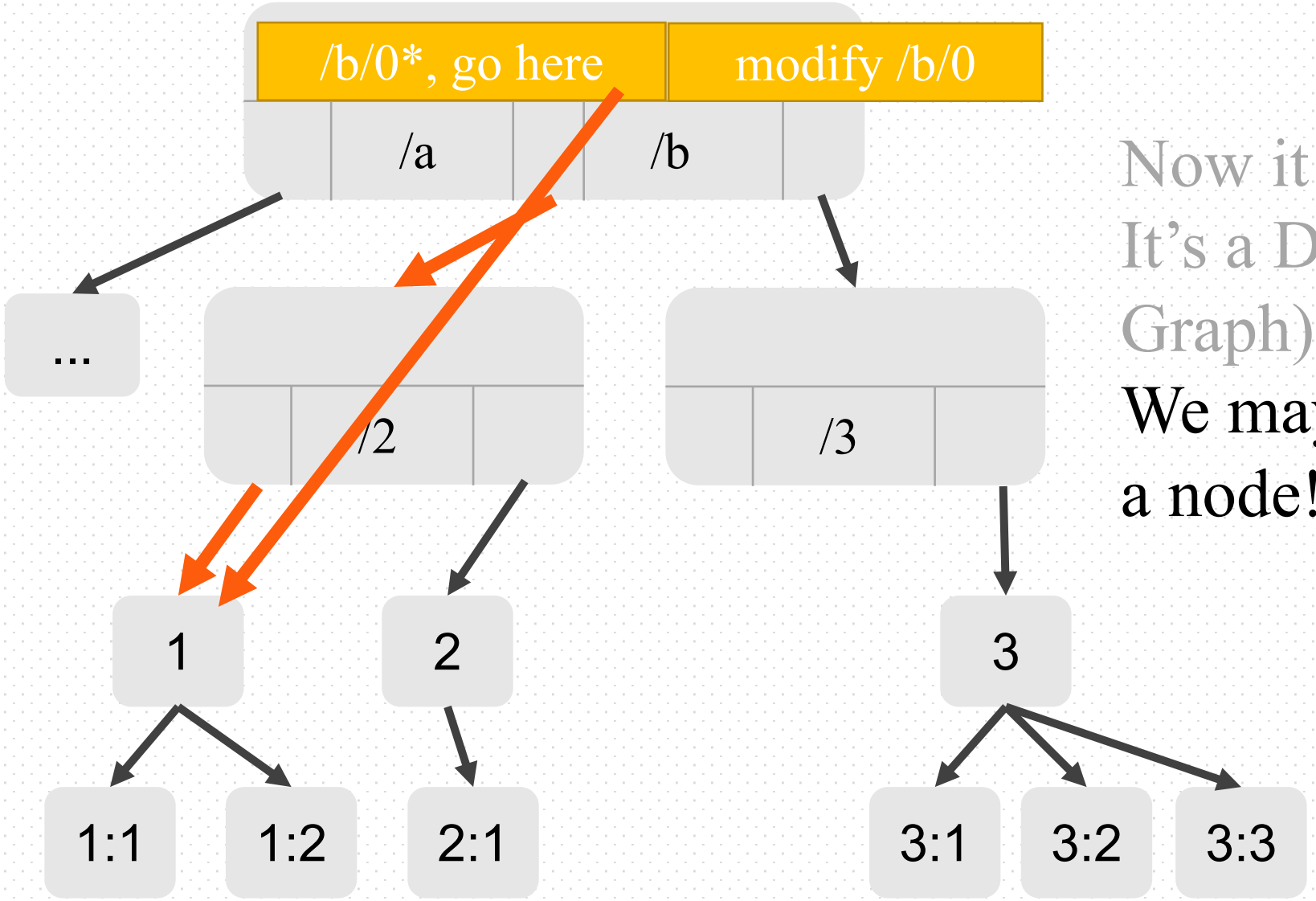
The destination should always be lower than the source node.

Lifted B$^{\varepsilon}$-DAG



Now it is not a “tree” any more. It’s a DAG(Directed Acyclic Graph).

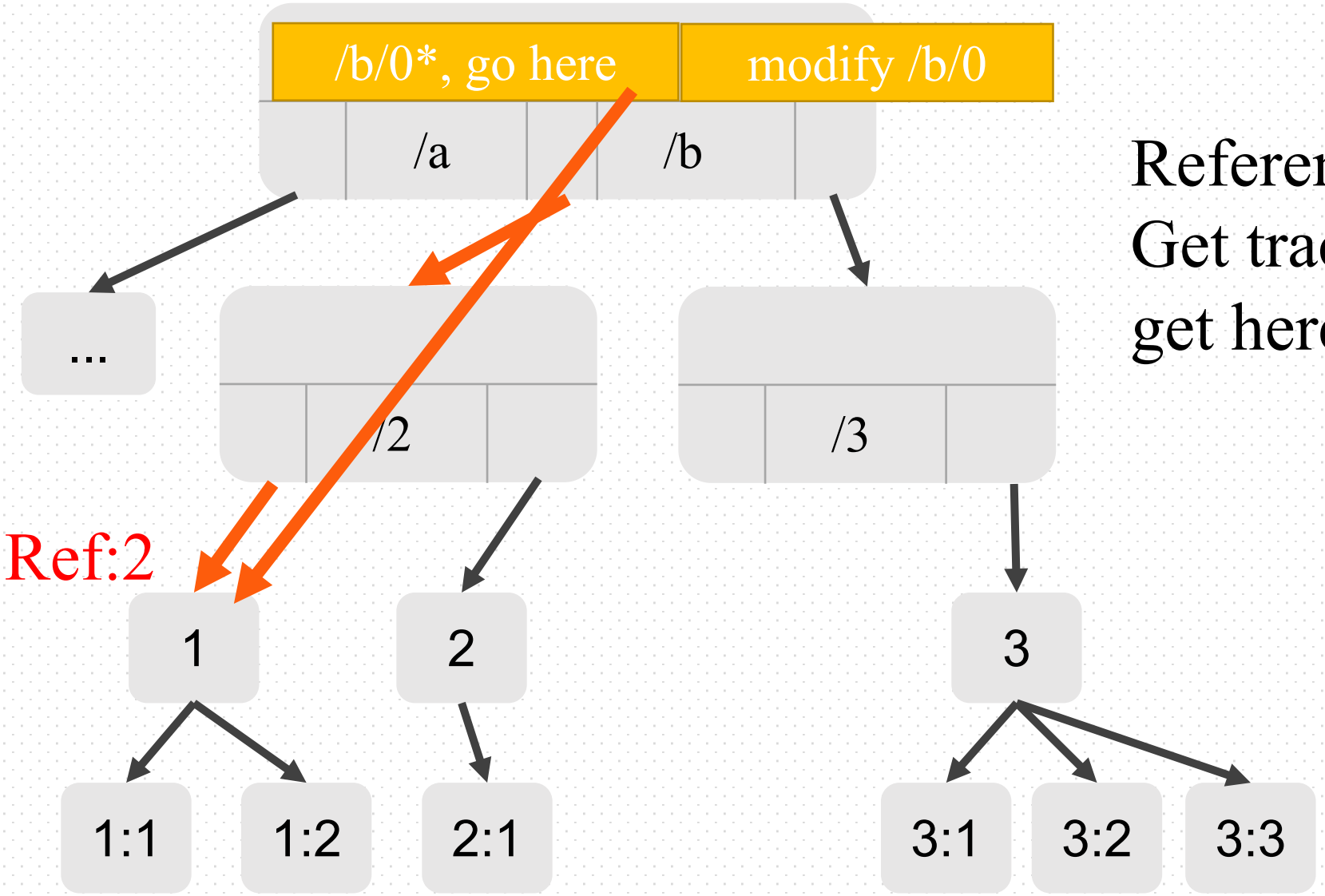
Lifted B$^{\varepsilon}$-DAG



Now it is not a "tree" any more. It's a DAG(Directed Acyclic Graph).

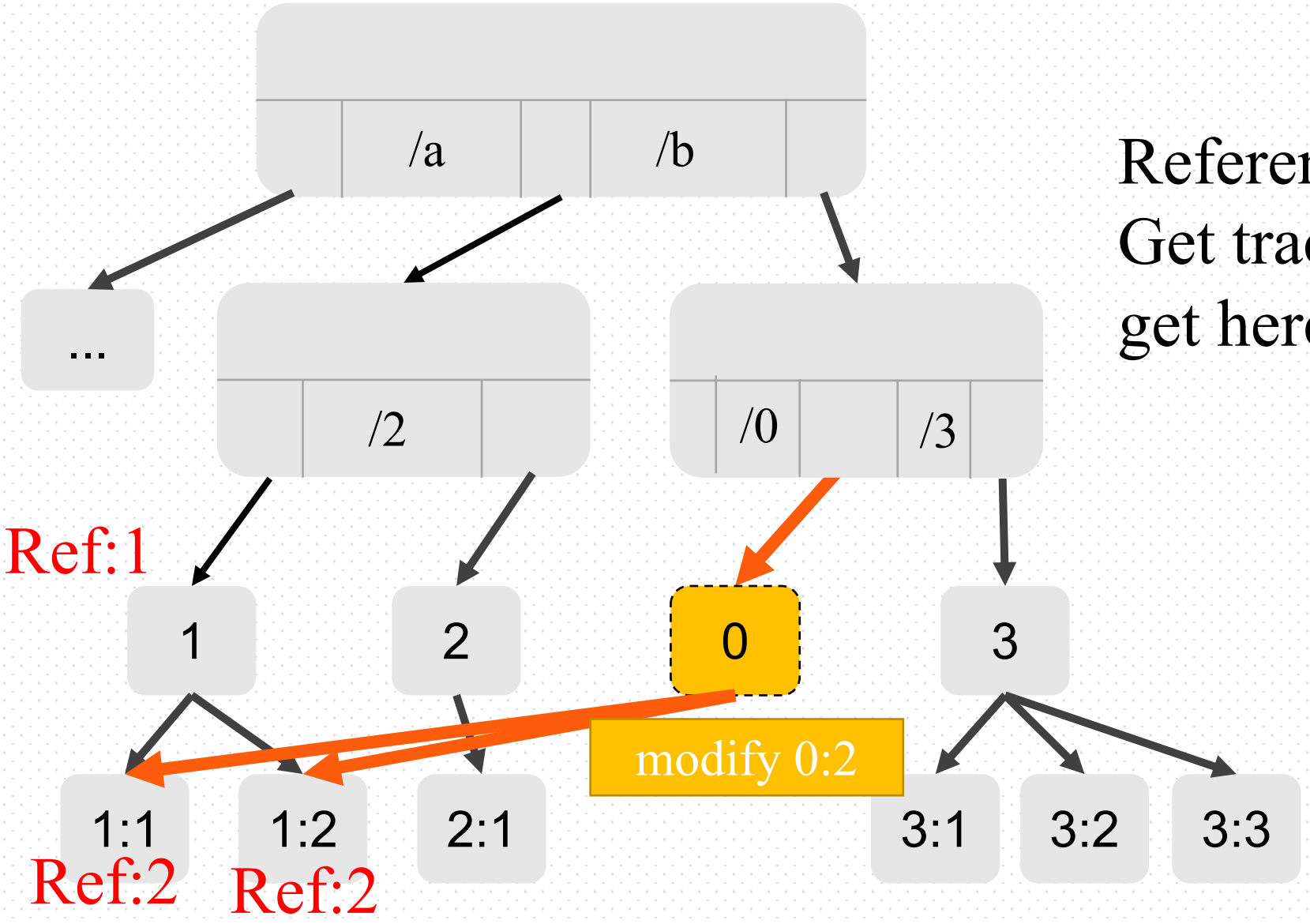
We may have multiple paths to a node!

