Emergent Properties in Modular Storage

A Study of Apple Desktop Applications, Facebook Messages, and Docker Containers

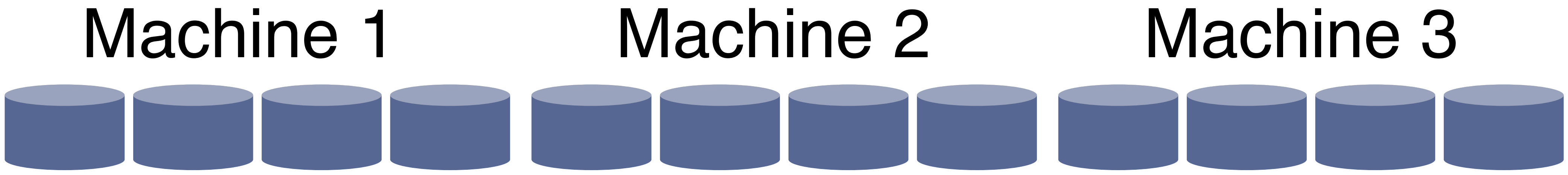
Tyler Harter

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

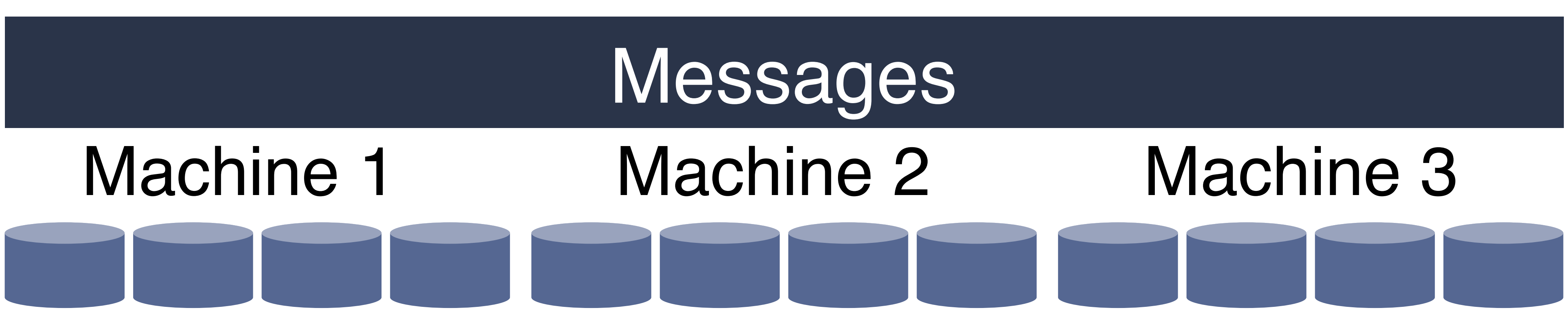
How are complex applications built?
(for example, Facebook Messages)

We have many machines with many disks.

How should we use them to store messages?



One option: use machines and disks directly.

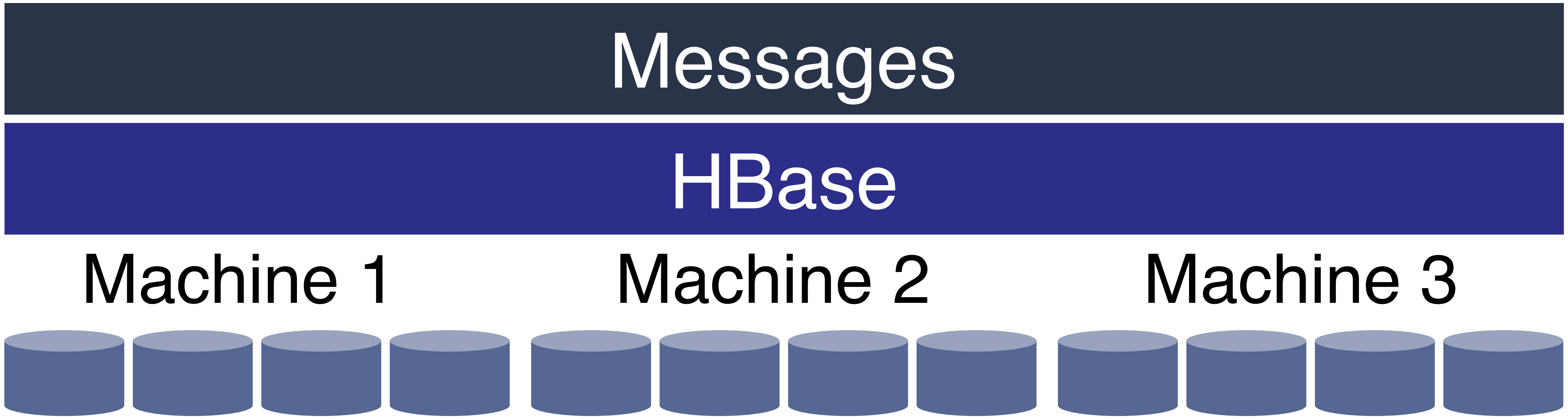


One option: use machines and disks directly.

