InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems

Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, Jiwu Shu

Tsinghua University,Xiamen University,Alibaba Group

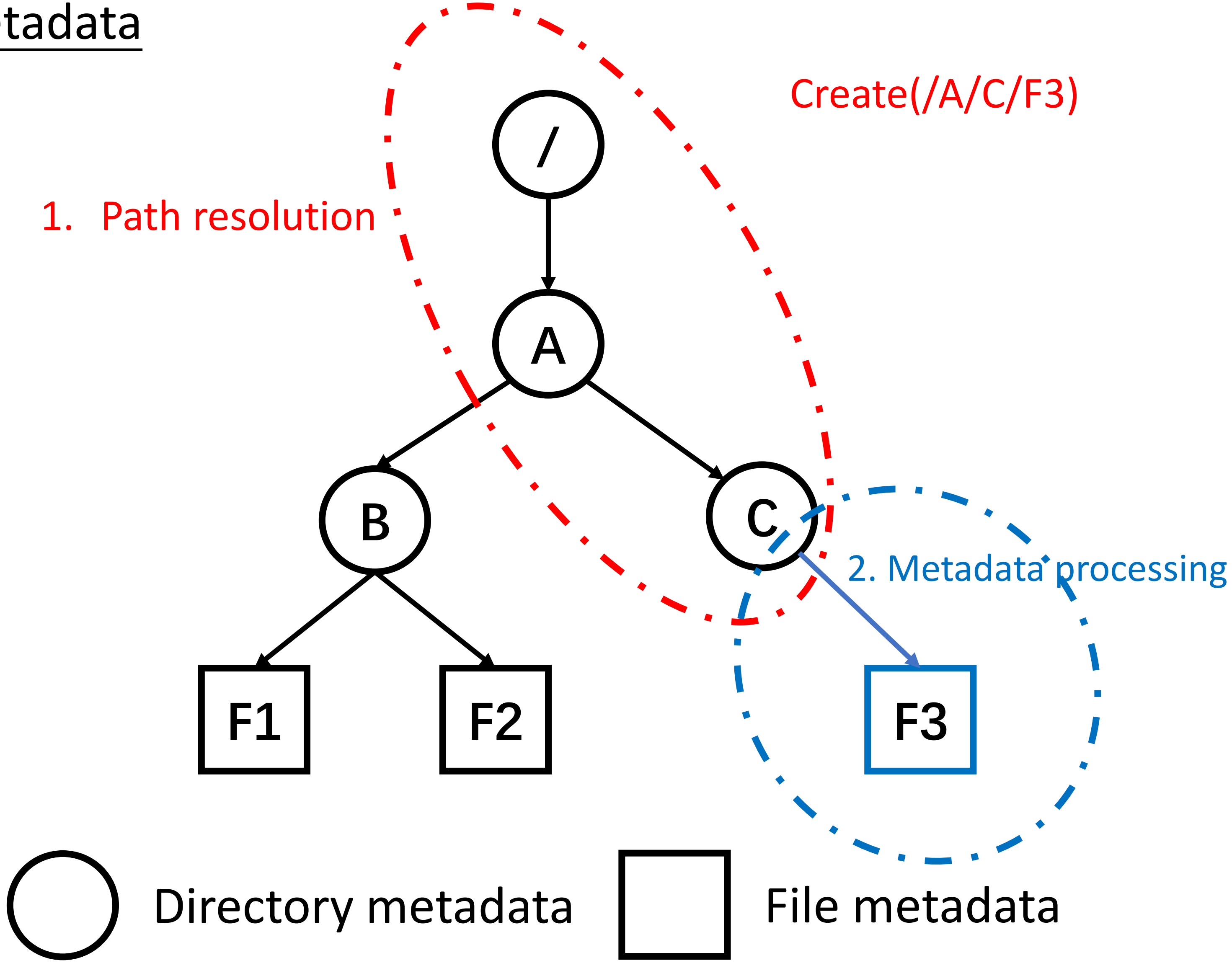
Speaker wrl

FAST 2022

Background Filesystem Metadata

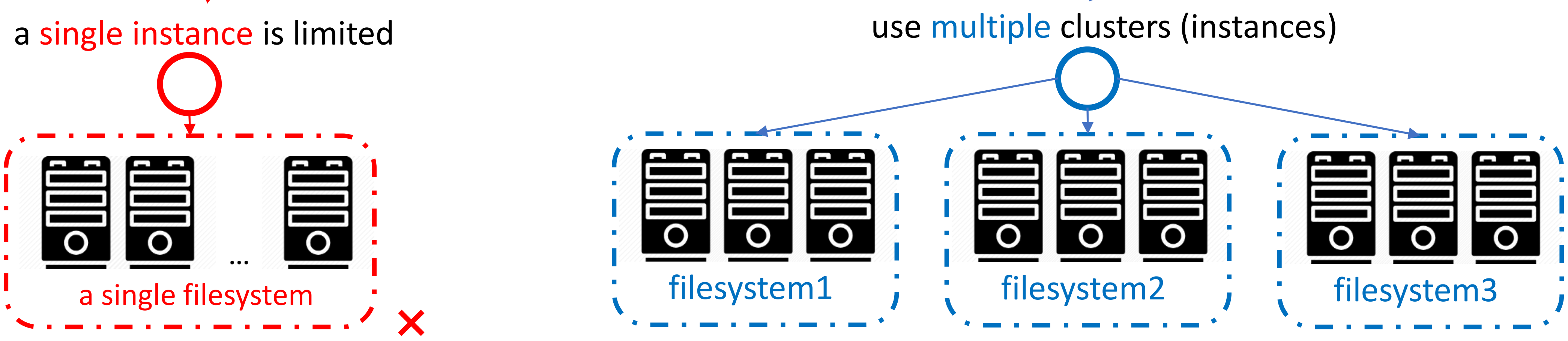
- Filesystem directory tree
  - Hierarchical namespace
  - Directory metadata
  - File metadata

- Metadata operation
  1. Path resolution
  2. Metadata processing



Background <u>Large-Scale Distributed Filesystem</u>

- **Modern datacenters contain a huge number of files**
  - Facebook: billions of files (Tectonic [FAST ’21])
  - Alibaba Cloud: tens of billions of files (thousands of Pangu)


a single instance is limited
...
a single filesystem
×
use multiple clusters (instances)
filesystem1
filesystem2
filesystem3

- **One single large-scale filesystem spanning the entire datacenter is desirable**
  - ✓ Global data sharing
  - ✓ High resource utilization
  - ✓ Low operational complexity

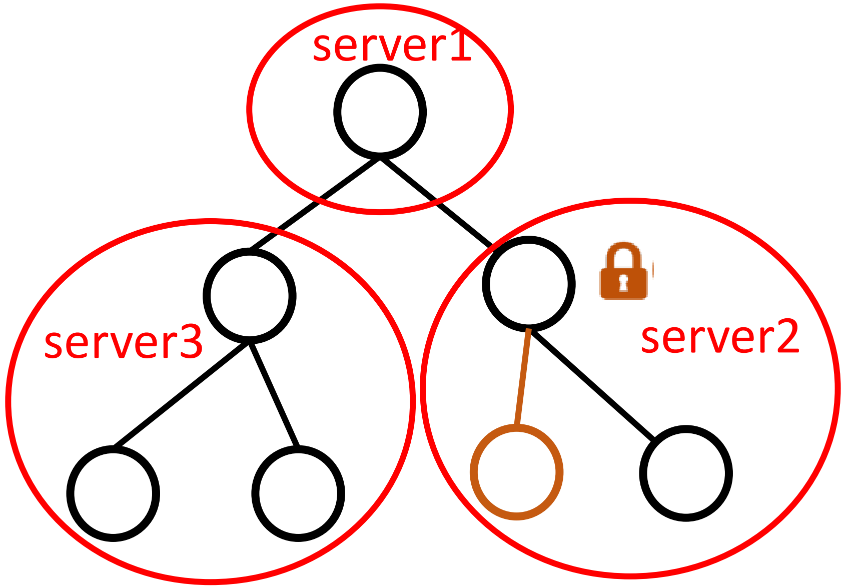
Problem

- Main idea： An Efficient Metadata Service for Large-Scale Distributed Filesystems

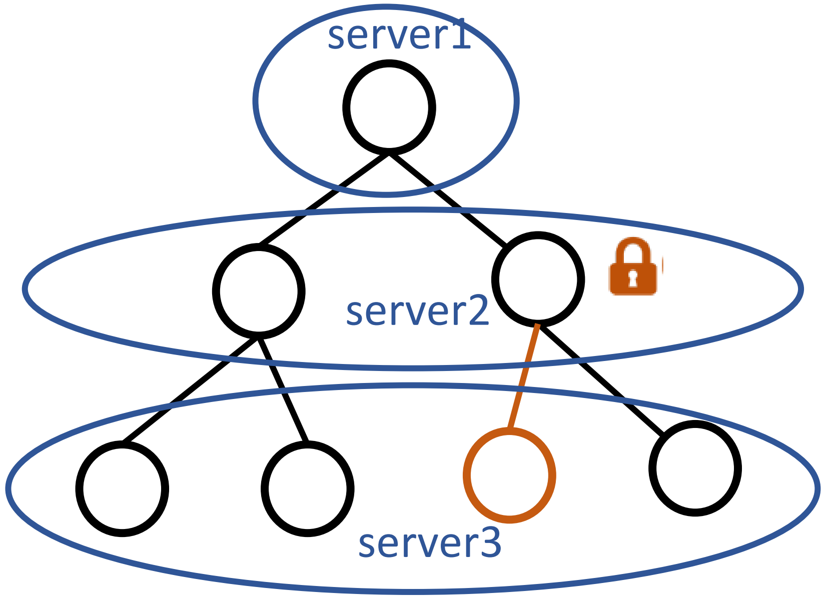
- Managing such huge number of files in one single filesystem brings severe challenges to the metadata service.

1. Partitioning of the directory tree
2. High latency of path resolution
3. High overhead of cache coherence maintenance

Challenge Partitioning of the directory tree



Coarse-grained Partitioning



Fine-grained Partitioning

Metadata Locality	☺	☹
Load Balancing	☹	☺

Design Access-Content Decoupled Partitionin

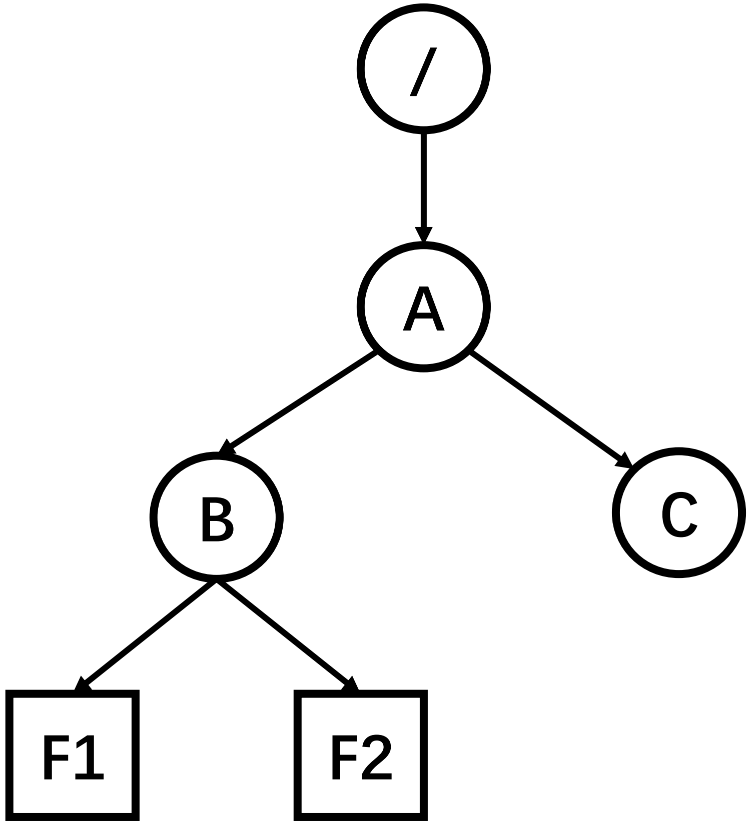
Root cause:

treat directory metadata as a whole

Key idea:

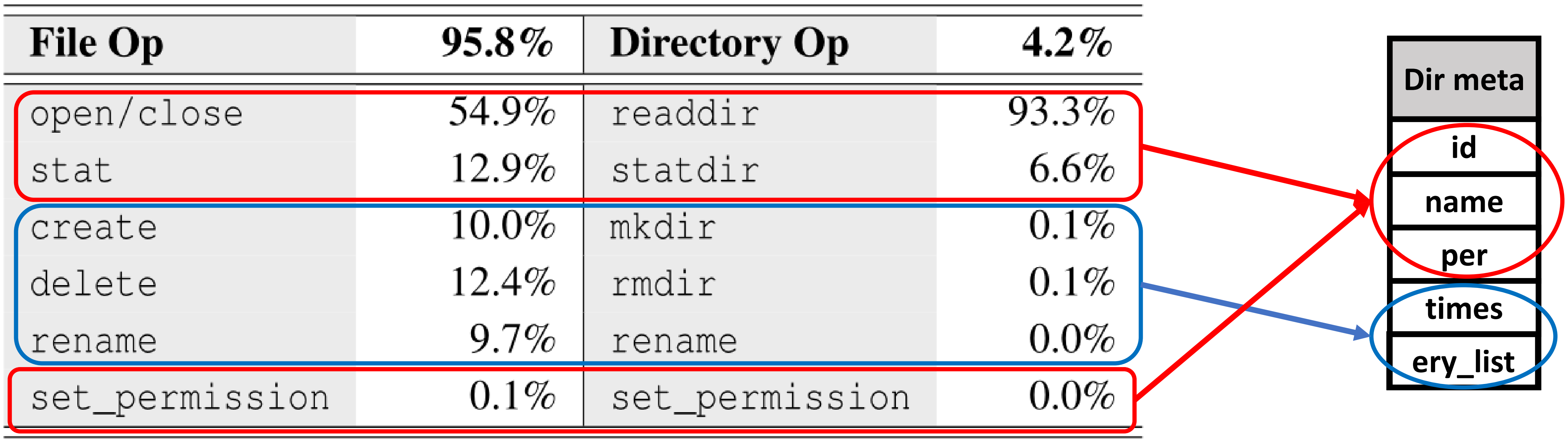
Decoupling directory metadata

Dir meta
id
name
per
times
ery_list



Insite Modern data center business workload characteristics

Three Pangu filesystem instances that support different services:
data processing and analyzing service, object storage service, and block storage service



File Op	95.8%	Directory Op	4.2%
open/close	54.9%	readdir	93.3%
stat	12.9%	statdir	6.6%
create	10.0%	mkdir	0.1%
delete	12.4%	rmdir	0.1%
rename	9.7%	rename	0.0%
set_permission	0.1%	set_permission	0.0%

Dir meta
id
name
per
times
ery_list

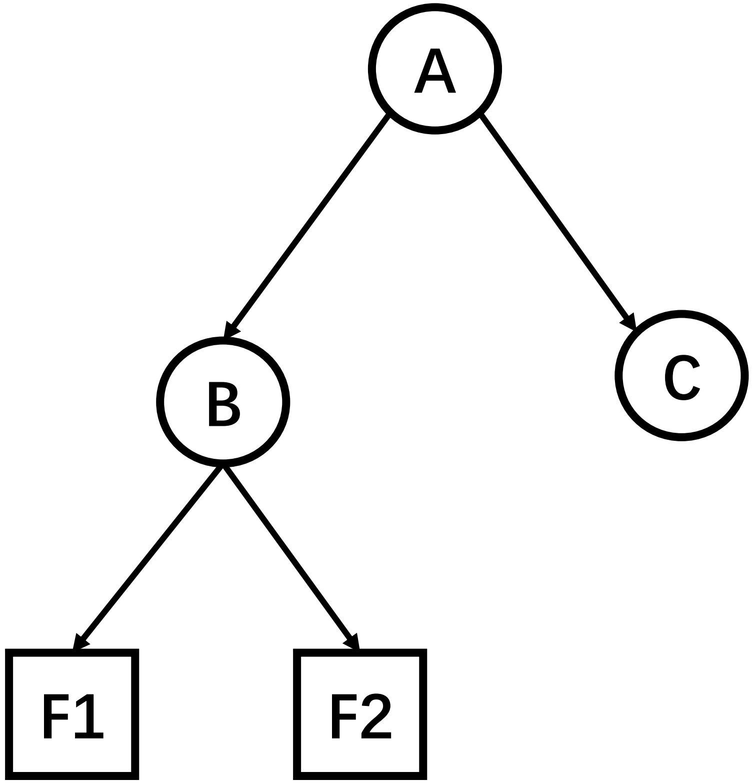
- **File operations** account for **95.8%** of all operations.
- The directory **readdir** is the most frequent directory operation, accounting for **93.3%** of all directory operations.
- **Directory rename** and **directory set_permission** operations rarely occur, accounting for only **0.0083%** of all metadata operations.