Lifted B$^{\varepsilon}$-DAG



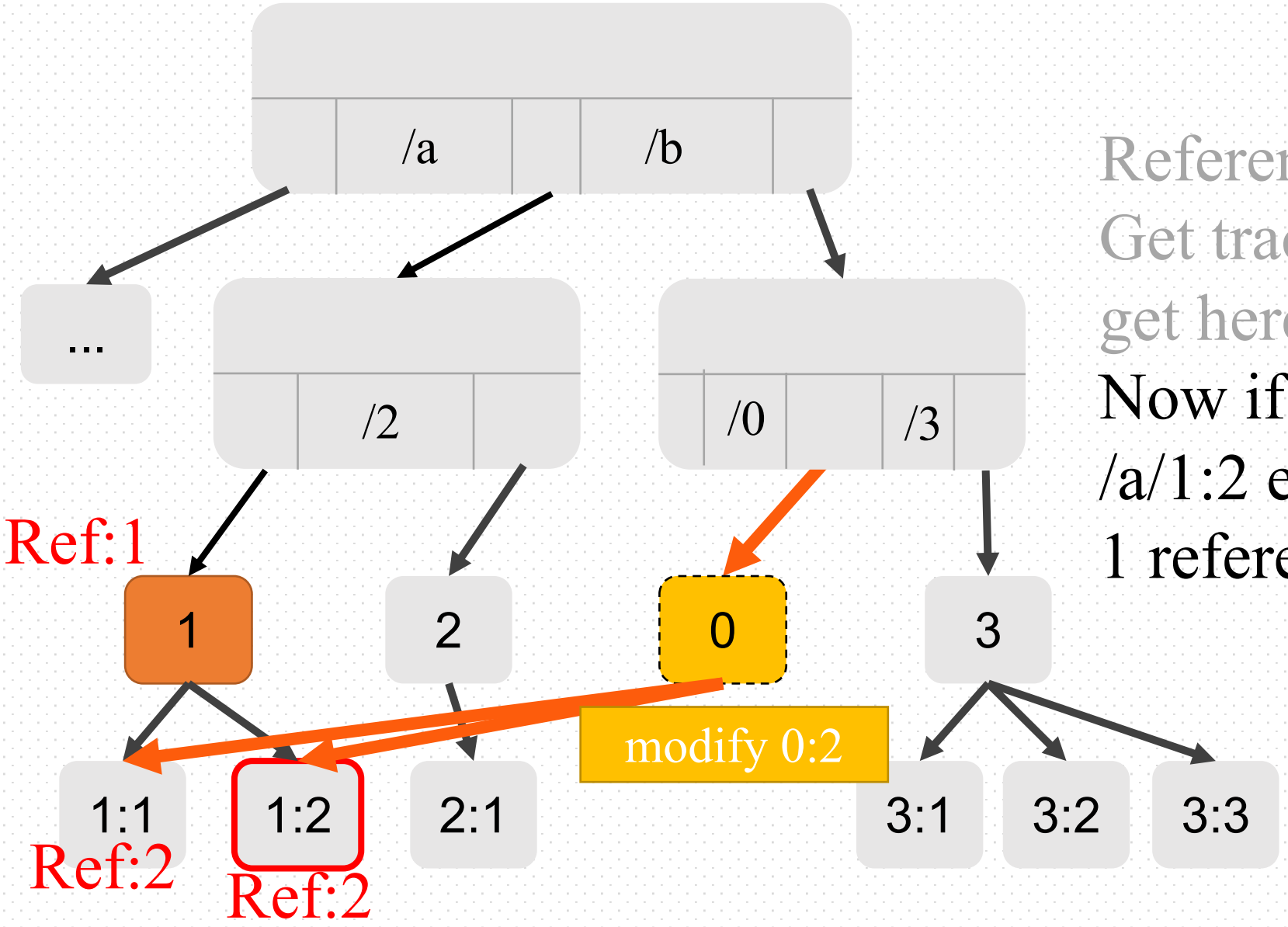
Reference counting:
Get tracked of how many nodes get here.

Lifted B$^{\varepsilon}$-DAG



Reference counting:
Get tracked of how many nodes get here.

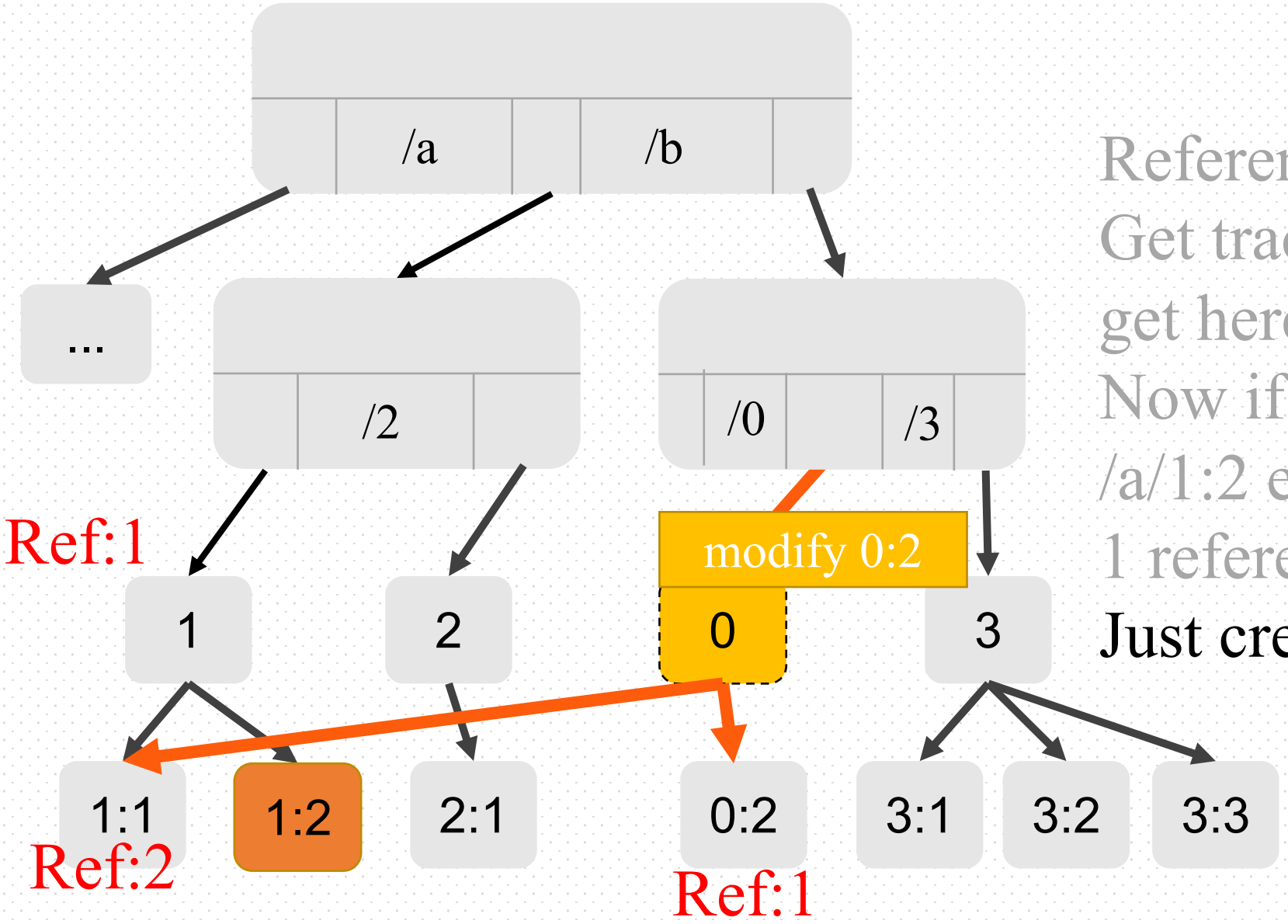
Lifted B$^{\varepsilon}$-DAG



Reference counting:
Get tracked of how many nodes get here.

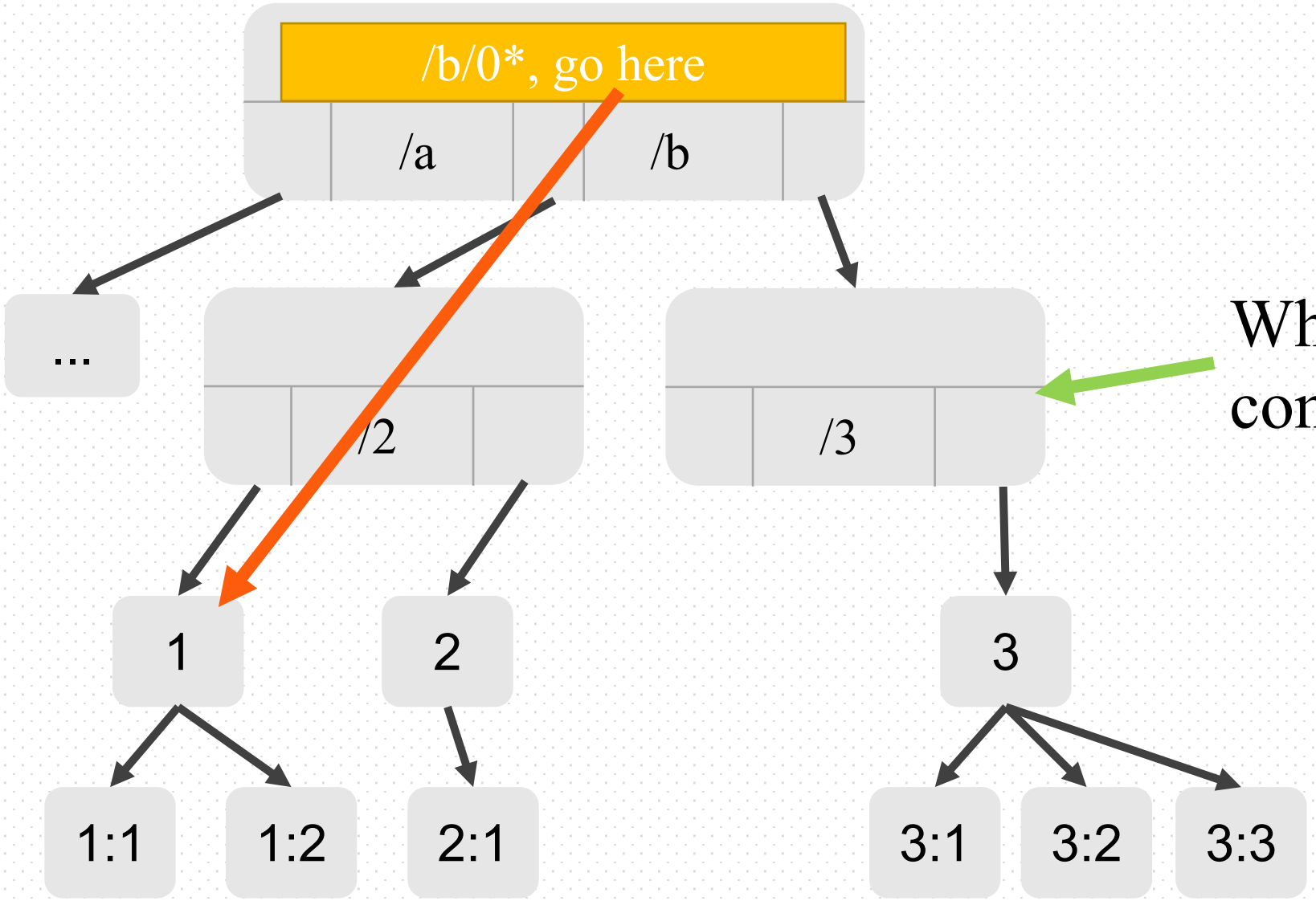
Now if abundant writes on /a/1:2 exists? 1:2 has more than 1 references.