Very specialized, but very high development cost.

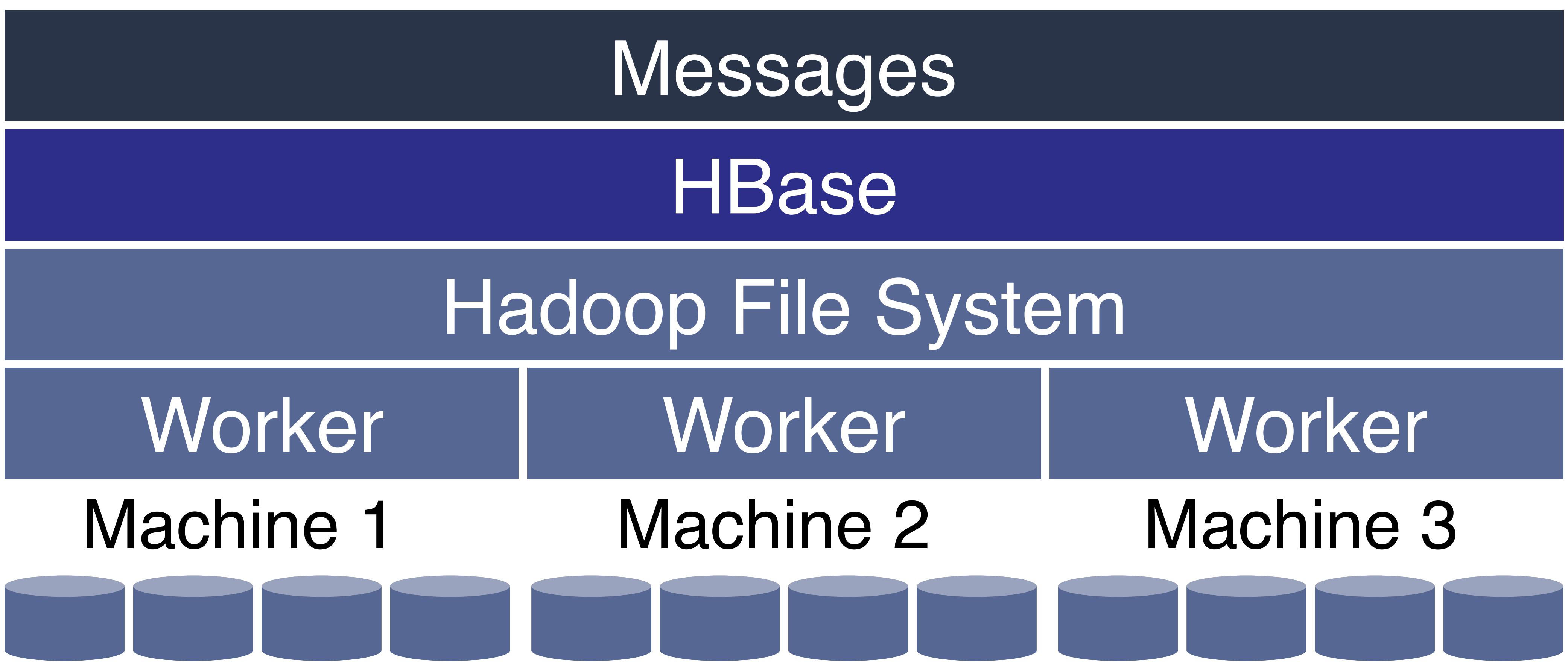

Messages
Machine 1
Machine 2
Machine 3

Use HBase for K/V logic

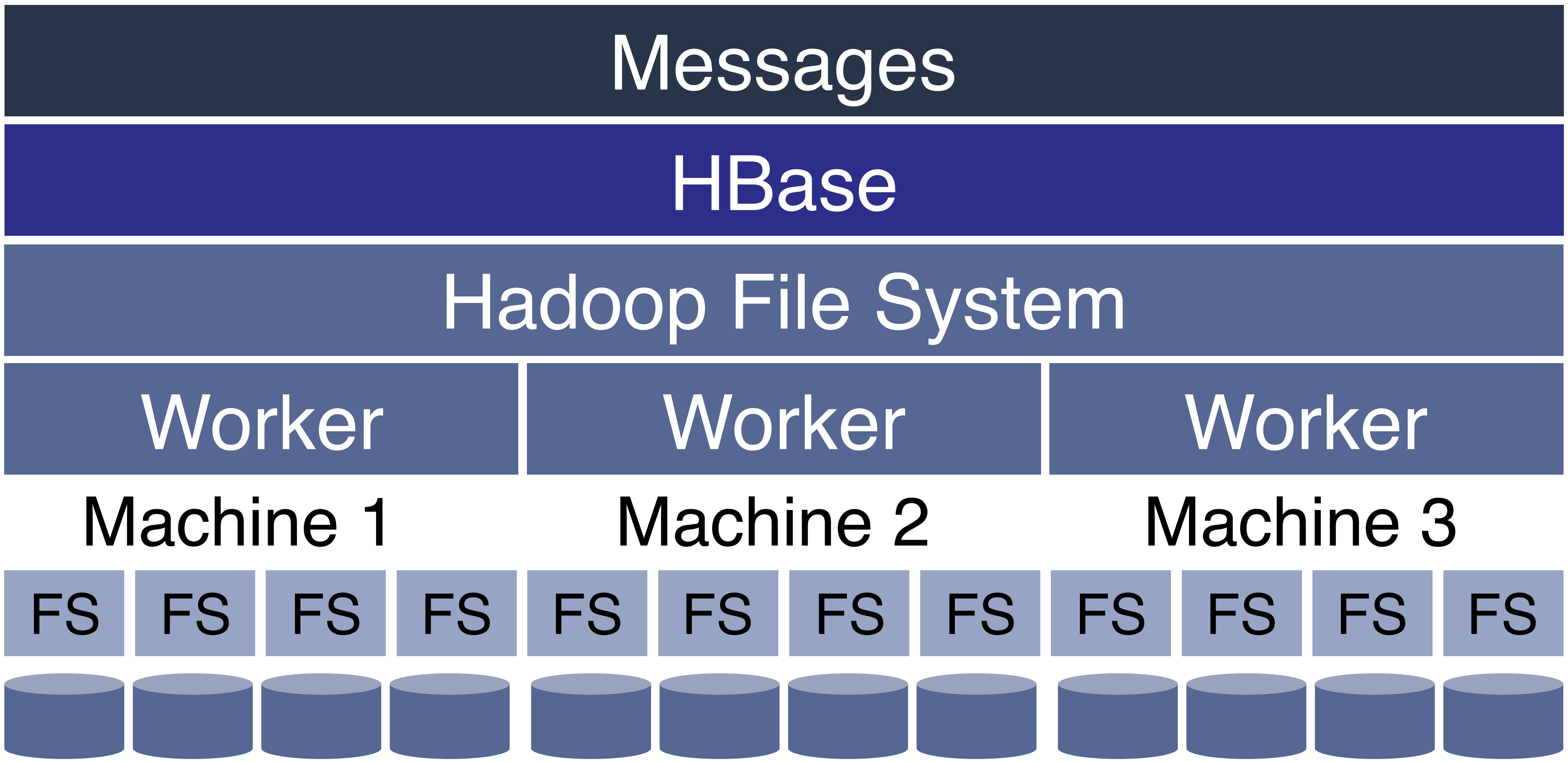


Use HBase for K/V logic

Use HDFS for replication



Use HBase for K/V logic
Use HDFS for replication
Use Local FS for allocation



Modules Divide Work

FB Messages (application logic)