Design Access-Content Decoupled Partitionin

Access metadata

Content metadata

Dir meta
id
name
per
times
ery_list

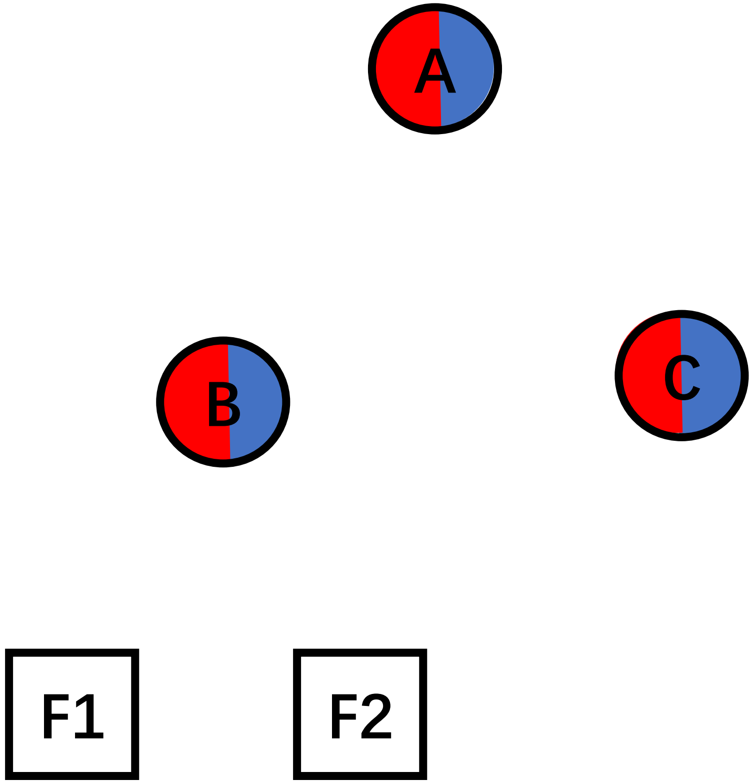


Design Access-Content Decoupled Partitionin

Access metadata

Content metadata

Dir meta
id
name
per
times
ery_list



Design Access-Content Decoupled Partitionin

Access metadata

Related to directory tree accessing

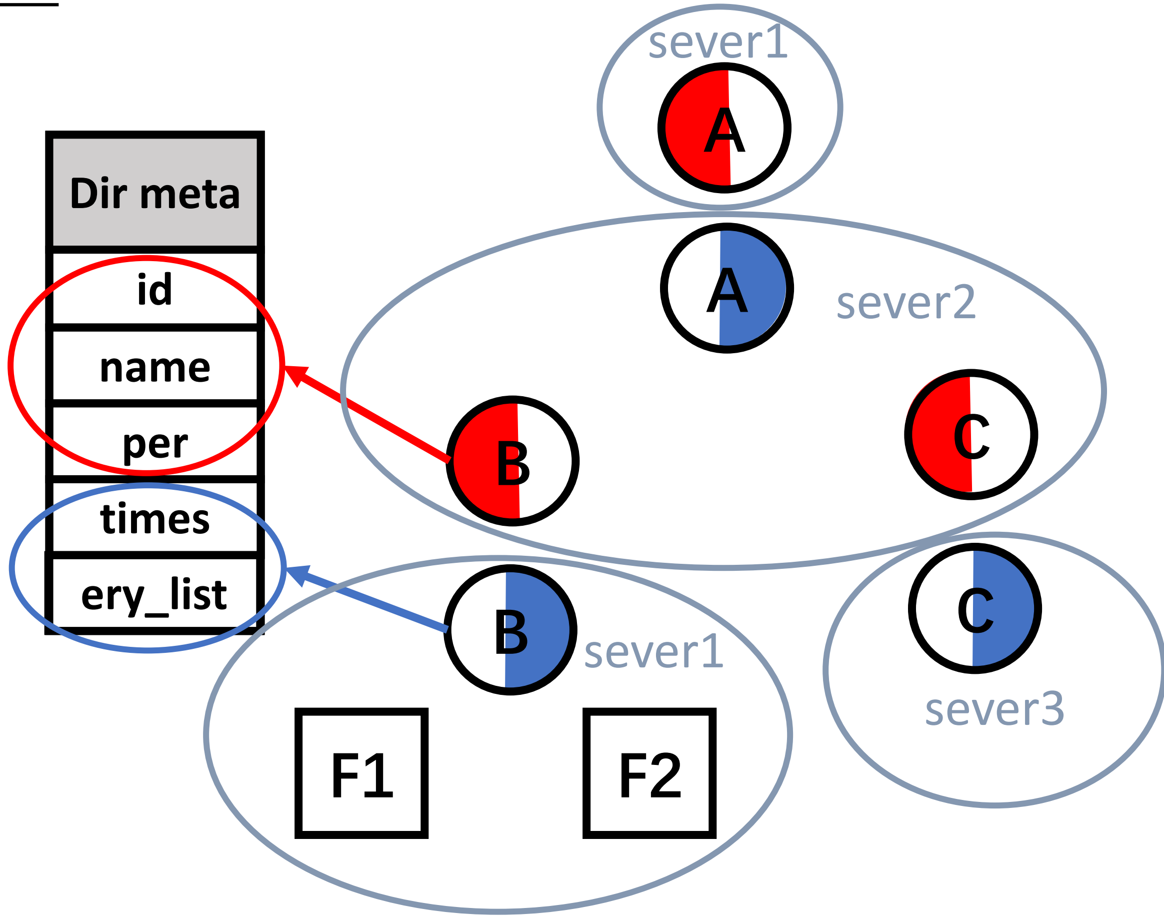
Content metadata

Related to the children

Grouping related metadata for locality

- **Access metadata with the parent**
- **Content metadata with the children**

Hash Partitioning for load balancing

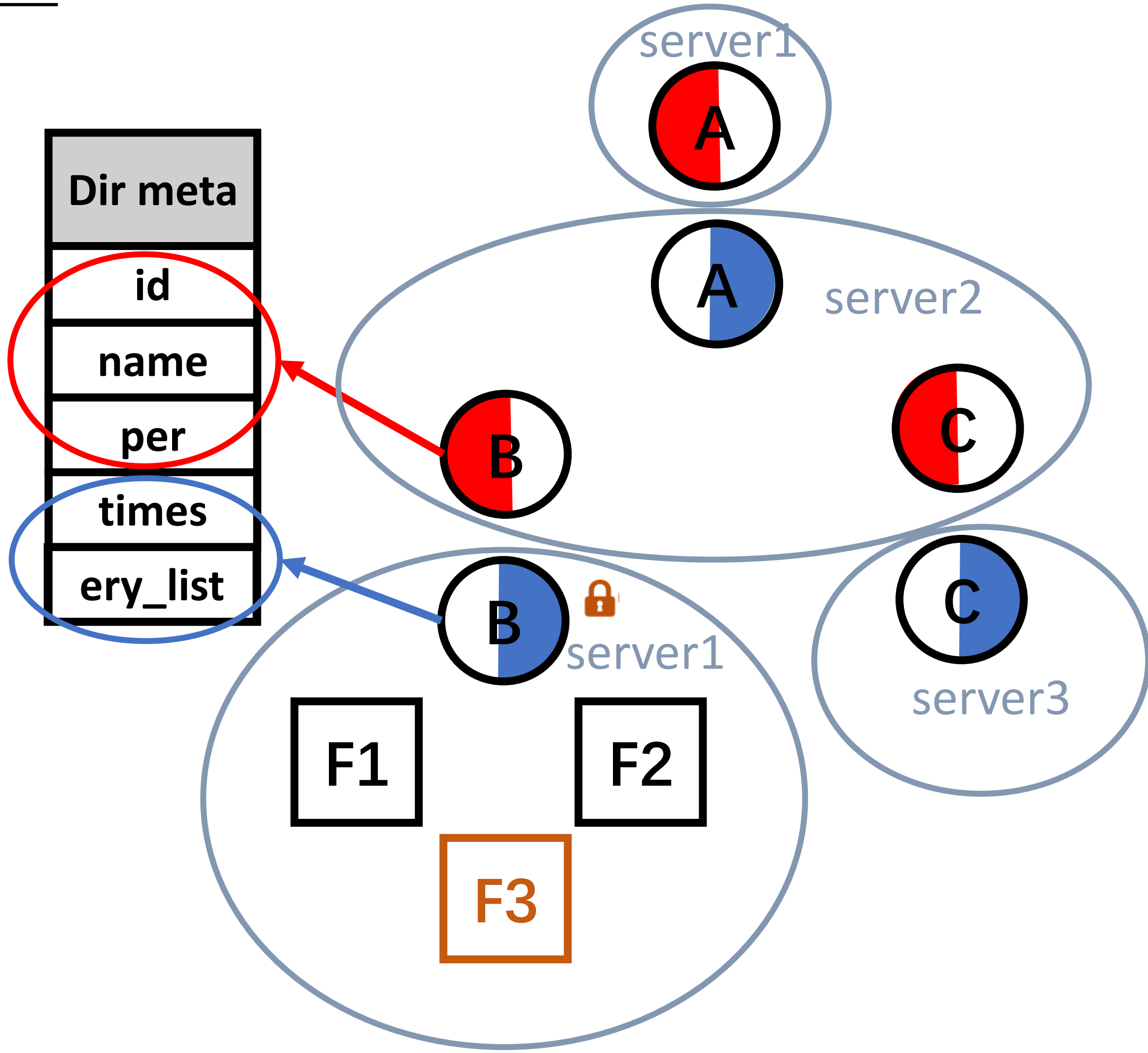


Design <u>Access-Content Decoupled Partitionin</u>

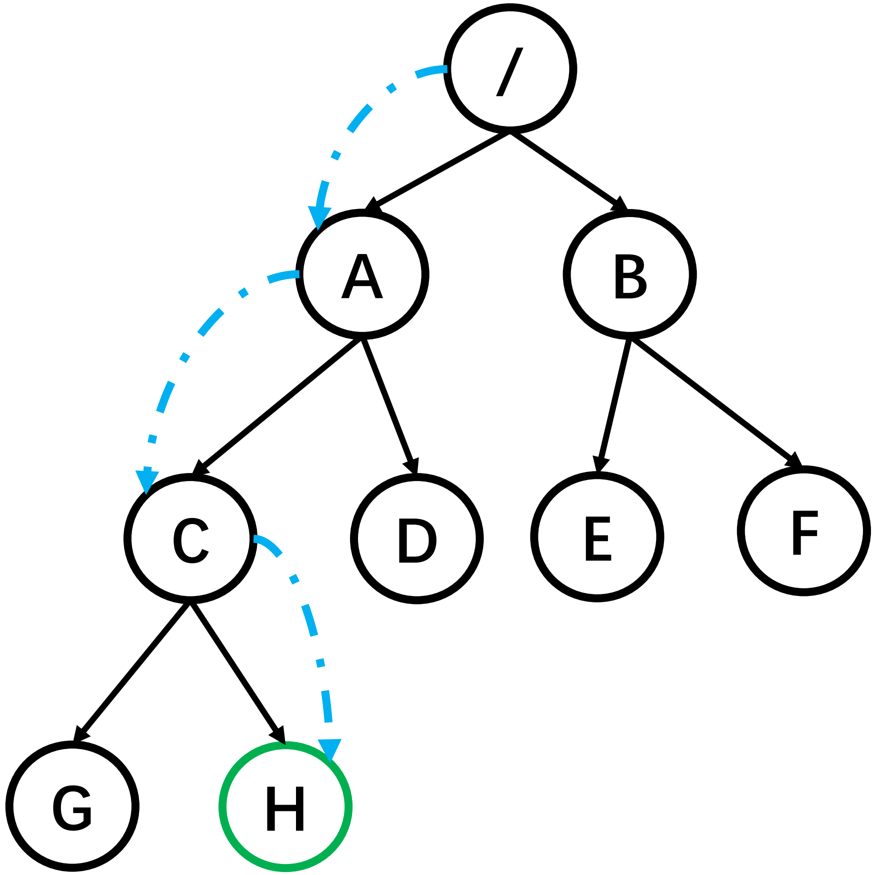
Good load balancing and high metadata locality for common operations like **file create, delete,** and **directory readdir**

E.g., **create(/A/B/f3)**
Step 1. lock the directory
Step 2. insert new file's metadata
Step 3. update directory's dirent and timestamps

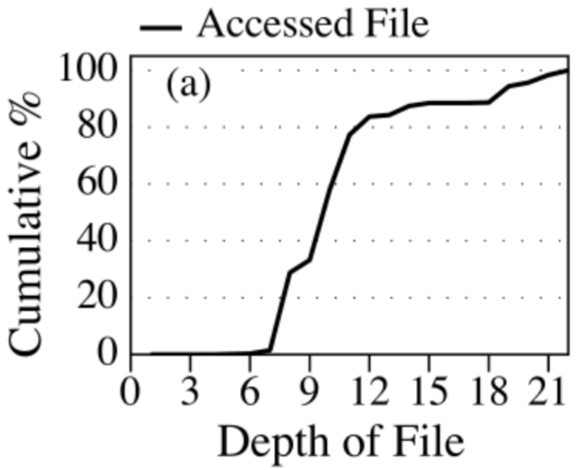
Only involve one single metadata serve!

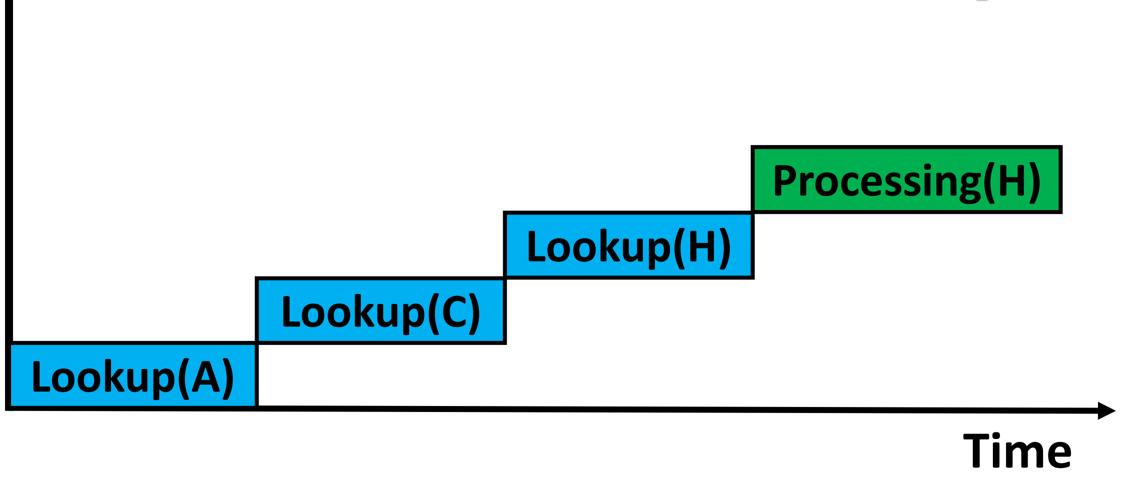


Challenge High latency of path resolution



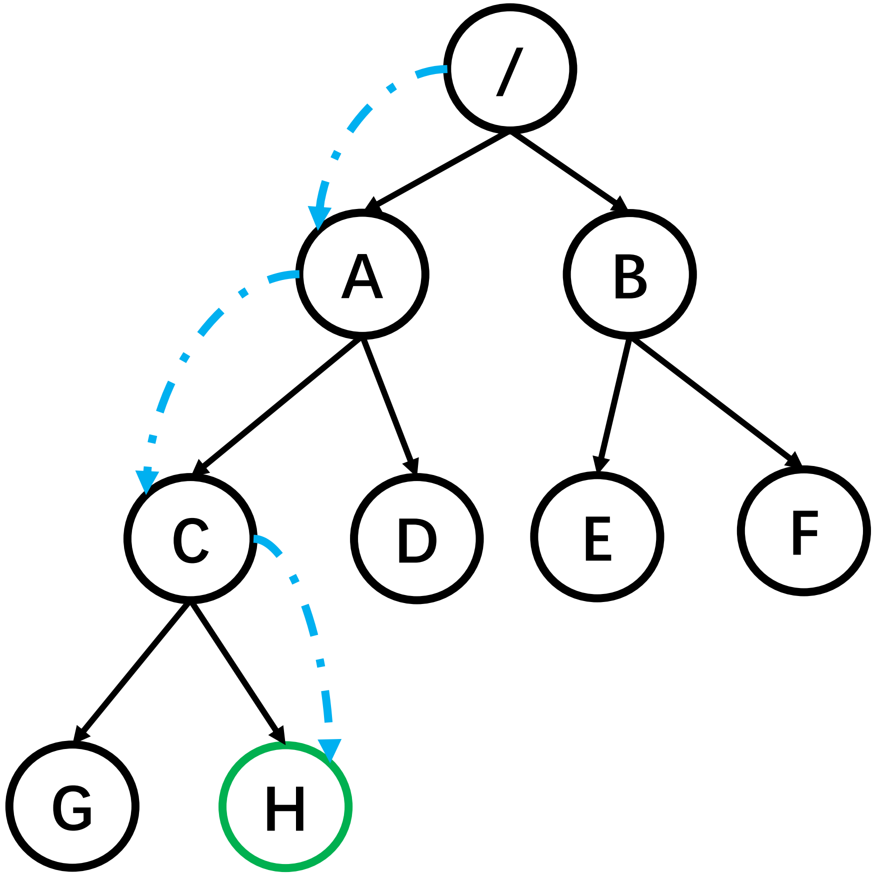
1. Path resolution
2. Metadata processing