Lifted B$^{\varepsilon}$-DAG



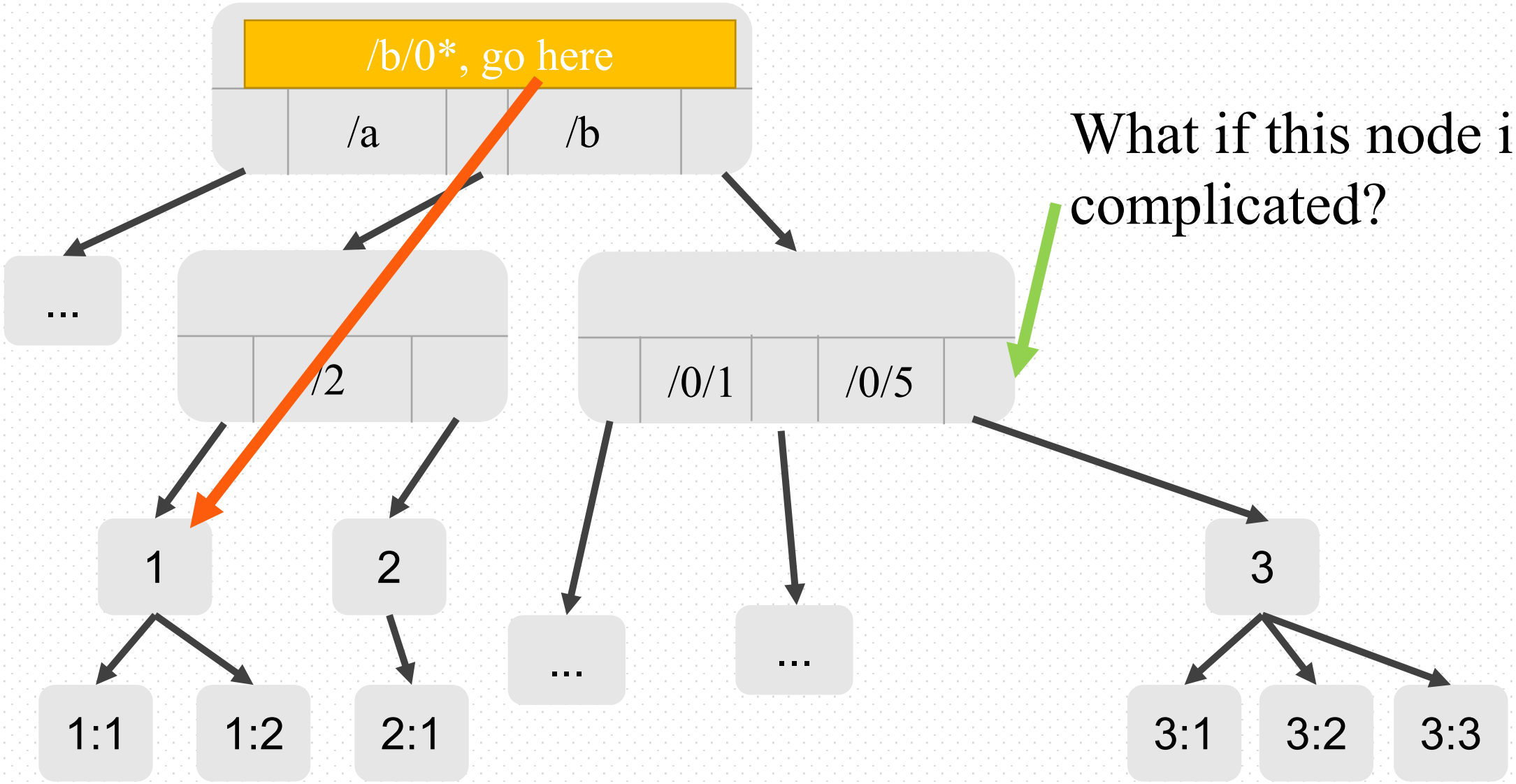
Reference counting:
Get tracked of how many nodes get here.
Now if abundant writes on /a/1:2 exists? 1:2 has more than 1 references.
Just create a new 1:2.

Handling Fringe Nodes

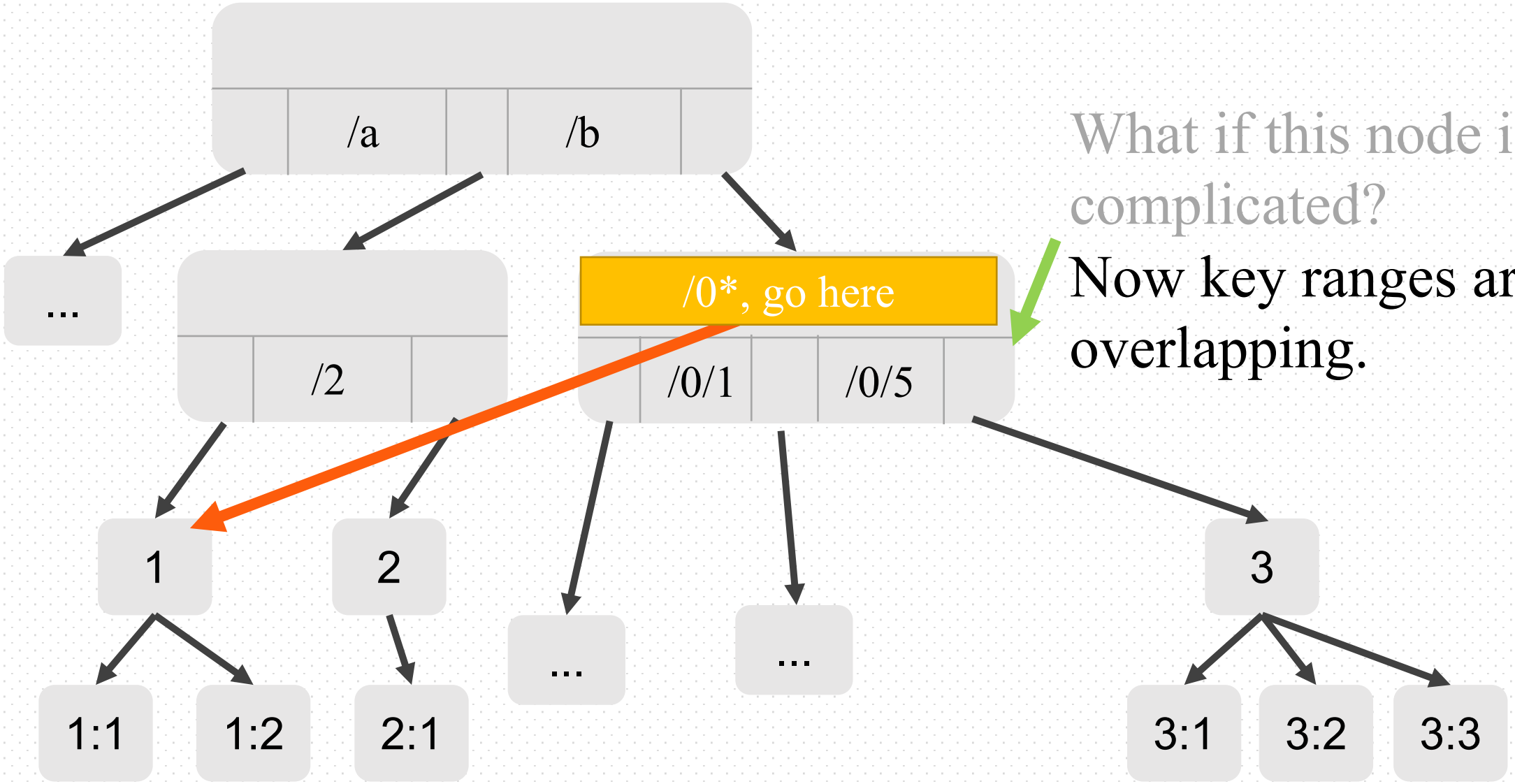


What if this node is more complicated?