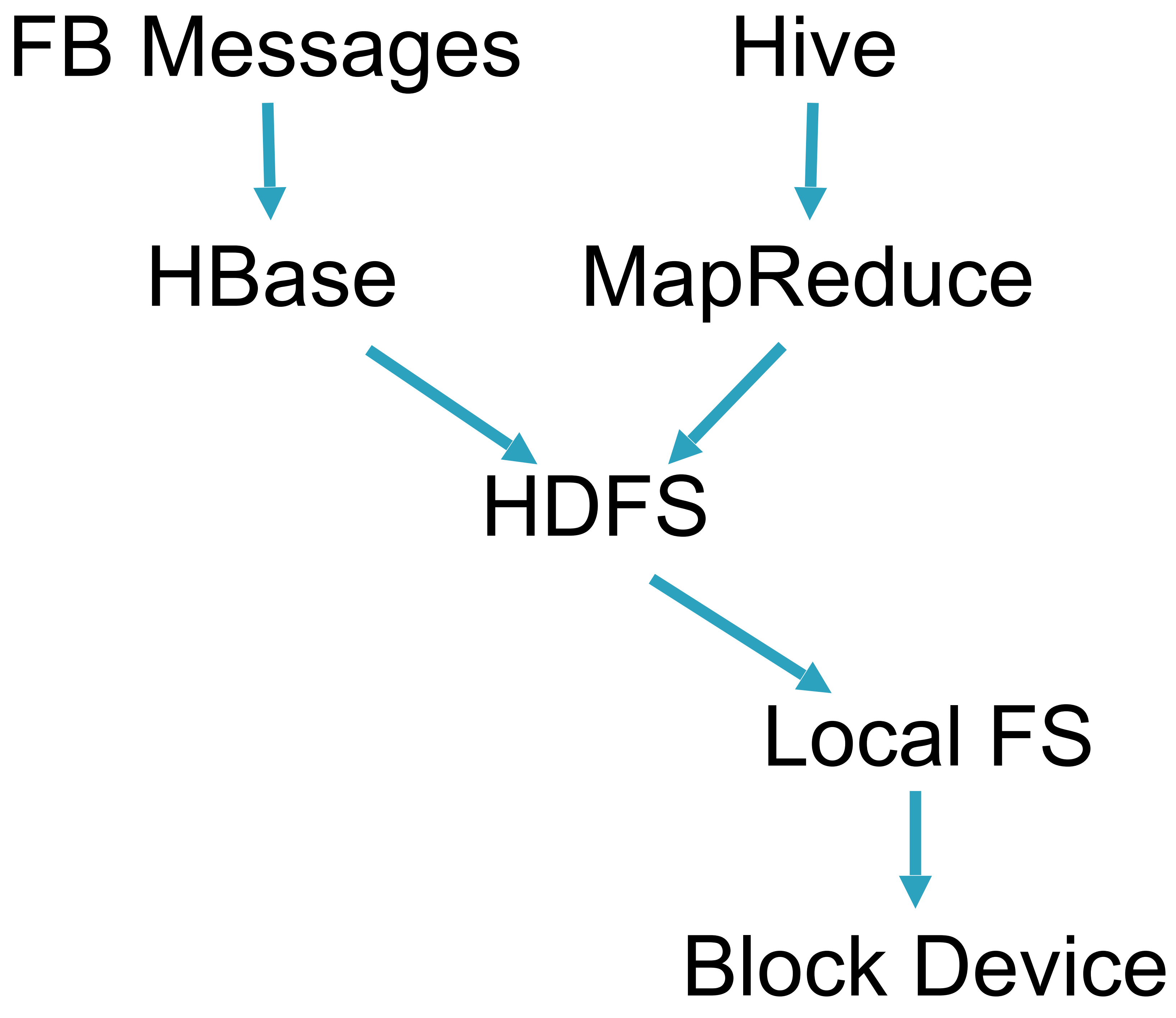
HBase (database logic)

HDFS (replication logic)