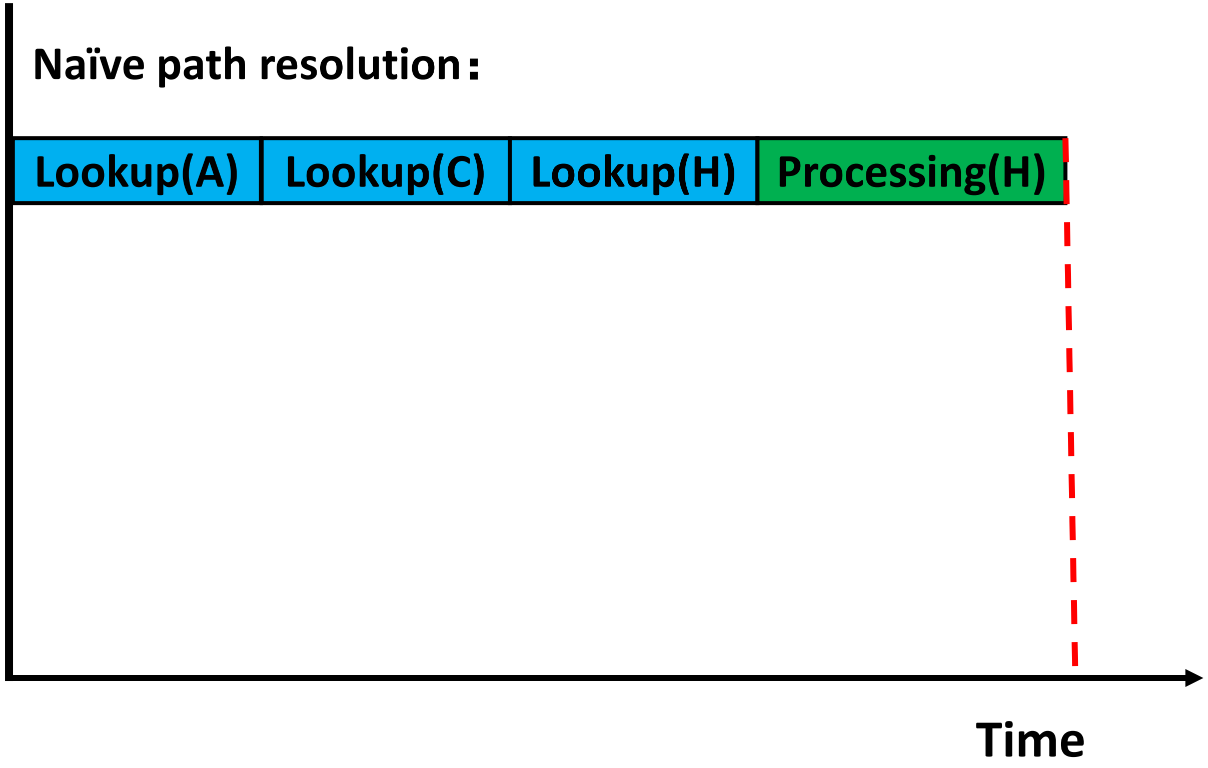
High file depth → High latency

Design Speculative Path Resolution

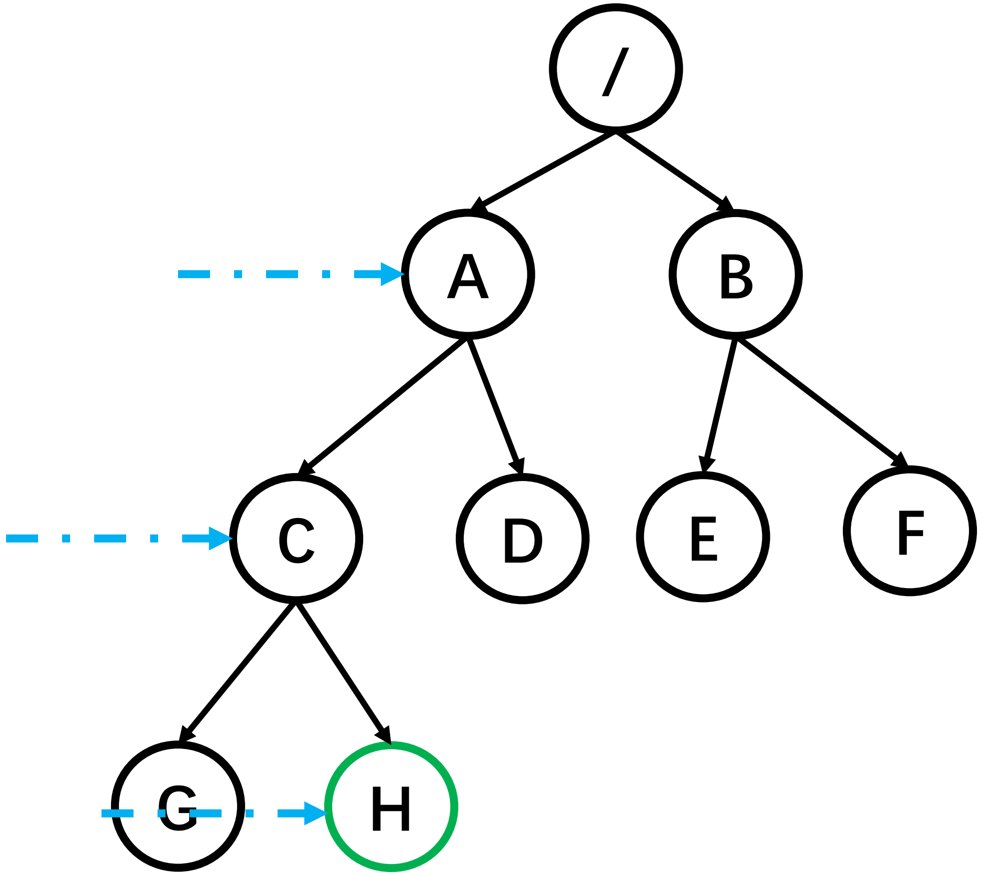


Key idea：

Predict directory IDs and parallelize lookup requests

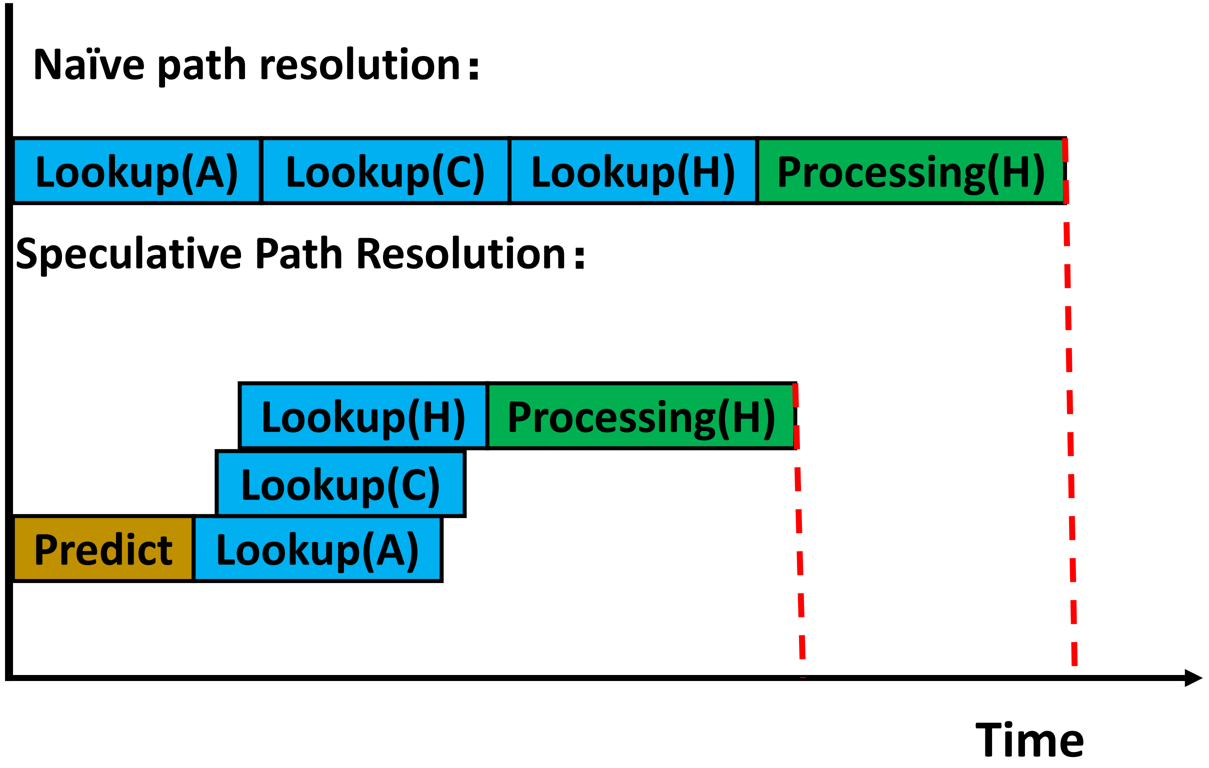


Design Speculative Path Resolution



Key idea：

Predict directory IDs and parallelize lookup requests


Naïve path resolution：

Lookup(A) Lookup(C) Lookup(H) Processing(H)

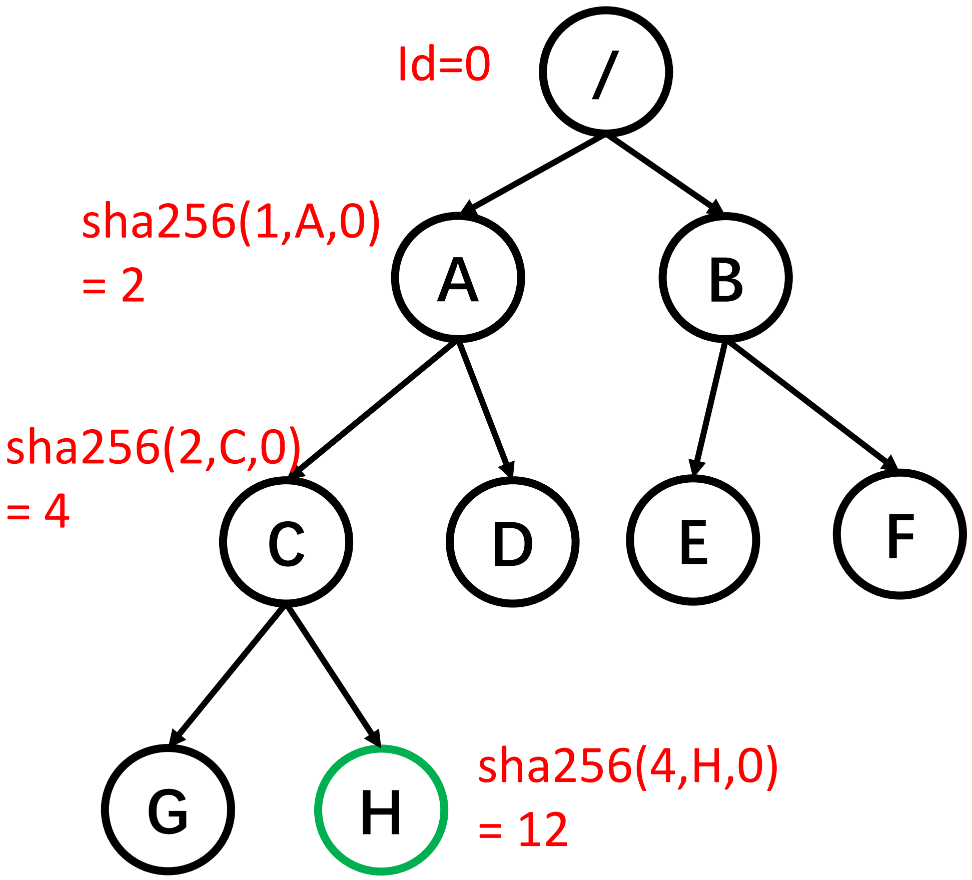
Speculative Path Resolution：

Lookup(H) Processing(H)

Lookup(C)

Predict Lookup(A)

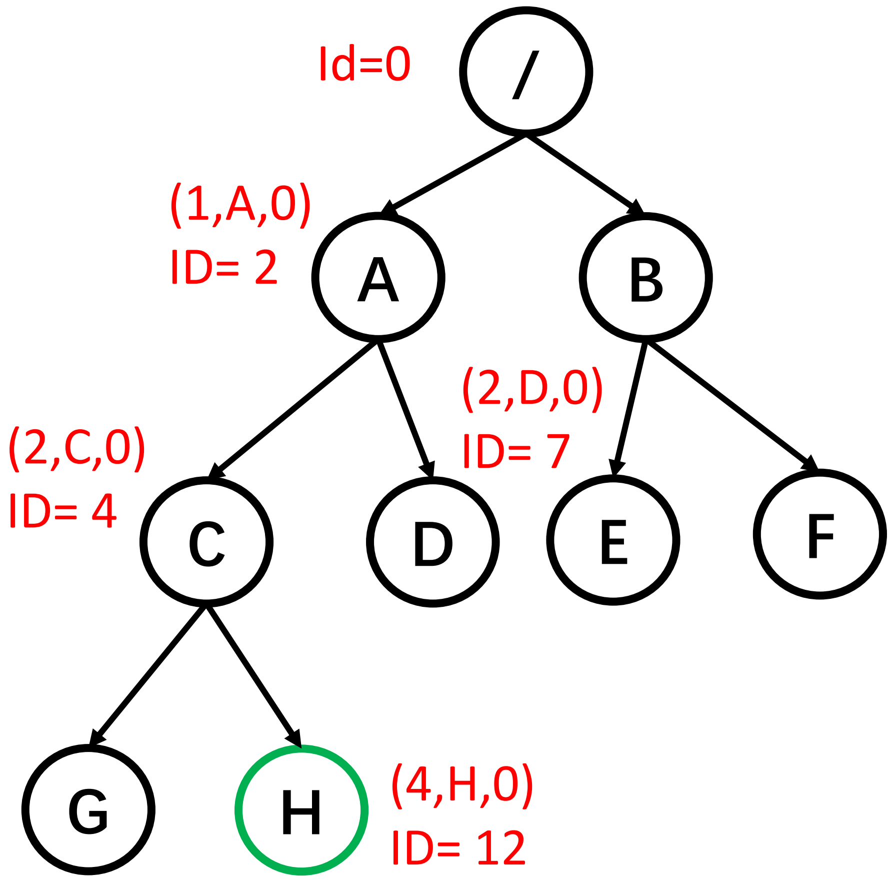
Design Speculative Path Resolution



Predictable Directory ID

- SHA256(parent ID, name, version)
- Version is 0 by default, unless the ID collision is detected

Design Speculative Path Resolution



Predictable Directory ID

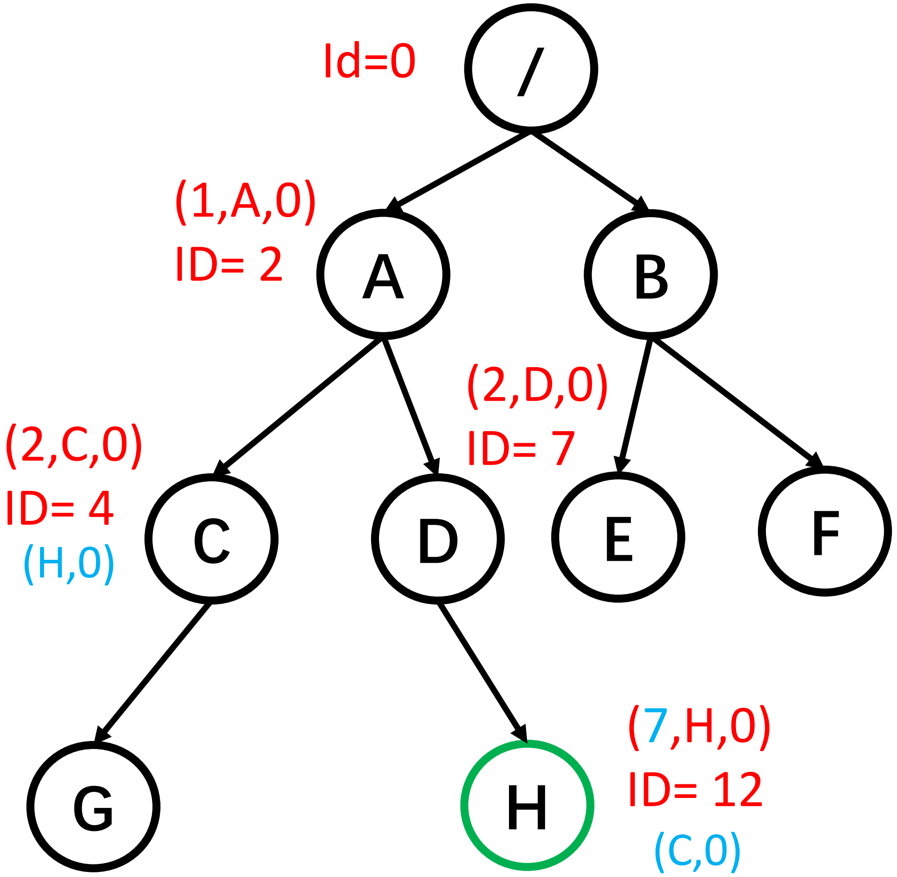
- SHA256(parent ID, name, version)
- Version is 0 by default, unless the ID collision is detected

When directory rename

- Old parent dir create a list to record child version
- Directory create a map to record old parent ID