Handling Fringe Nodes



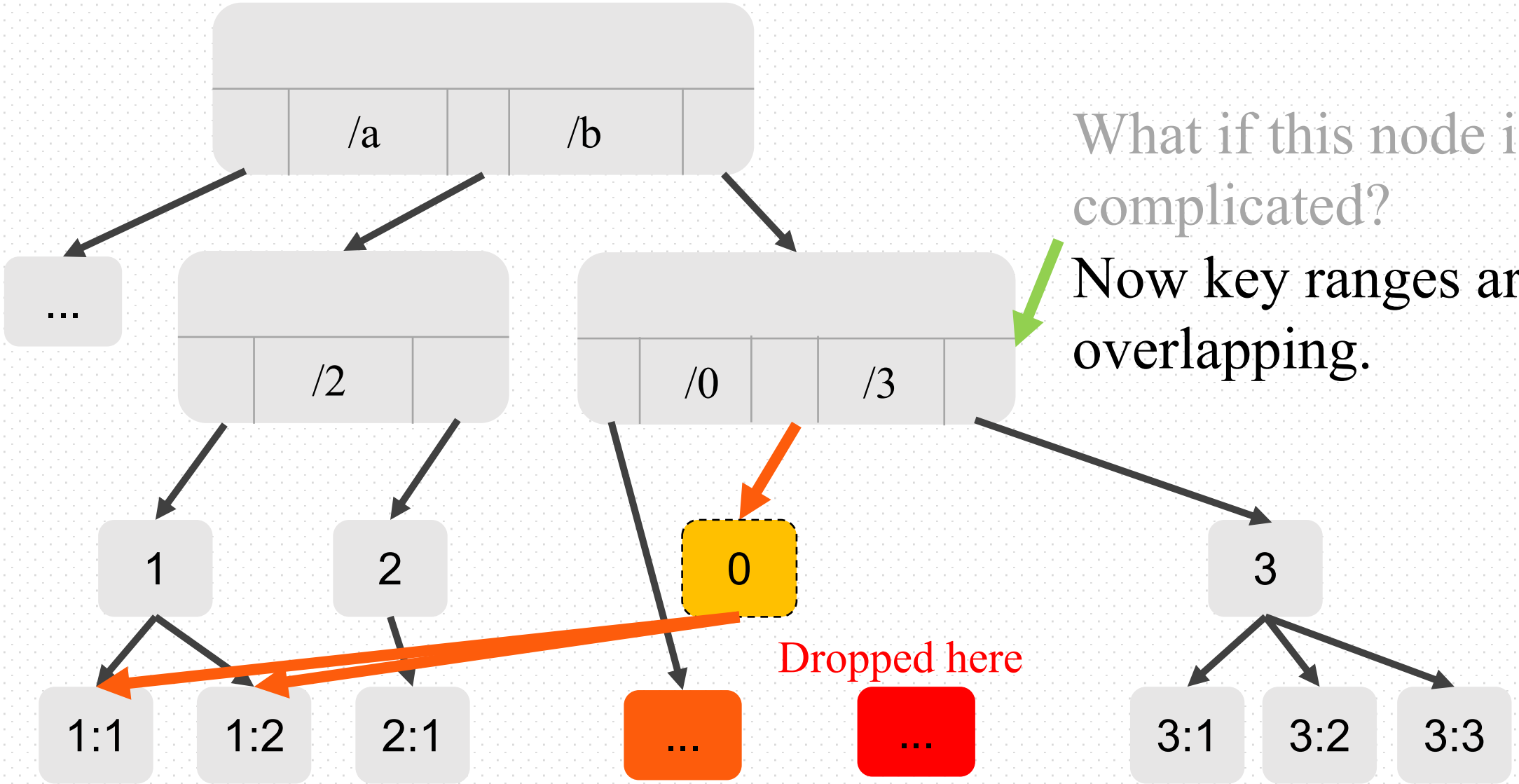
Handling Fringe Nodes



What if this node is more complicated?

Now key ranges are overlapping.

Handling Fringe Nodes



What if this node is more complicated?

Now key ranges are overlapping.

Implementation and Optimization

Preferential splitting

- When splitting leaf nodes, select the key that maximizes the common prefix of the leaf, instead of the middle key in BetrFS, as pivots.
- Since data in the same file share the same prefix (as do files in the same directory), preferential splitting reduces the likelihood of having fringe nodes in a clone.

Node reclamation

- Reclaim nodes with reference 0.

- A dedicated thread.

Background cleaning

- Flush messages for frequently queried items down the tree, for better query performance.

- A dedicated thread.

Has BεtrFS done “Nimble Copy”?

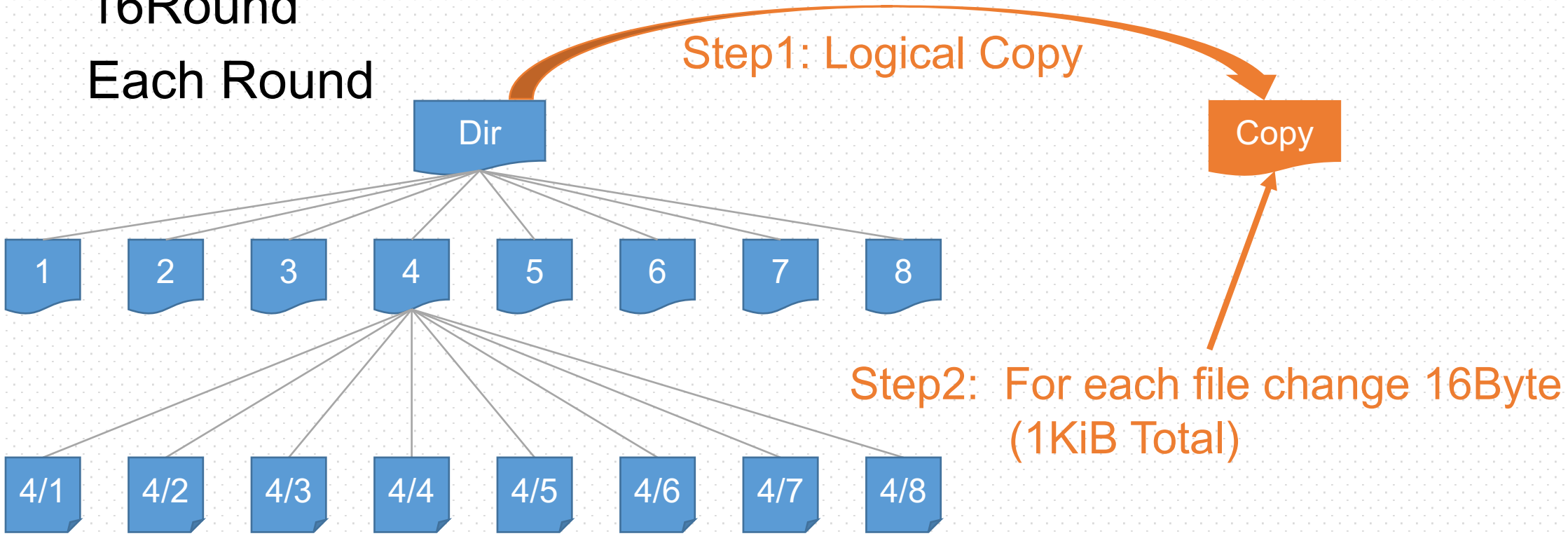
How to evaluate?

- Is BεtrFS good enough than before?
- Is BεtrFS worse than before?
- Is BεtrFS useful for real world?
  - General/Specialized workload