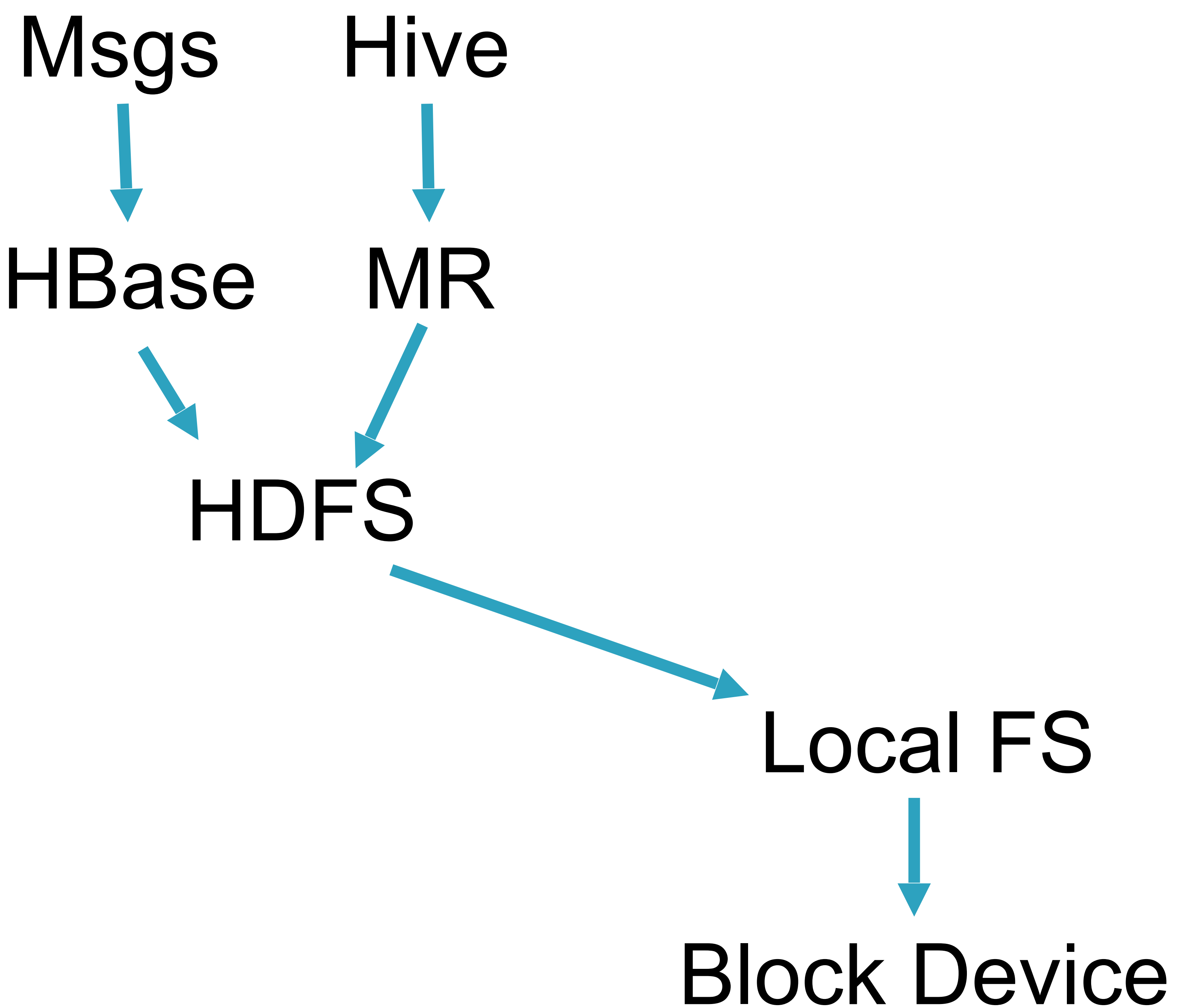
Local FS (allocation logic)