Rename /A/C/H /A/D/H

Design Speculative Path Resolution



Predictable Directory ID

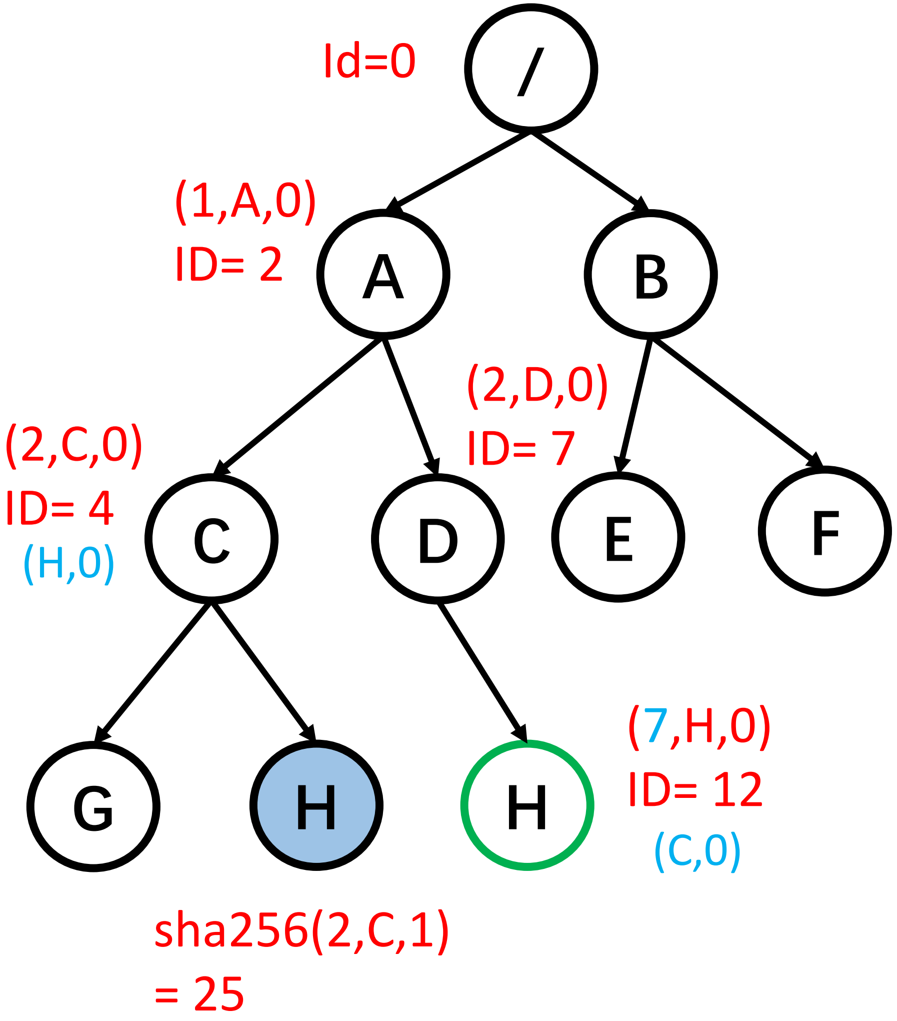
- SHA256(parent ID, name, version)
- Version is 0 by default, unless the ID collision is detected

When directory rename

- Old parent dir create a list to record child version
- Directory create a map to record old parent ID and version

Create /A/C/H

Design Speculative Path Resolution



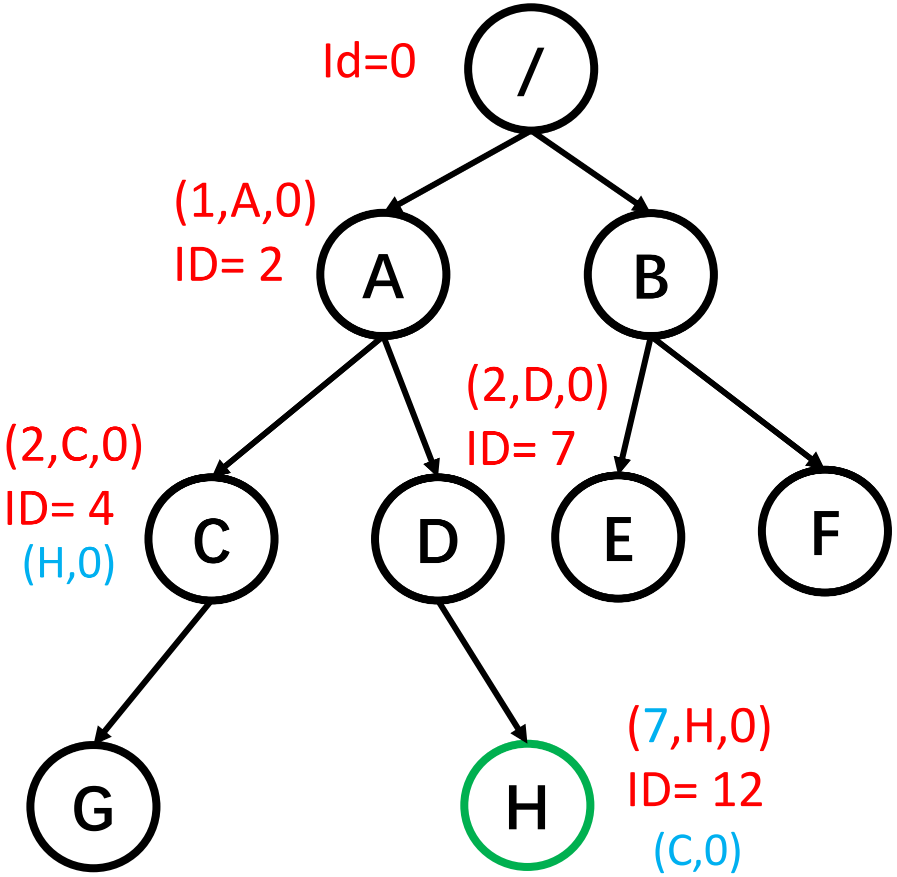
Predictable Directory ID

- SHA256(parent ID, name, version)
- Version is 0 by default, unless the ID collision is detected

When directory rename

- Old parent dir create a list to record child version
- Directory create a map to record old parent ID and version

Design Speculative Path Resolution



Predictable Directory ID

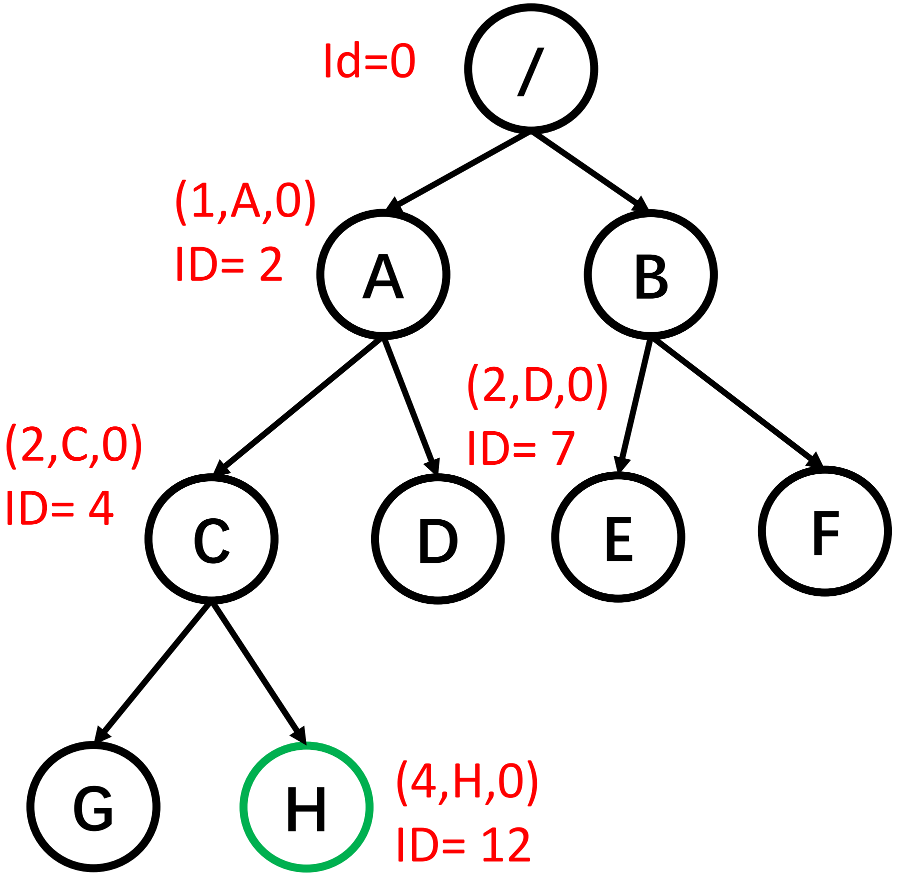
- SHA256(parent ID, name, version)
- Version is 0 by default, unless the ID collision is detected

When directory rename

- Old parent dir create a list to record child version
- Directory create a map to record old parent ID and version

Delete /A/D/H

Design Speculative Path Resolution



Predictable Directory ID

- SHA256(parent ID, name, version)
- Version is 0 by default, unless the ID collision is detected

When directory rename

- Old parent dir create a list to record child version
- Directory create a map to record old parent ID and version

Create /A/C/H

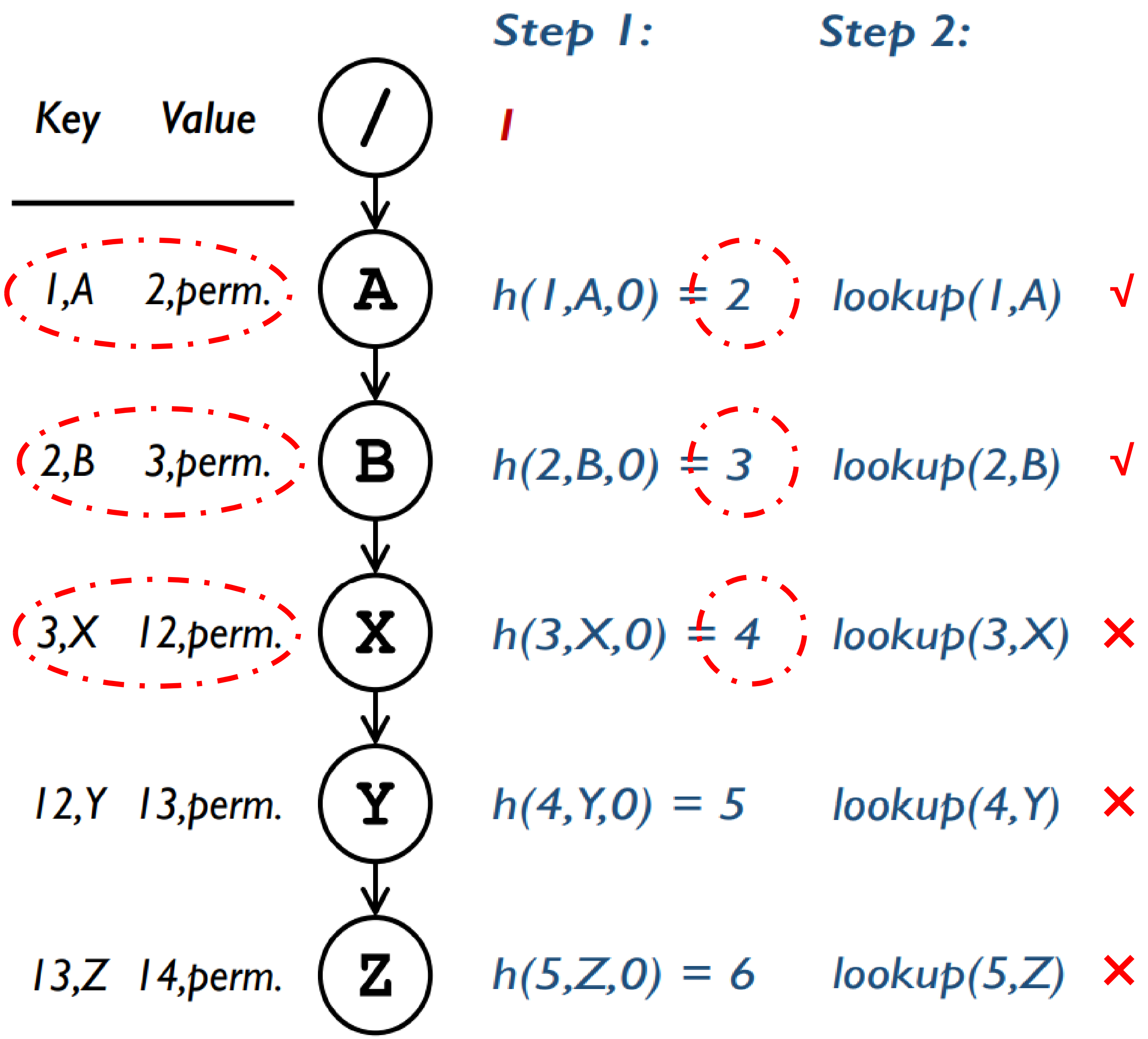
Design Speculative Path Resolution

Parallel Path Resolution

Step 1. predict directory IDs

Step 2. send lookups in parallel

- check permissions
- verify predicted IDs



Key	Value		Step 1:	Step 2:	
		/	1		
1,A	2,perm.	A	h(1,A,0) = 2	lookup(1,A)	√
2,B	3,perm.	B	h(2,B,0) = 3	lookup(2,B)	√
3,X	12,perm.	X	h(3,X,0) = 4	lookup(3,X)	✕
12,Y	13,perm.	Y	h(4,Y,0) = 5	lookup(4,Y)	✕
13,Z	14,perm.	Z	h(5,Z,0) = 6	lookup(5,Z)	✕