Workload

16Round

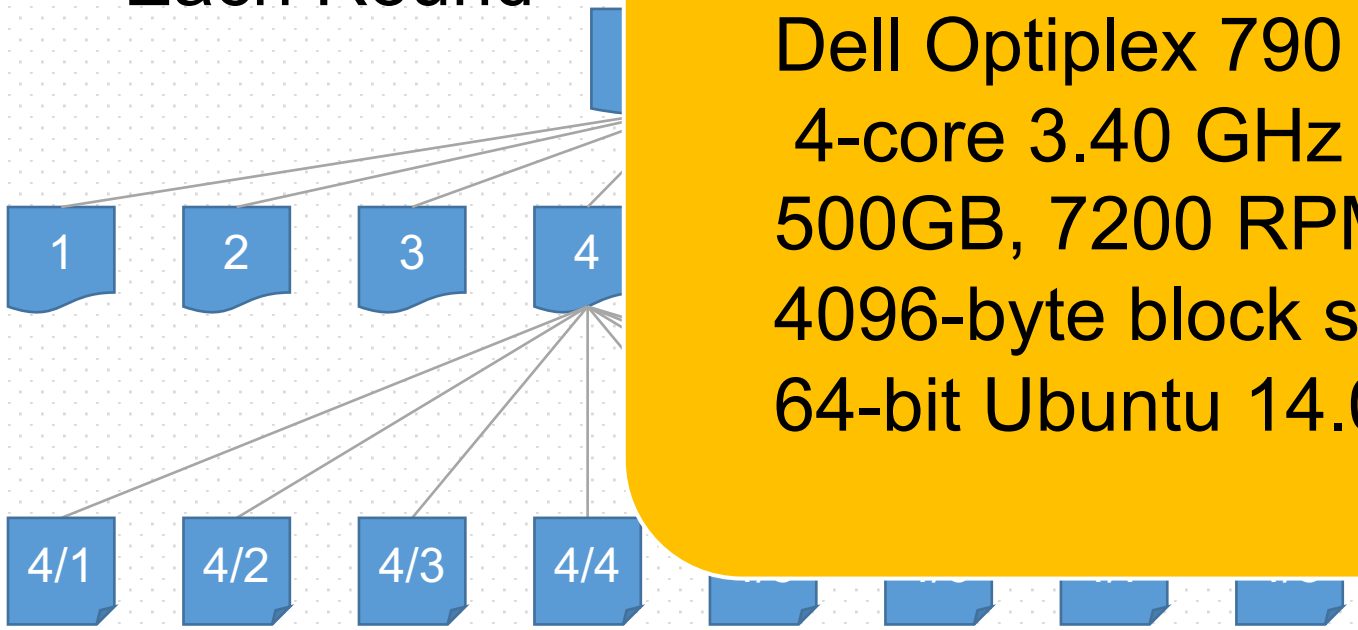
Each Round



Workload

16Round

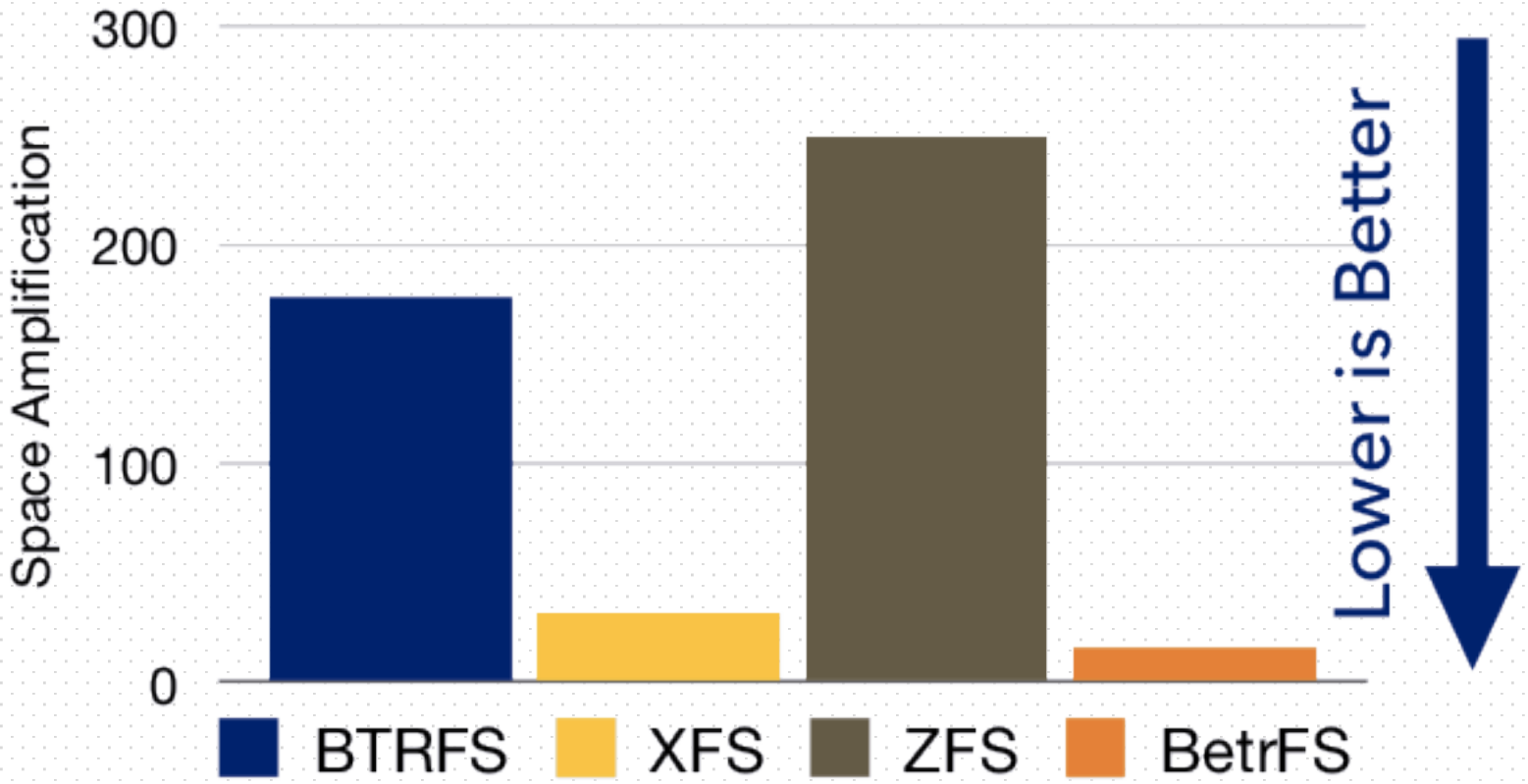
Each Round



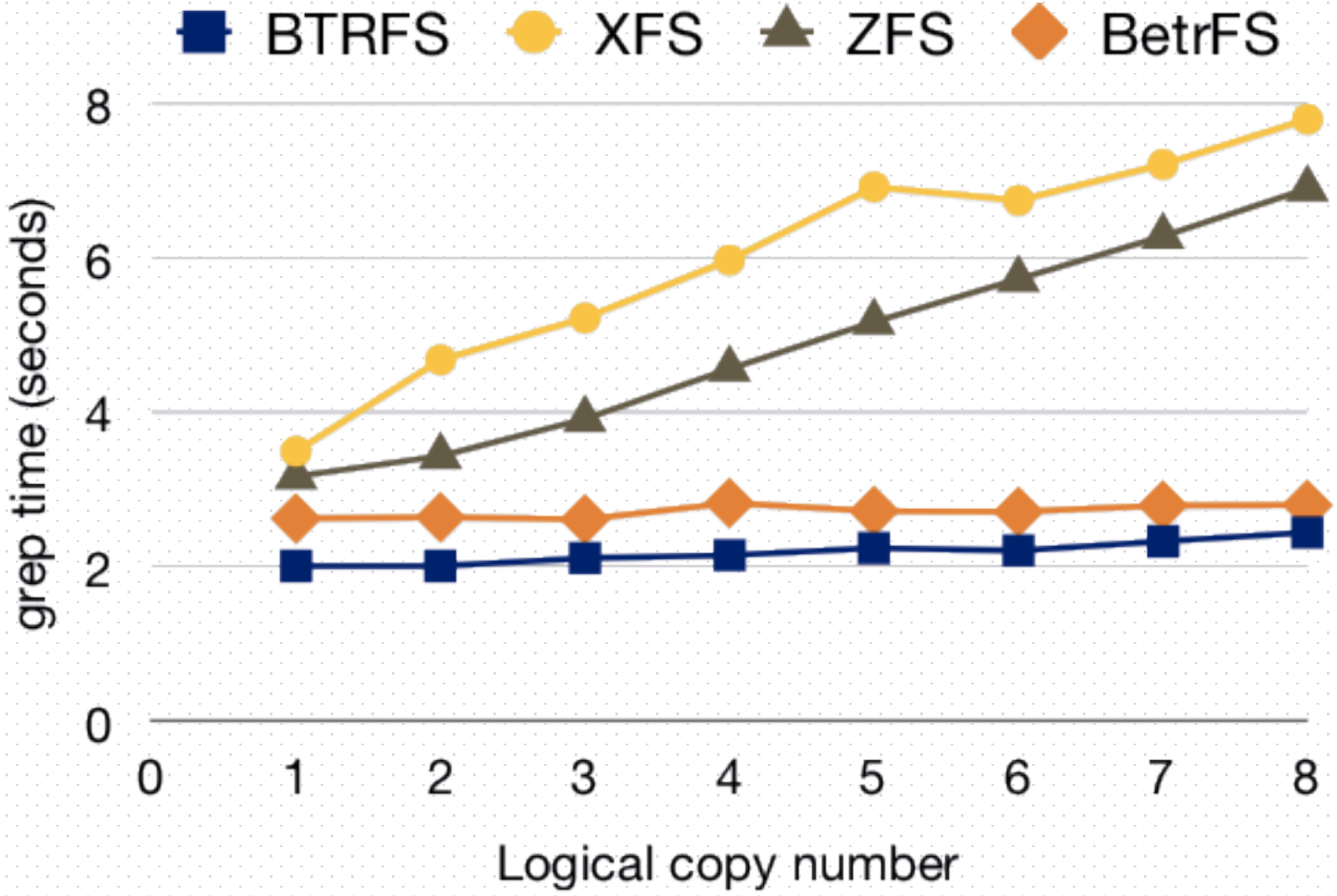
Dell Optiplex 790
4-core 3.40 GHz Intel Core i7 CPU
500GB, 7200 RPM SATA disk
4096-byte block size
64-bit Ubuntu 14.04.5

16Byte

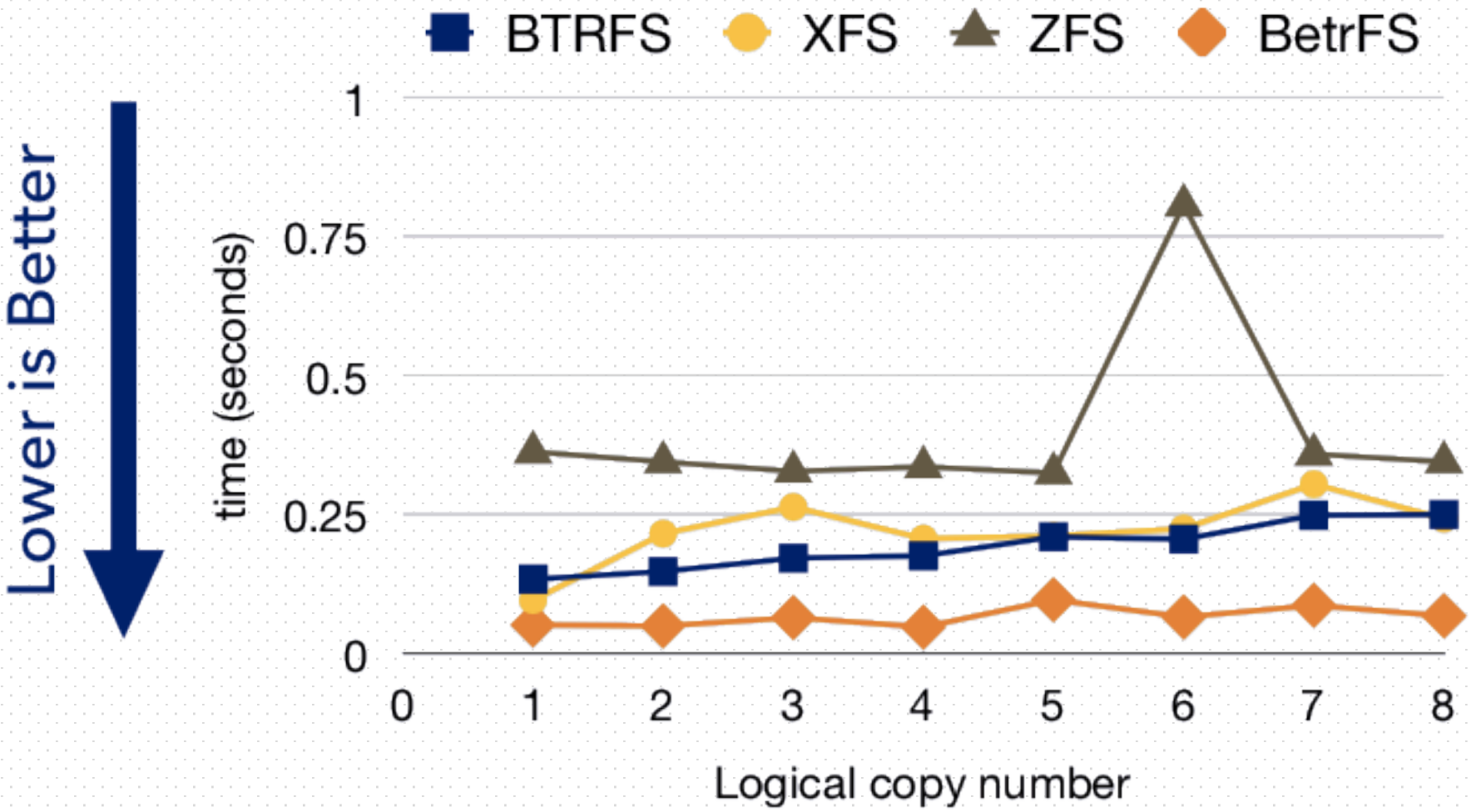
Write Amplification



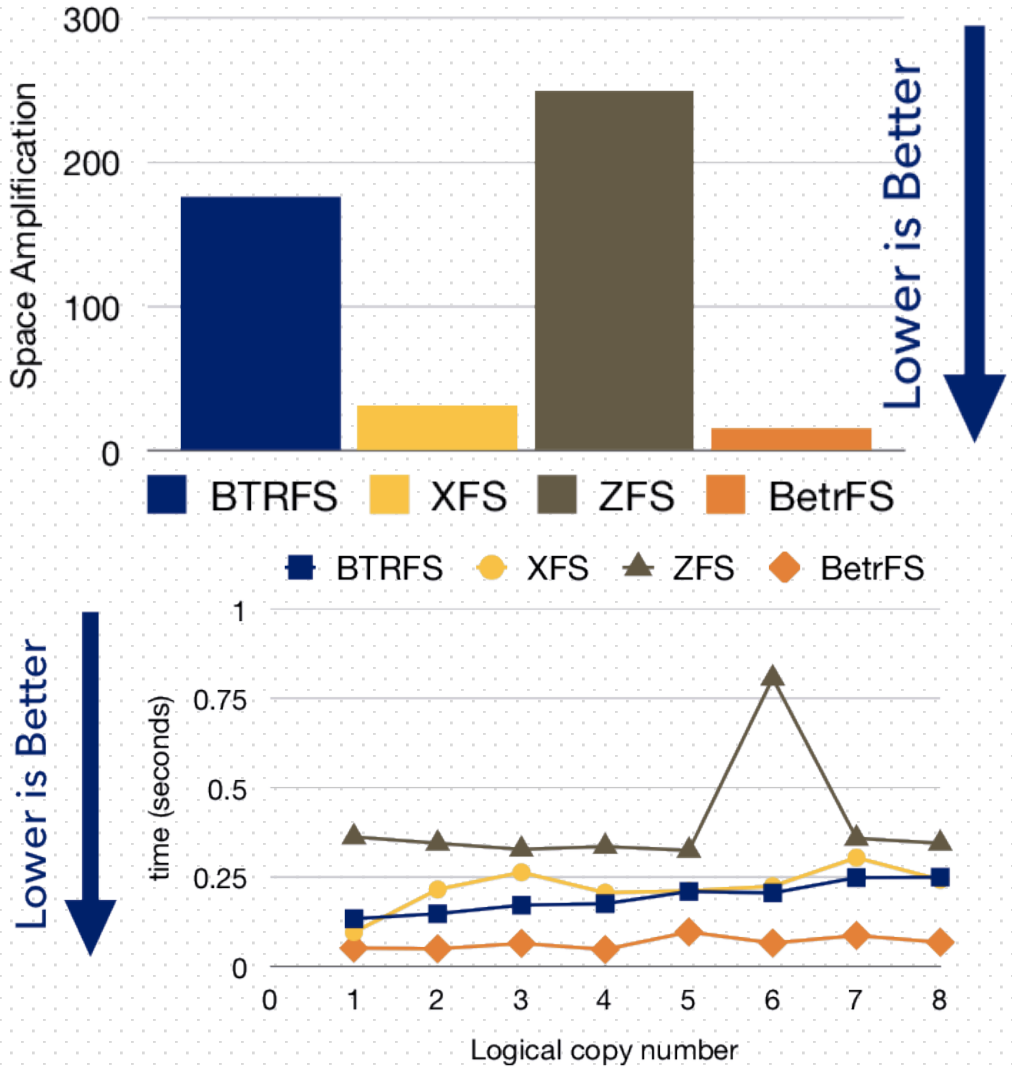
Fragmentation

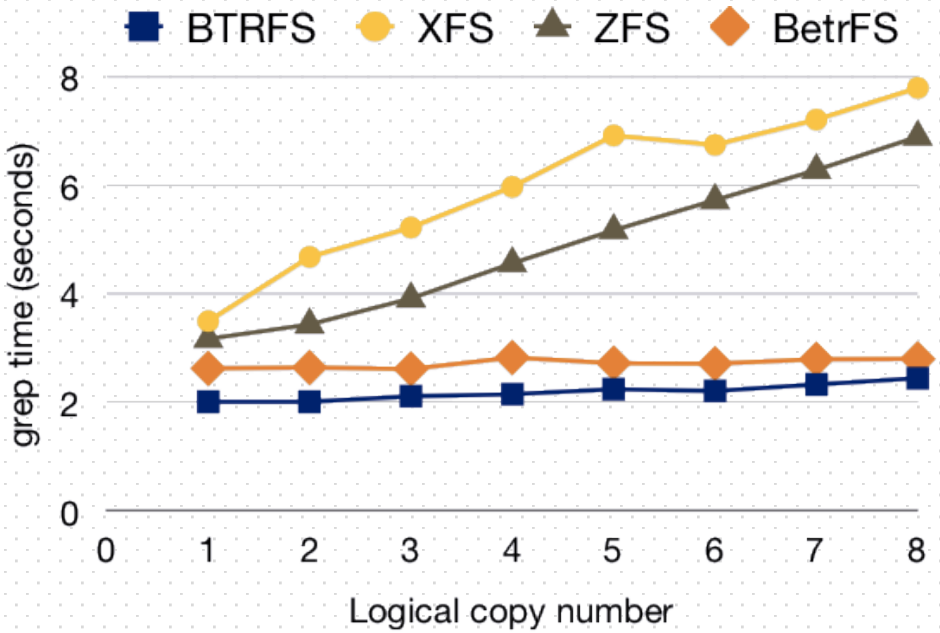


Clone Latency



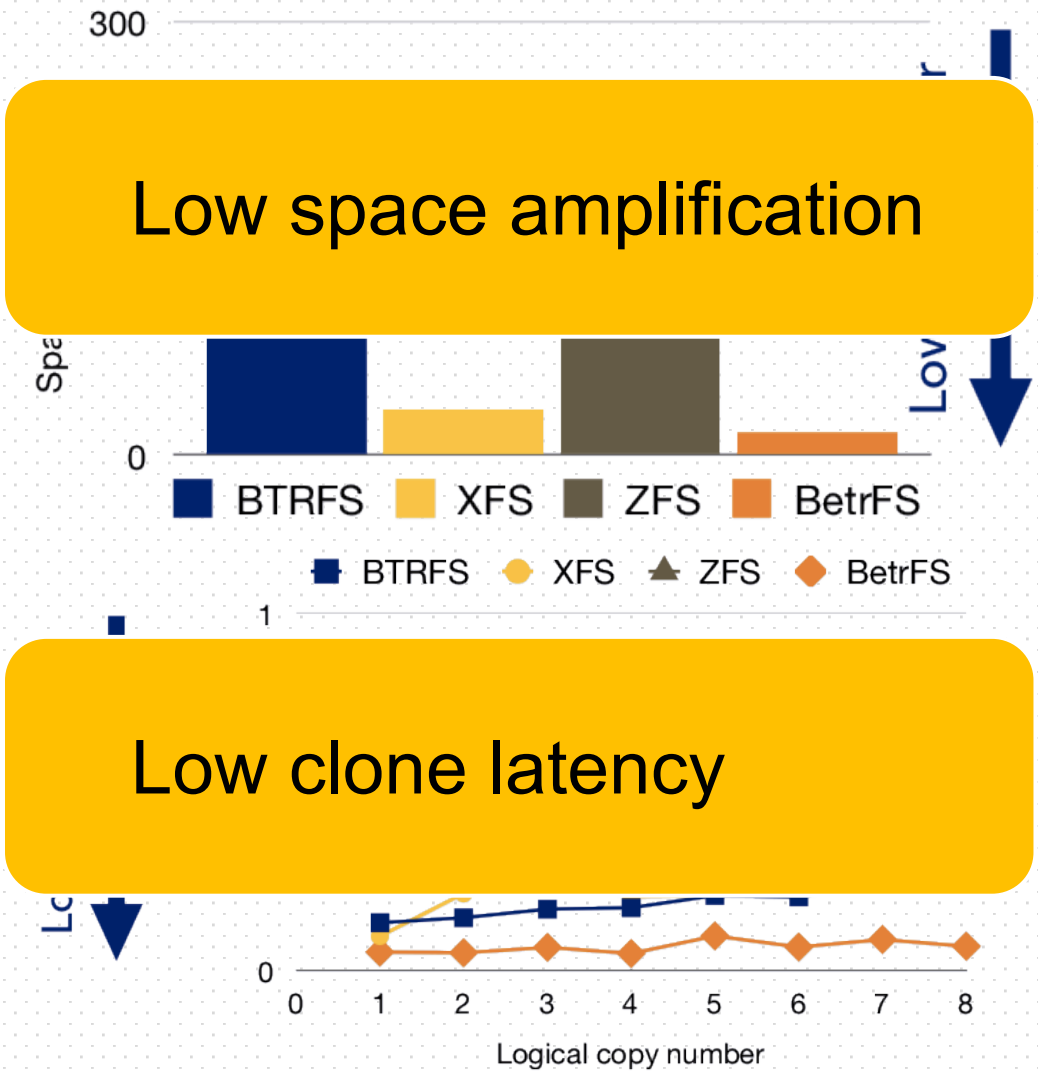
Clone Latency



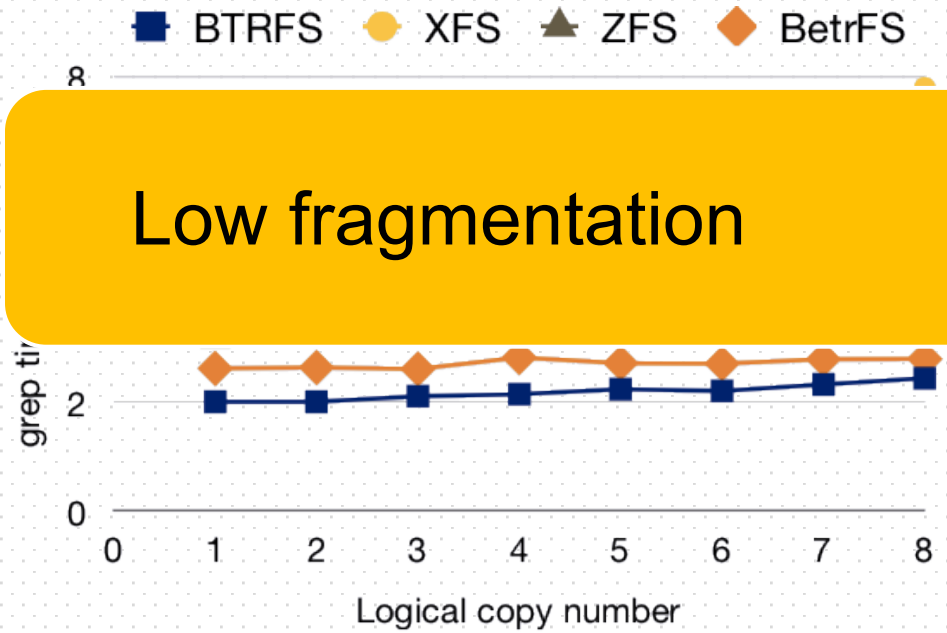


Clone Latency

Low space amplification



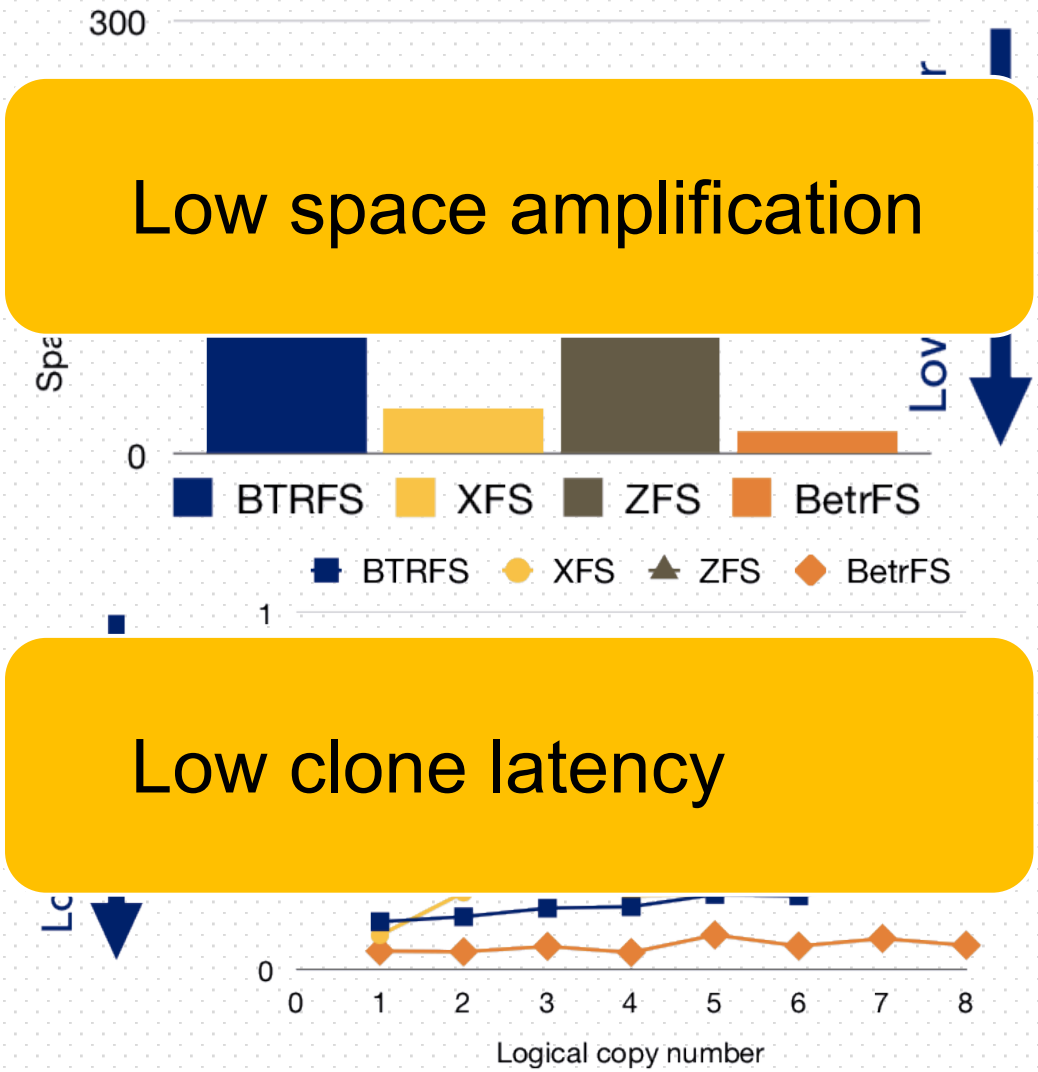
Low fragmentation



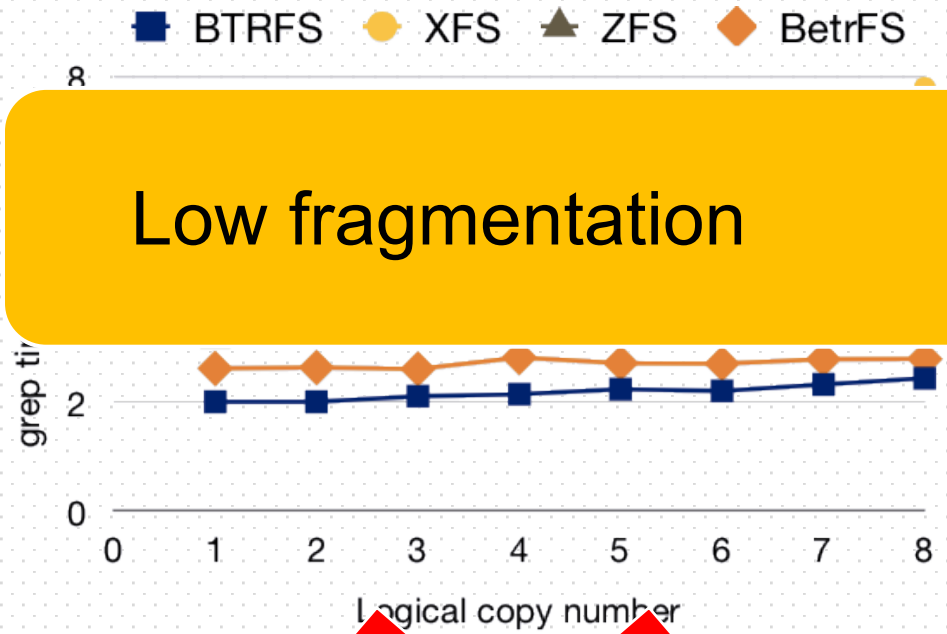
Low clone latency

Clone Latency

Low space amplification



Low fragmentation

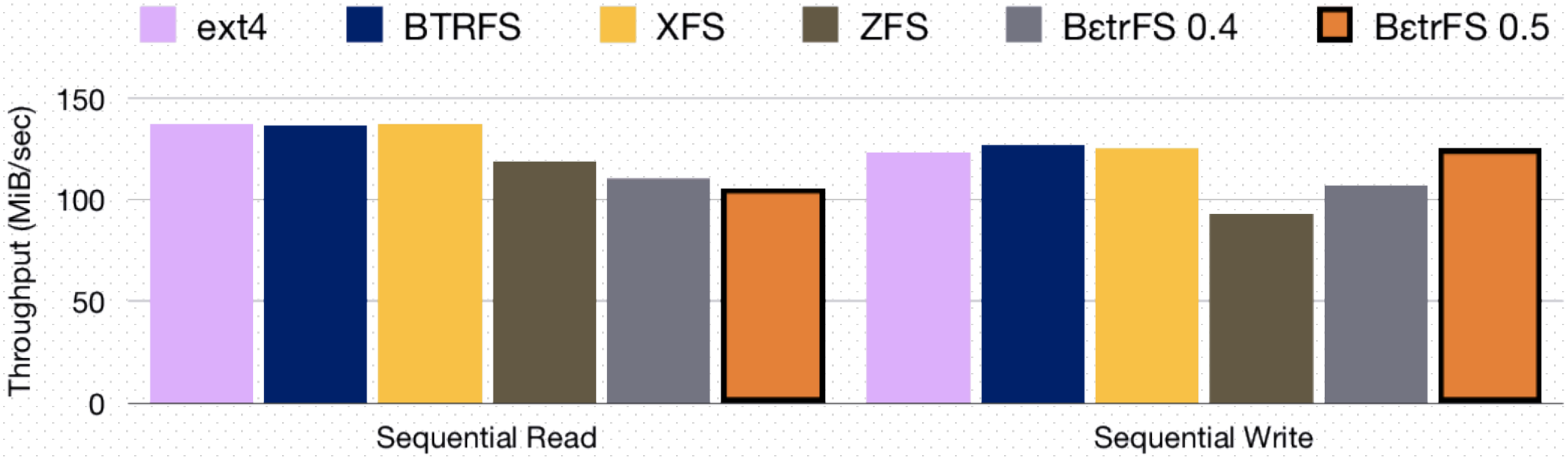


Low clone latency

Trade-off

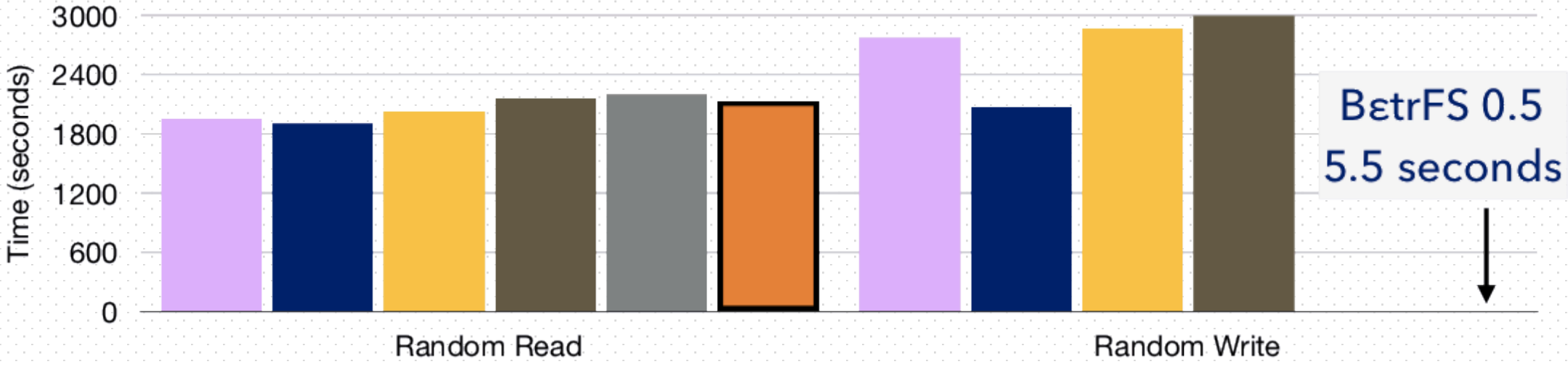
General Filesystem Performance