Design <u>Speculative Path Resolution</u>

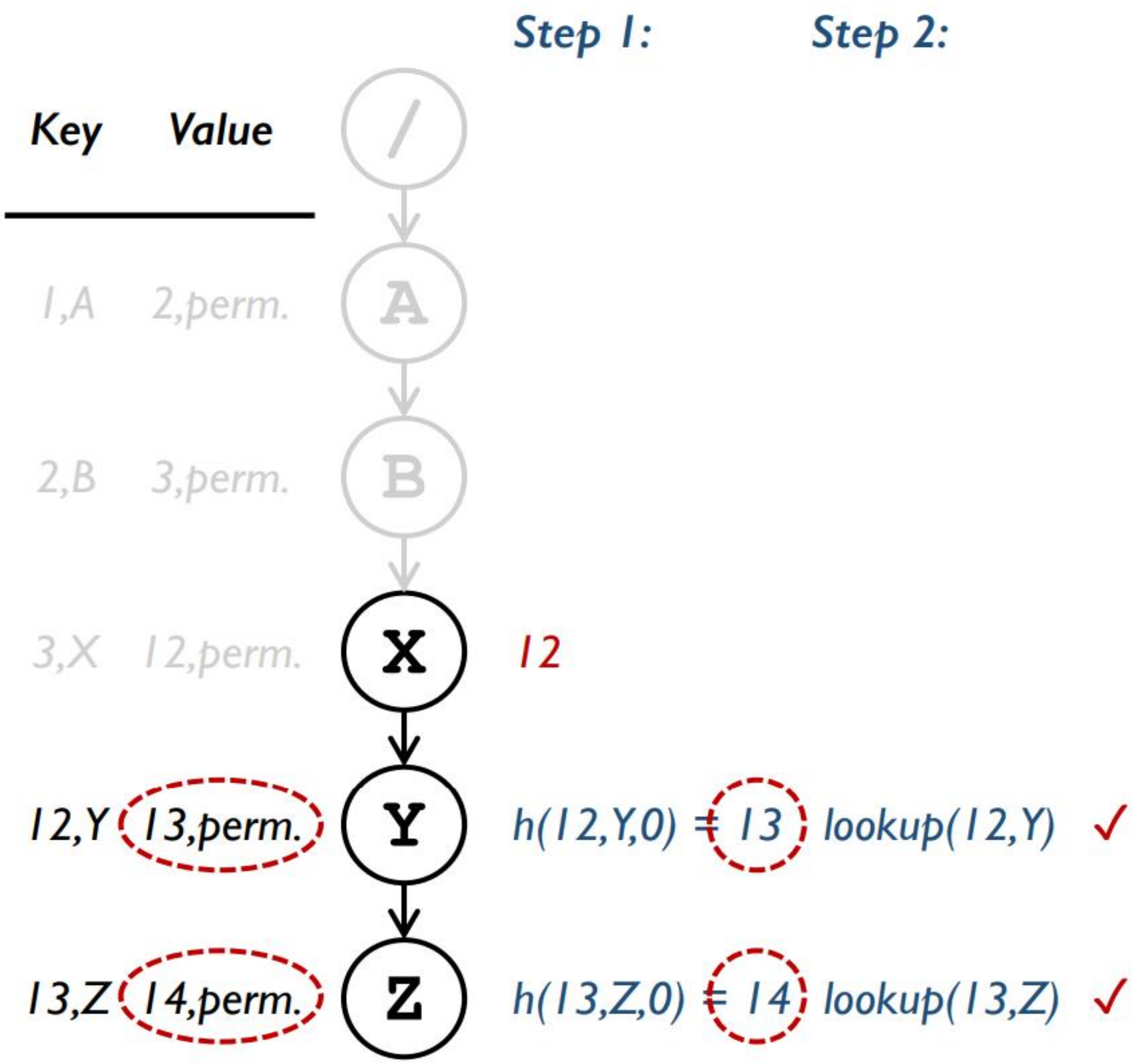
Parallel Path Resolution

Step 1. predict directory IDs

Step 2. send lookups in parallel

- check permissions
- verify predicted IDs

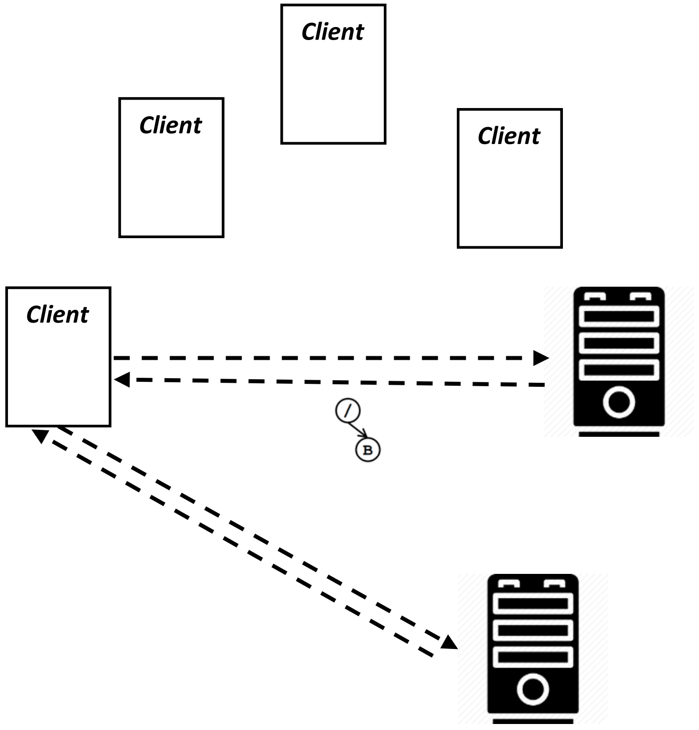
Step 3. repeat until finished

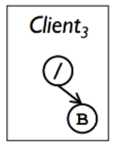


Key	Value		Step 1:	Step 2:
		/		
1,A	2,perm.	A		
2,B	3,perm.	B		
3,X	12,perm.	X	12	
12,Y	13,perm.	Y	h(12,Y,0) = 13	lookup(12,Y) ✓
13,Z	14,perm.	Z	h(13,Z,0) = 14	lookup(13,Z) ✓

Challenge High overhead of cache coherence maintenance

Near-root hotspots caused by the path resolution



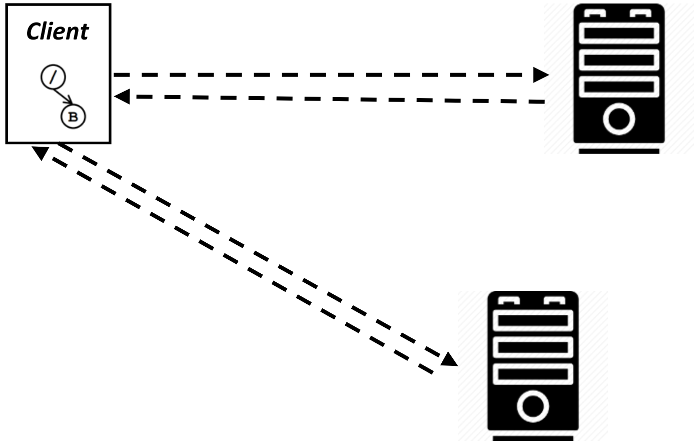


Cache metadata on the client-side

Challenge High overhead of cache coherence maintenance

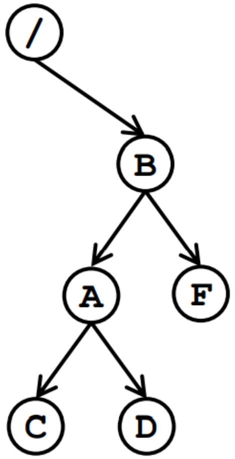
Near-root hotspots caused by the path resolution

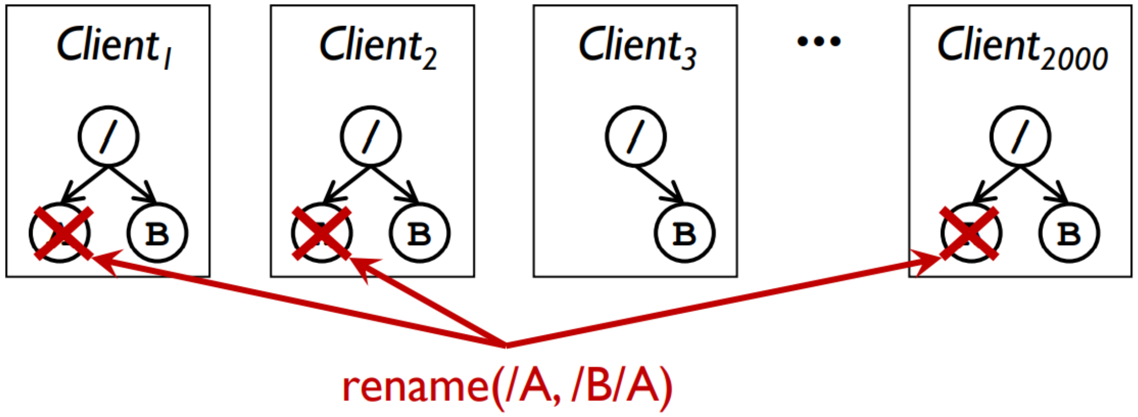
Cache metadata on the client-side



Challenge High overhead of cache coherence maintenance

Near-root hotspots caused by the path resolution



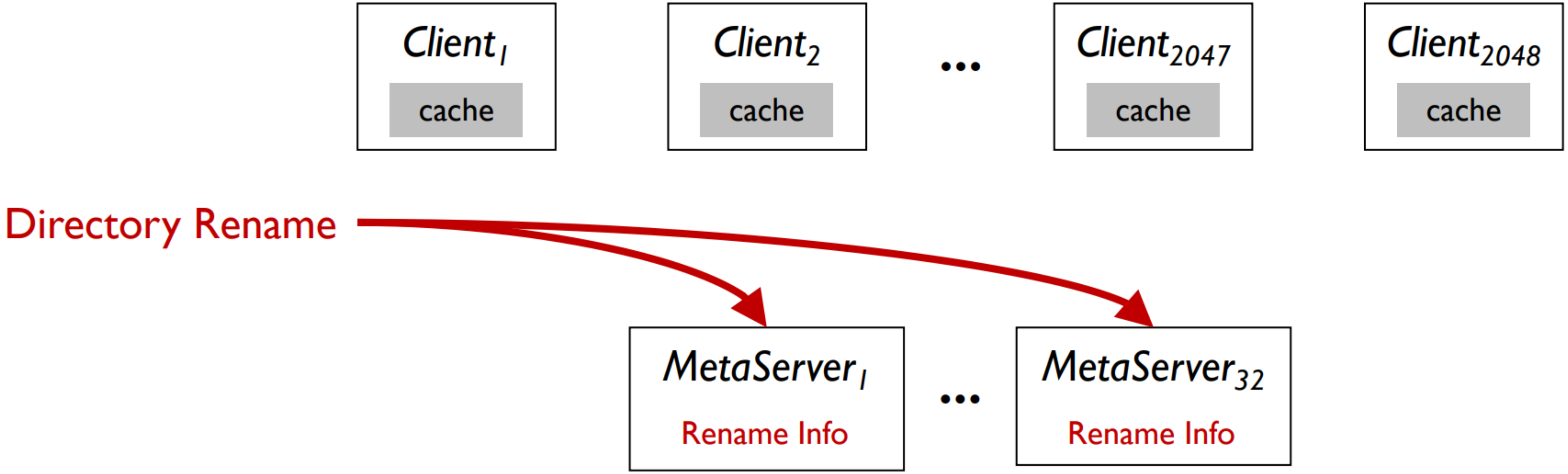


Huge number of Clients → **High coherence overhead**

Design Optimistic Access Metadata Cache

Key idea:

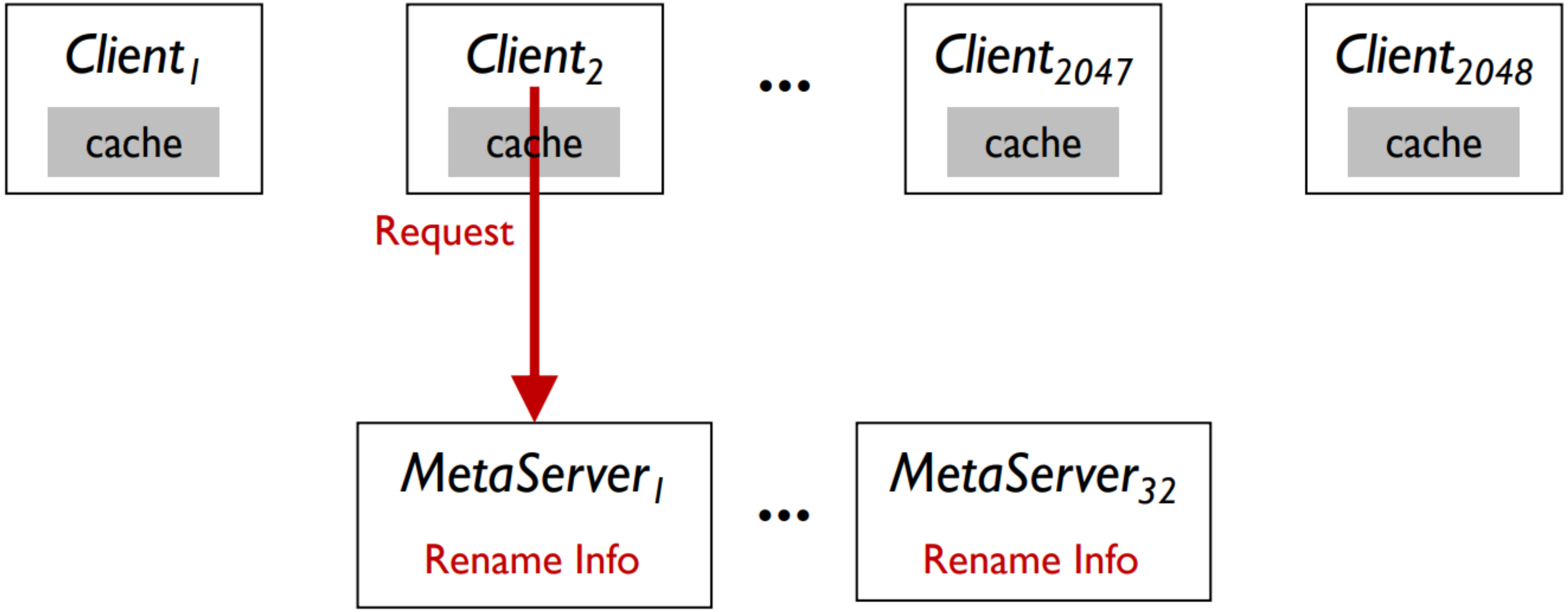
Validate cache staleness lazily on metadata servers



Design Optimistic Access Metadata Cache

Key idea:

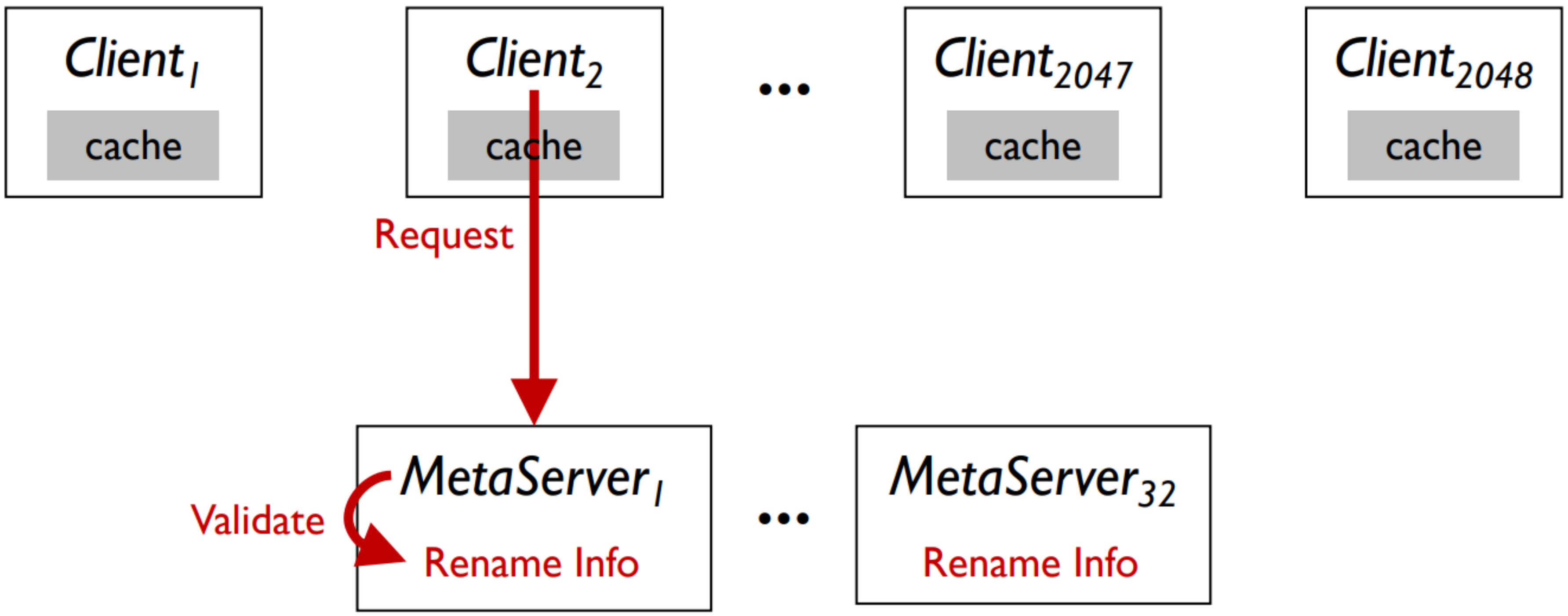
Validate cache staleness lazily on metadata servers



Design Optimistic Access Metadata Cache

Key idea:

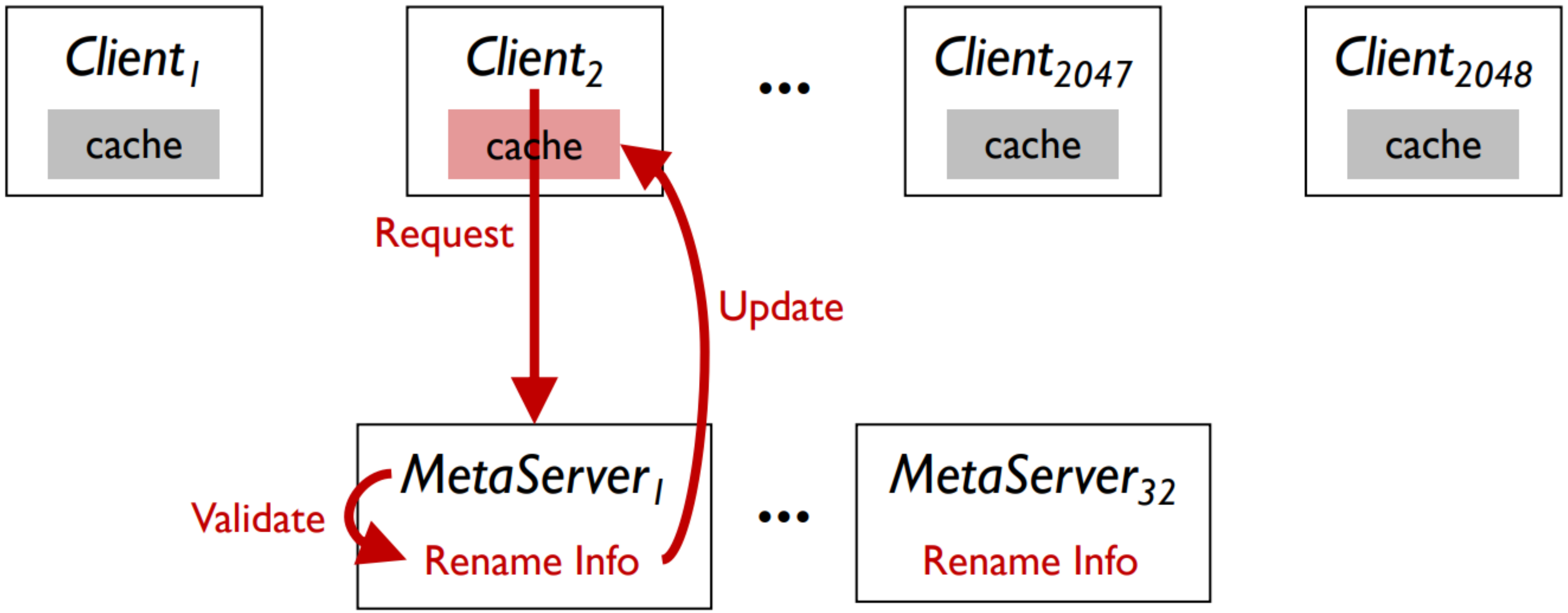
Validate cache staleness lazily on metadata servers



Design Optimistic Access Metadata Cache

Key idea:

Validate cache staleness lazily on metadata servers



Architecture

An efficient metadata service

Clients

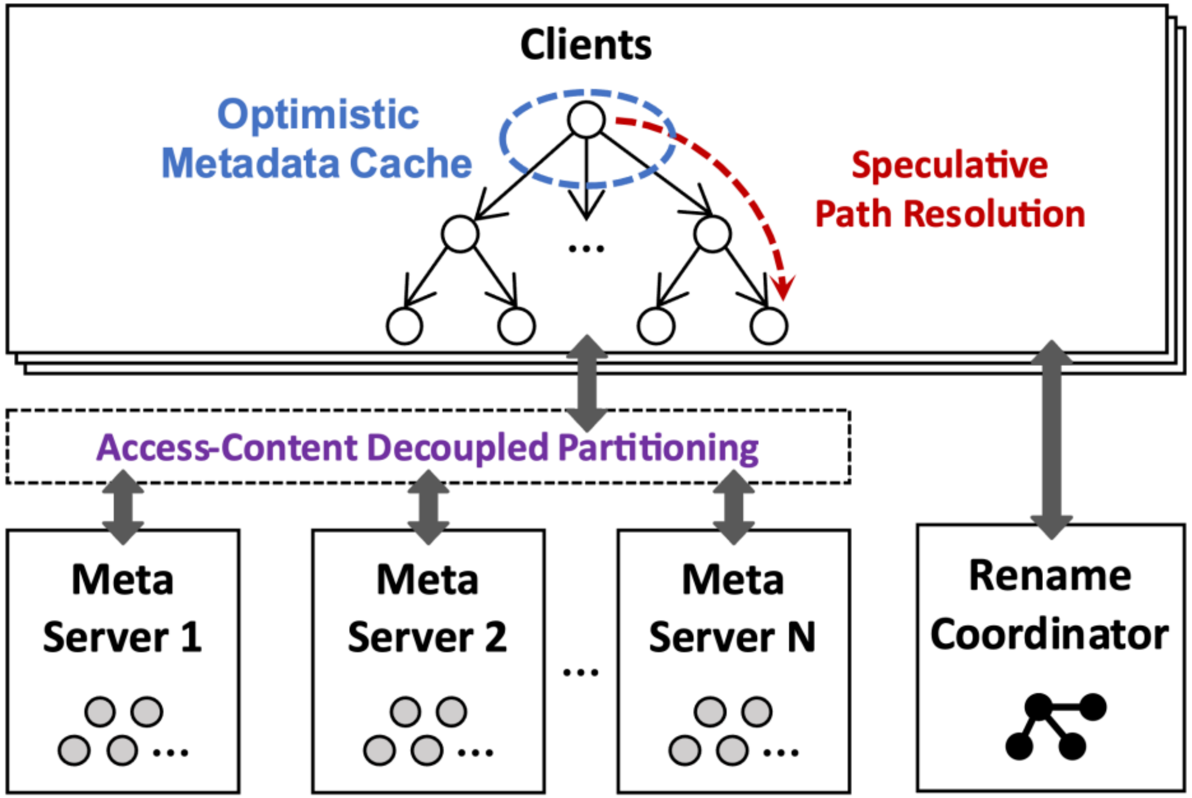
- ❖ Speculative path resolution
- ❖ Optimistic metadata cache

Metadata Servers

- ❖ Access-content decoupled partitioning

Rename Coordinator

- ❖ Check concurrent directory renames


Clients
Optimistic Metadata Cache
Speculative Path Resolution
...
Access-Content Decoupled Partitioning
Meta Server 1
Meta Server 2
...
Meta Server N
Rename Coordinator

Evaluation Experimental Setup

Hardware Platform

- 32 server nodes; 32 client nodes; up to 100 billion files

CPU	Intel Xeon Platinum 2.50GHz, 96 cores
Memory	Micron DDR4 2666MHz 32GB × 16
Storage	RAMdisk
Network	ConnectX-4 Lx Dual-port 25Gbps

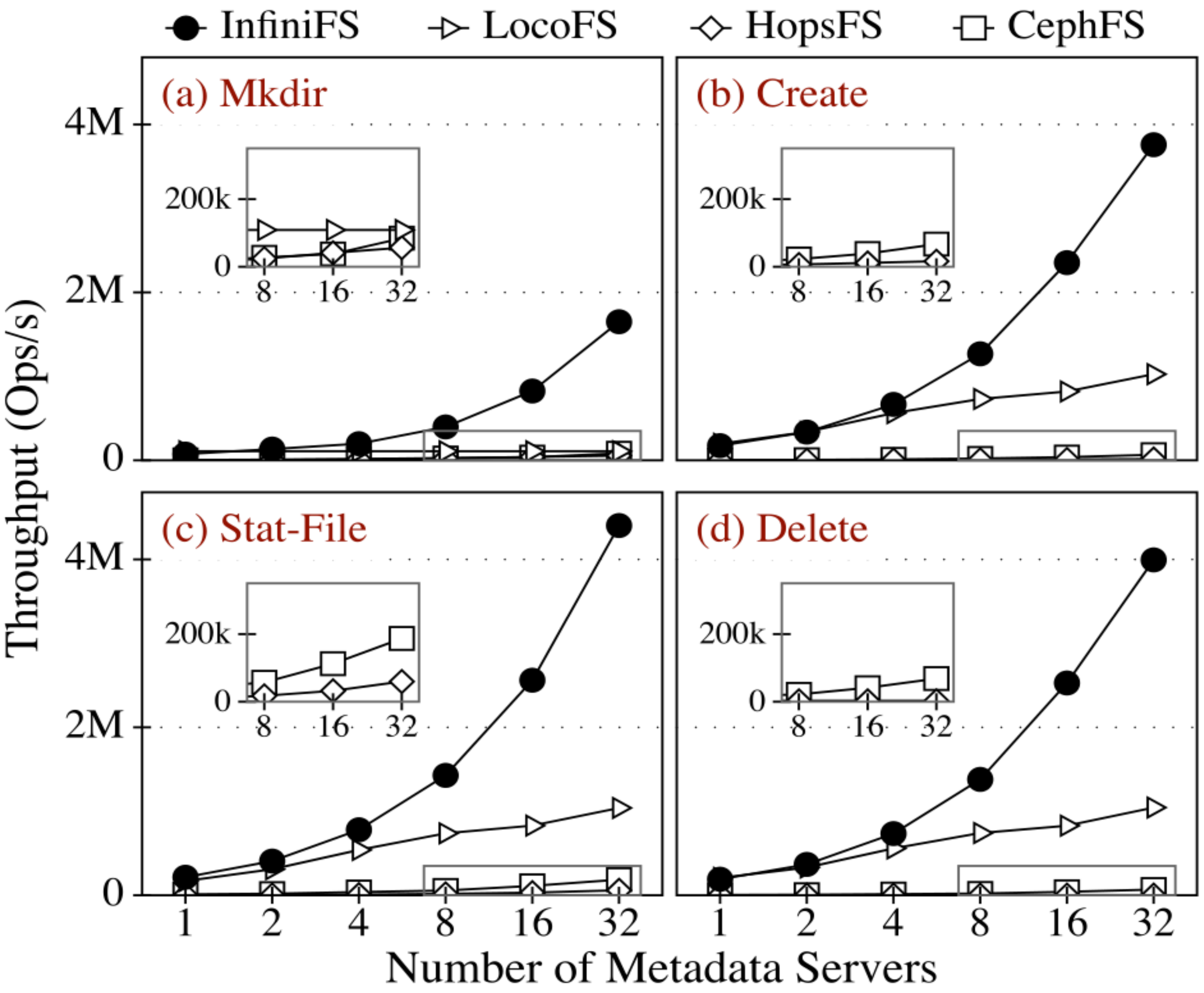
Compared System

- LocoFS [SC '17], HopsFS [FAST '17] , IndexFS [SC '14], CephFS [OSDI '06]