- Sequential IO:
  - Write 10GB file (4GB RAM)
  - Then read

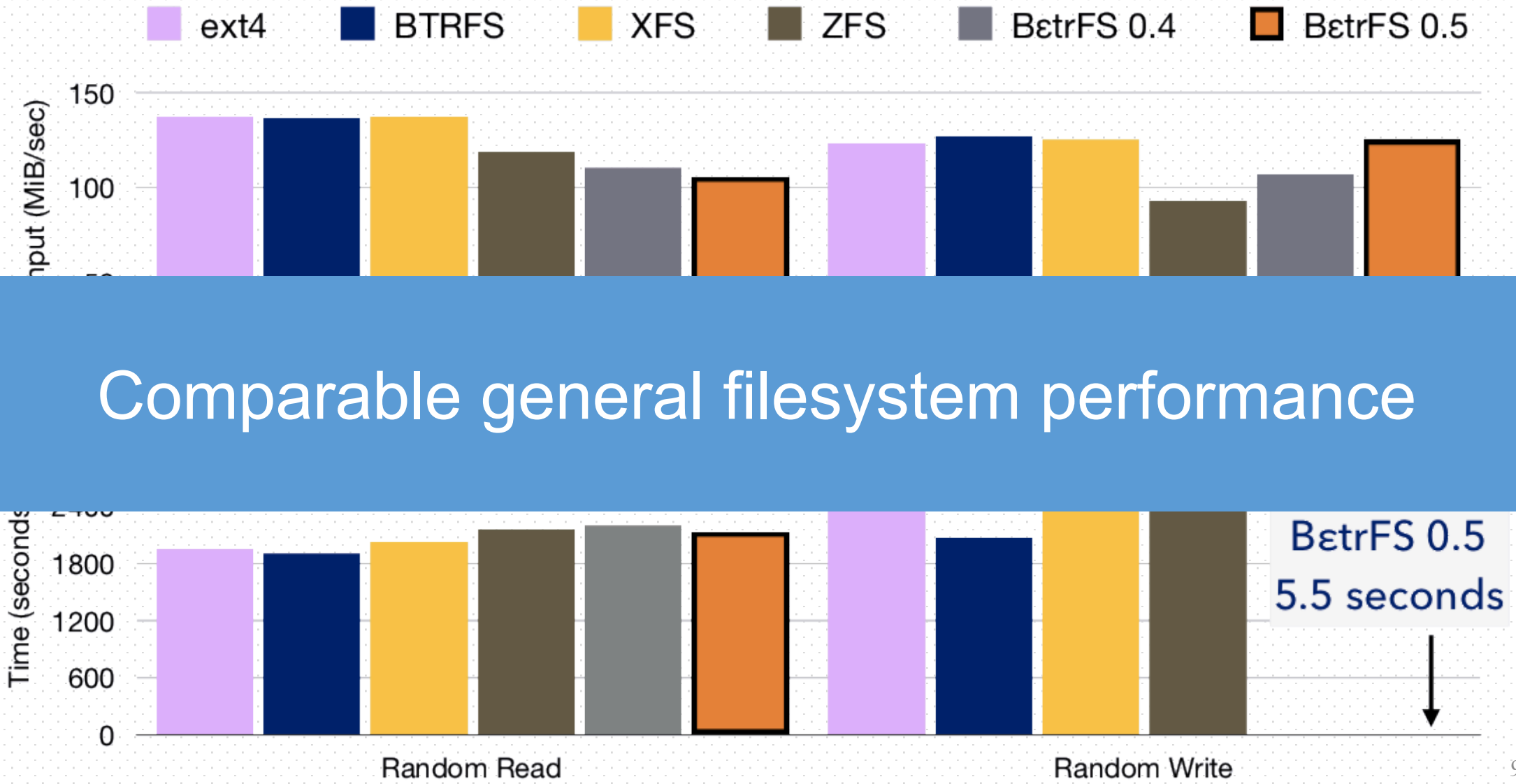


General Filesystem Performance

- Random IO:
  - Issues 256K 4-byte overwrites at random offsets within a 10GiB file, followed by an fsync
  - Issuing 256K 4-byte reads at random offsets within an existing 10GiB file

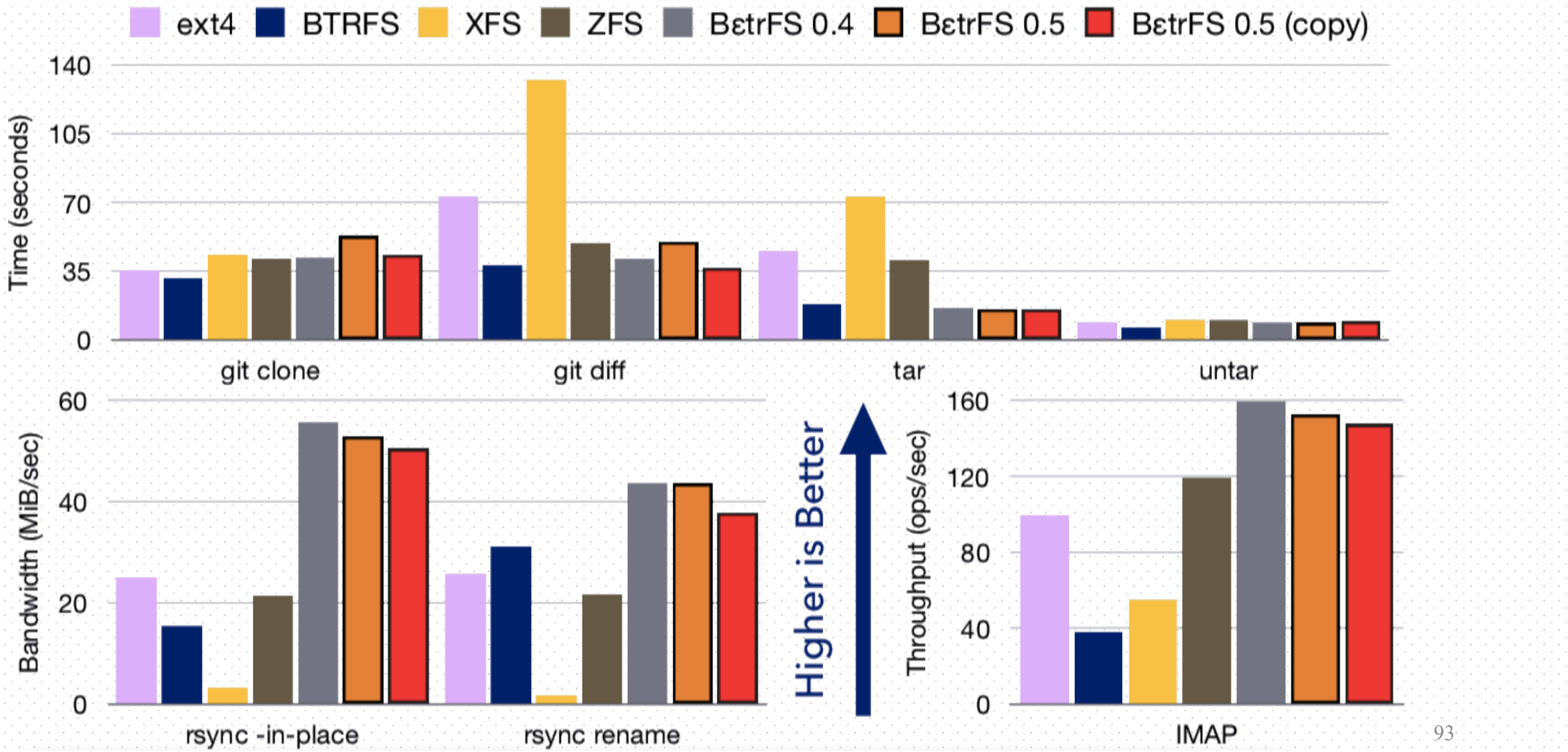


General Filesystem Performance



Comparable general filesystem performance

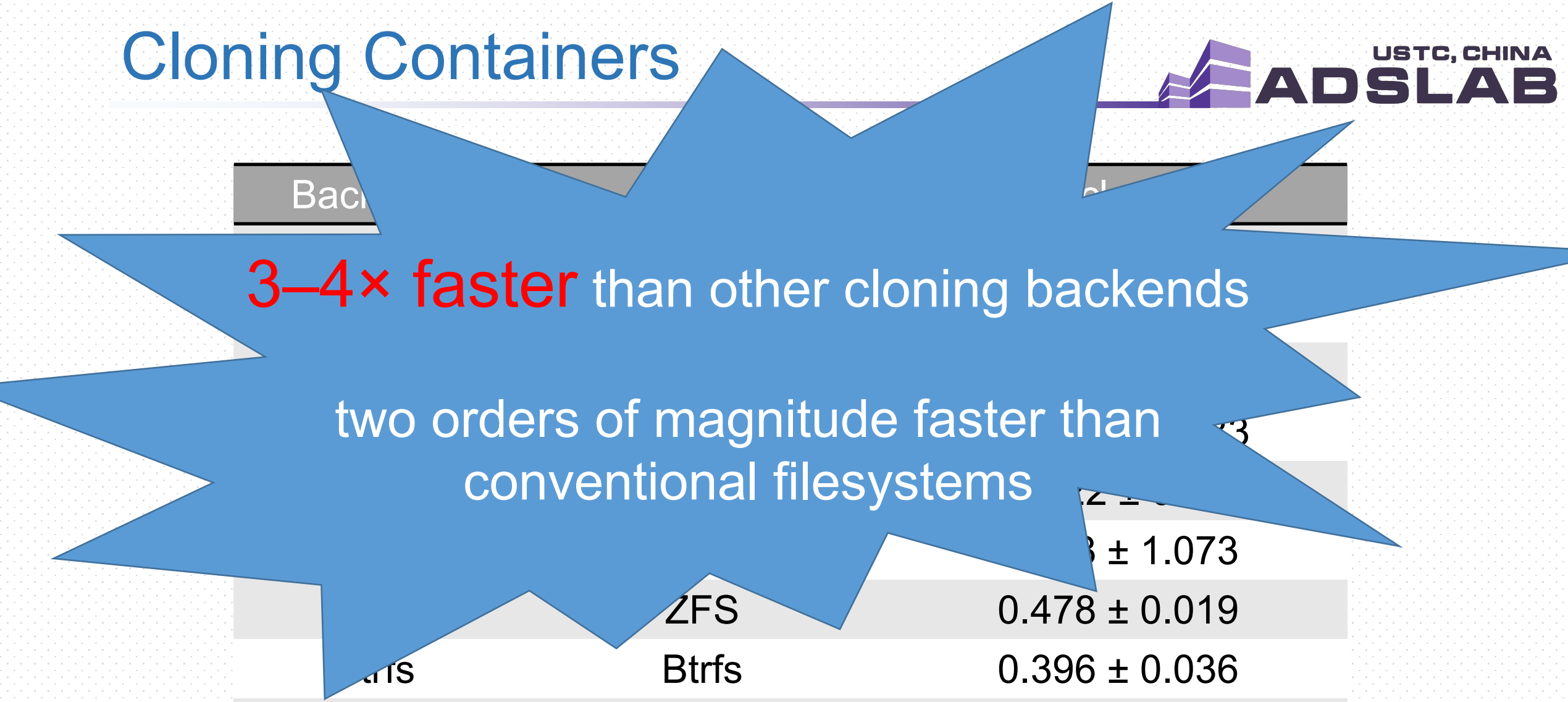
Application Benchmarks



Cloning Containers

Back-end	FS	LXC-clone (s)
Dir	ext4	19.514 ± 1.214
	Btrfs	14.822 ± 0.076
	ZFS	16.194 ± 0.538
	XFS	55.104 ± 1.033
	NILFS2	26.622 ± 0.396
	BetrFS 0.5	8.818 ± 1.073
ZFS	ZFS	0.478 ± 0.019
Btrfs	Btrfs	0.396 ± 0.036
Betrfs 0.5	BetrFS 0.5-clone	0.118 ± 0.010

Cloning Containers



Bac		
		3
		2 ±
		3 ± 1.073
	ZFS	0.478 ± 0.019
rfs	Btrfs	0.396 ± 0.036
Btrfs 0.5	BetrFS 0.5-clone	0.118 ± 0.010

Conclusion

- Use write-optimization to decouple writes from copies
- Implementation of nimble clone
  - efficient clones, efficient reads, efficient writes, and space efficiency.
- Does not harm performance of unrelated operations, and can unlock improvements for real applications.
- 3–4× improvement in LXC container cloning time compared to optimized back-ends.

Thanks!

Shared by Yiduo Wang, Daniel Shao

2020/05/08