Benchmark

- The *mdtest* benchmark
- All tests create files of zero length

Evaluation Throughput

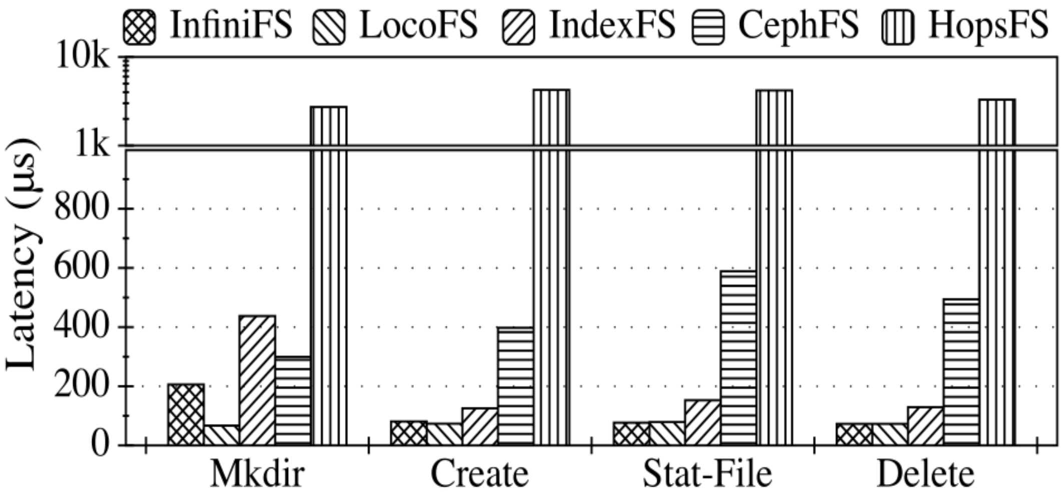


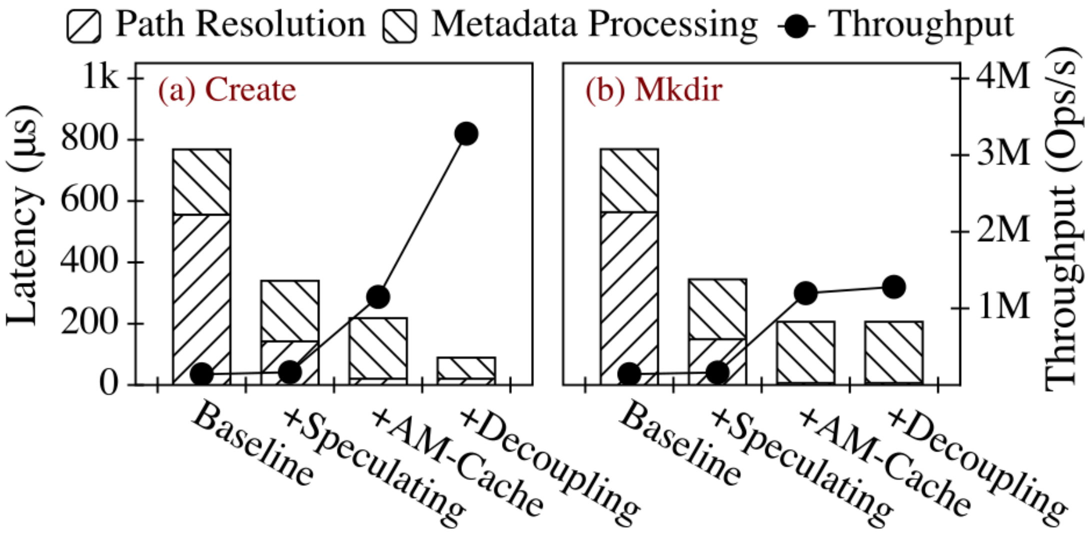
- Under a single metadata server, the infinifs file create performance (180K OPS) is slightly lower than that of locofs (200K OPS)

- Under 32 metadata servers, the performance of infinifs directory MKDIR and file stat are 18x and 4x that of locofs, respectively.

- The performance of infinifs is much better than that of hopsfs and cephfs, and its file create performance is 73x and 23x that of hopsfs, respectively

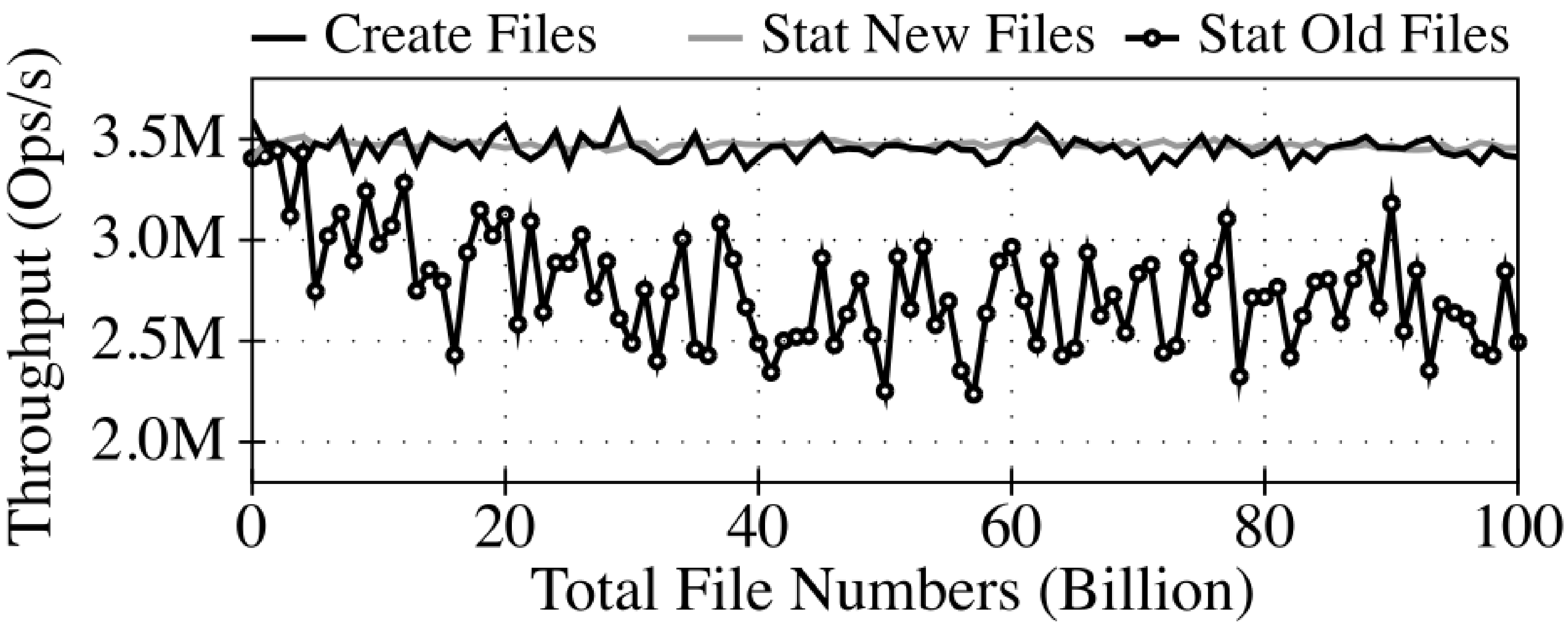
Evaluation Latency





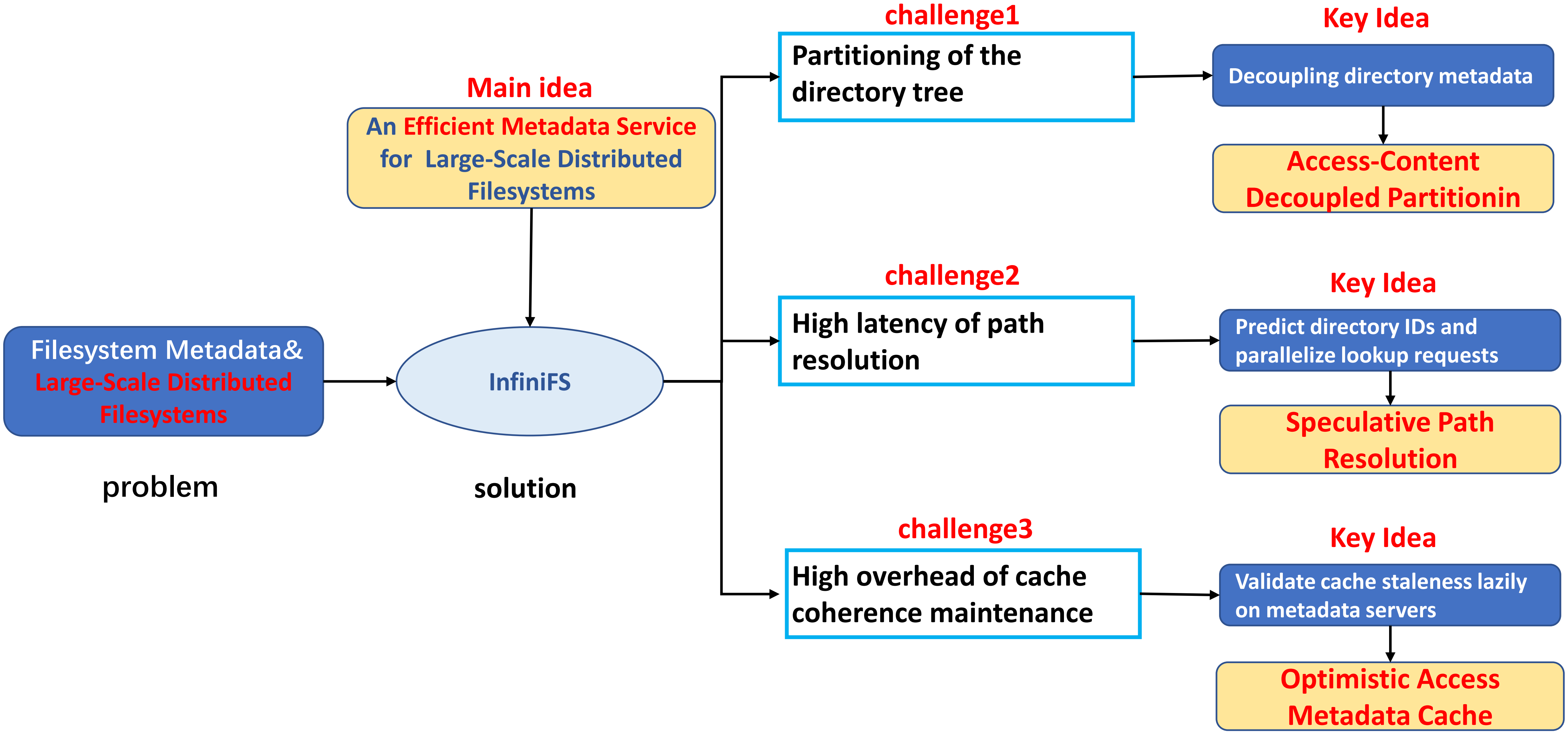
- For file create / STAT / delete operations, infinifs is equivalent to locofs, and the latency is relatively low.
- Infinifs operation delay is much lower than indexfs, cephfs and hopsfs

Evaluation Large-Scale Directory Tree



- Under the order of 100billion, infinifs can still provide stable create/stat performance, about 3.5m ops/sec.
- The throughput of stat old files is lower than that of stat new files. The reason is that rocksdb uses multi-level sstables to save key value pairs, which has the problem of read amplification.

Conclusion



Filesystem Metadata& Large-Scale Distributed Filesystems

problem

Main idea

An Efficient Metadata Service for Large-Scale Distributed Filesystems

InfiniFS

solution

challenge1

Partitioning of the directory tree

Key Idea

Decoupling directory metadata

Access-Content Decoupled Partitionin

challenge2

High latency of path resolution

Key Idea

Predict directory IDs and parallelize lookup requests

Speculative Path Resolution

challenge3

High overhead of cache coherence maintenance

Key Idea

Validate cache staleness lazily on metadata servers

Optimistic Access Metadata Cache