Block Device (scheduling logic)

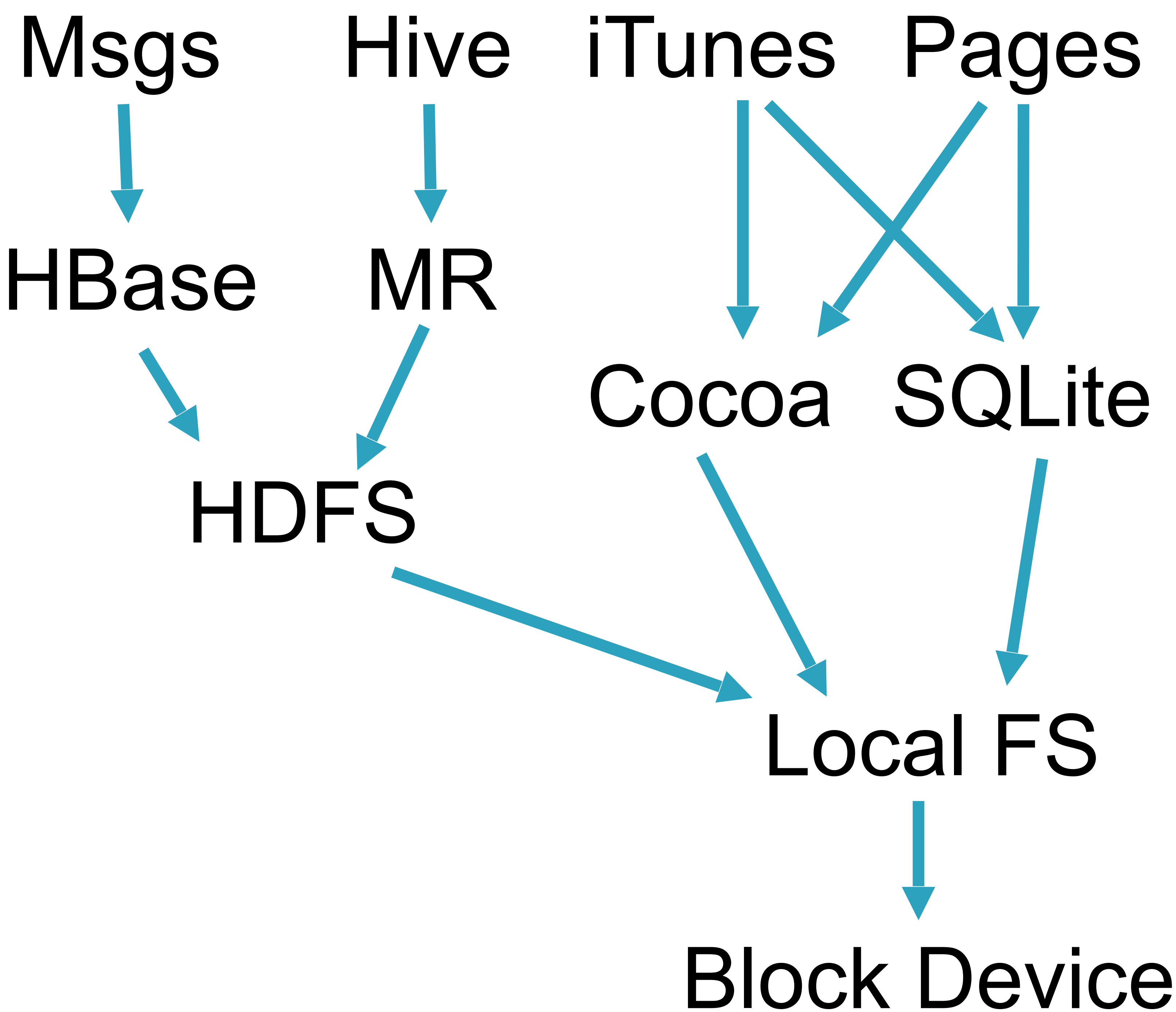
Modularity Enables Reuse



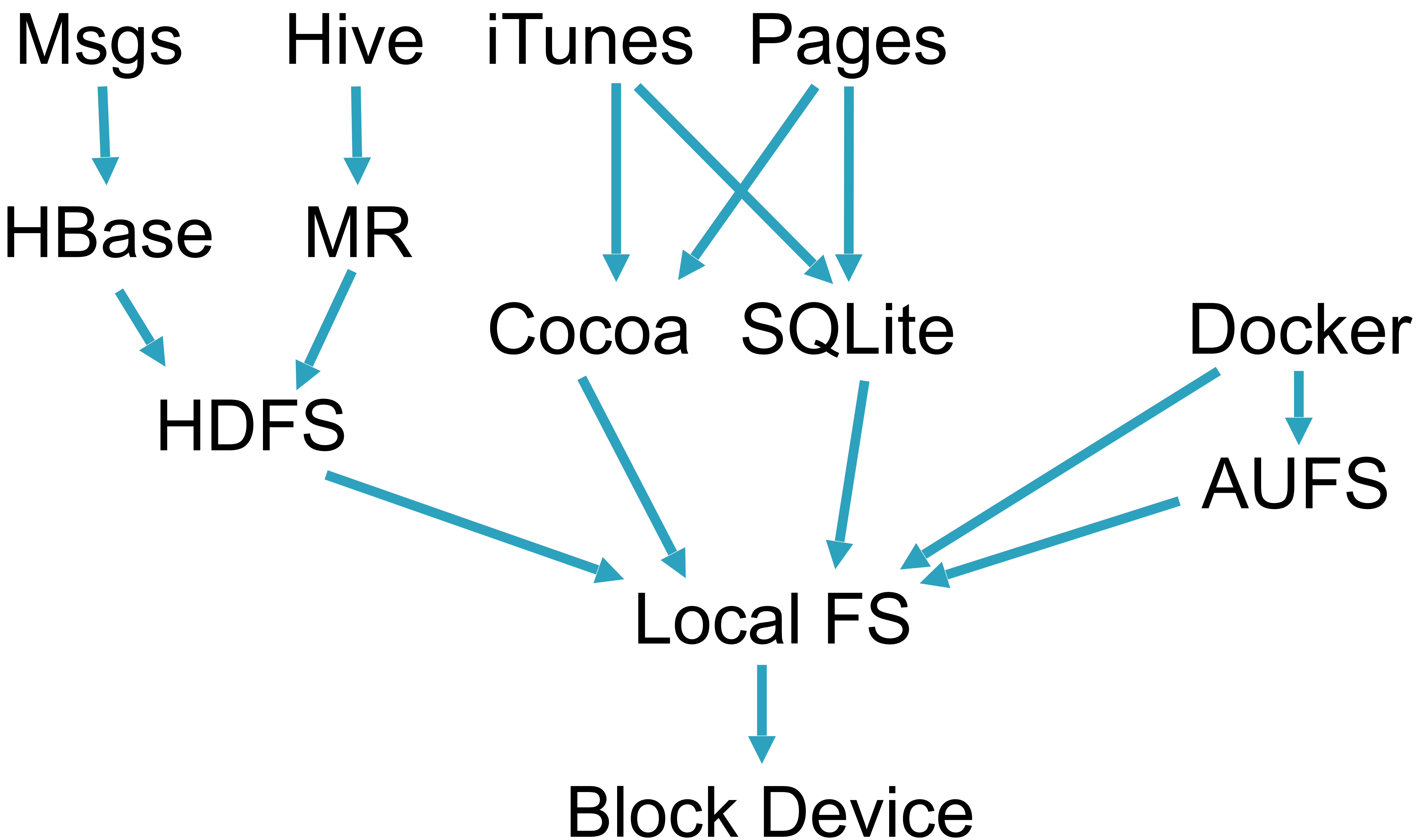
Modularity Enables Reuse



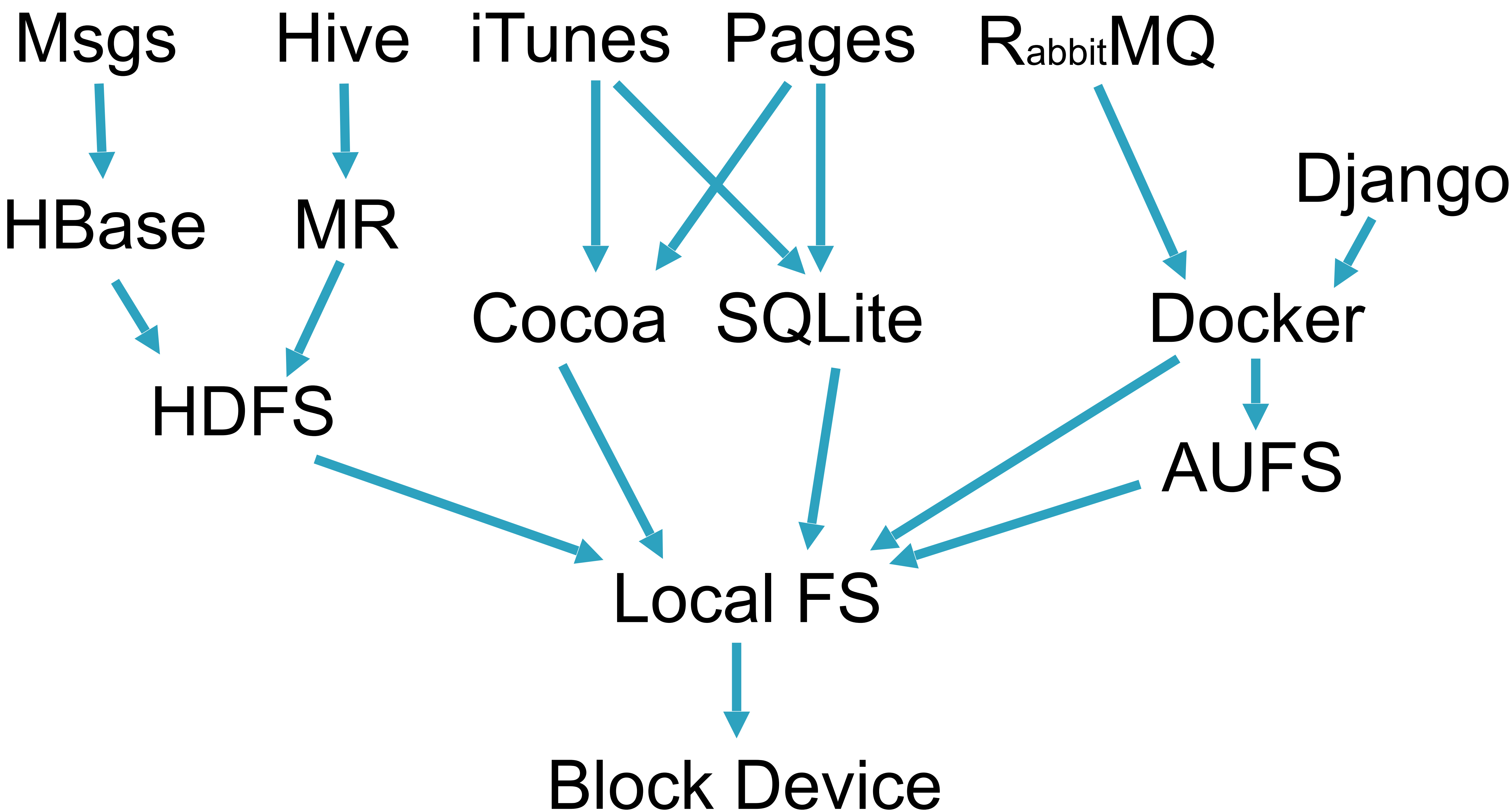
Desktop Applications



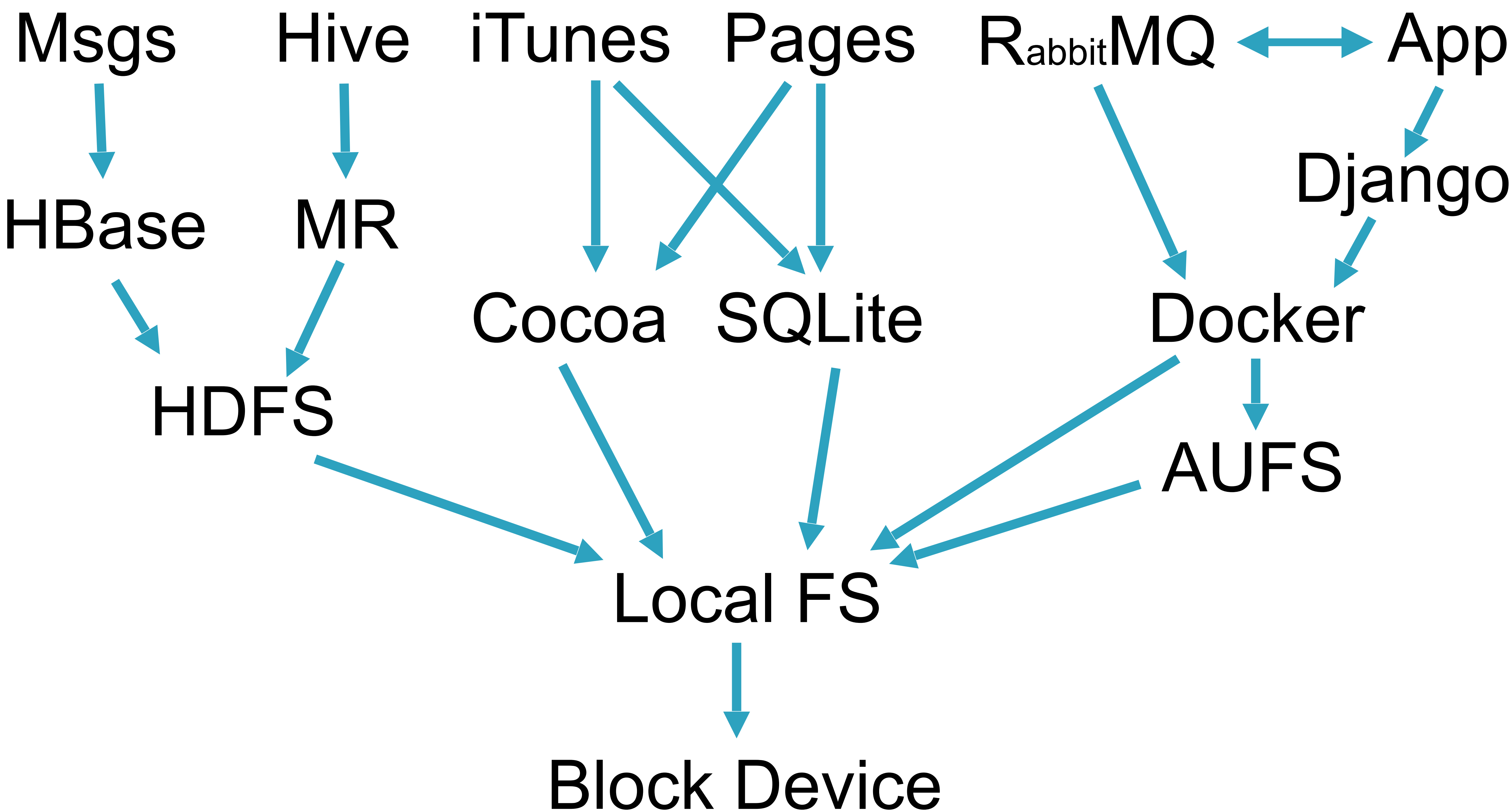
Docker Sandboxes



Microservices



Microservices



How are complex applications built?
(for example, Facebook Messages)

How are complex applications built?
(for example, Facebook Messages)

Answer: *by gluing together existing components*

Conceptual Integrity

Conceptual integrity *“dictates that the design must proceed from one mind, or from a very small number of agreeing resonant minds.”*

~ Frederick Brooks, *The Mythical Man-Month*

Conceptual Integrity

Conceptual integrity *"dictates that the design must proceed from one mind, or from a very small number of agreeing resonant minds."*

~ Frederick Brooks, *The Mythical Man-Month*

Premise: modern applications and storage systems are patched together and lack conceptual integrity.

Emergent Properties

Emergent Properties: *"properties that are not evident in the individual components, but they show up when combining those components"*

~ Saltzer and Kaashoek, *Principles of Computer System Design*

"they might also be called surprises"

Summarizing Modern Storage

Storage systems benefit from modular

- Modules divide work
- Modules enable reuse

But these systems lack conceptual integrity

Questions

- What are the storage needs of modern applications?
- What impact does modularity have on I/O patterns?
- How can we better modularize storage systems?

Outline

Motivation: Modularity in Modern Storage

Overview: Types of Modularity

Library Study: Apple Desktop Applications

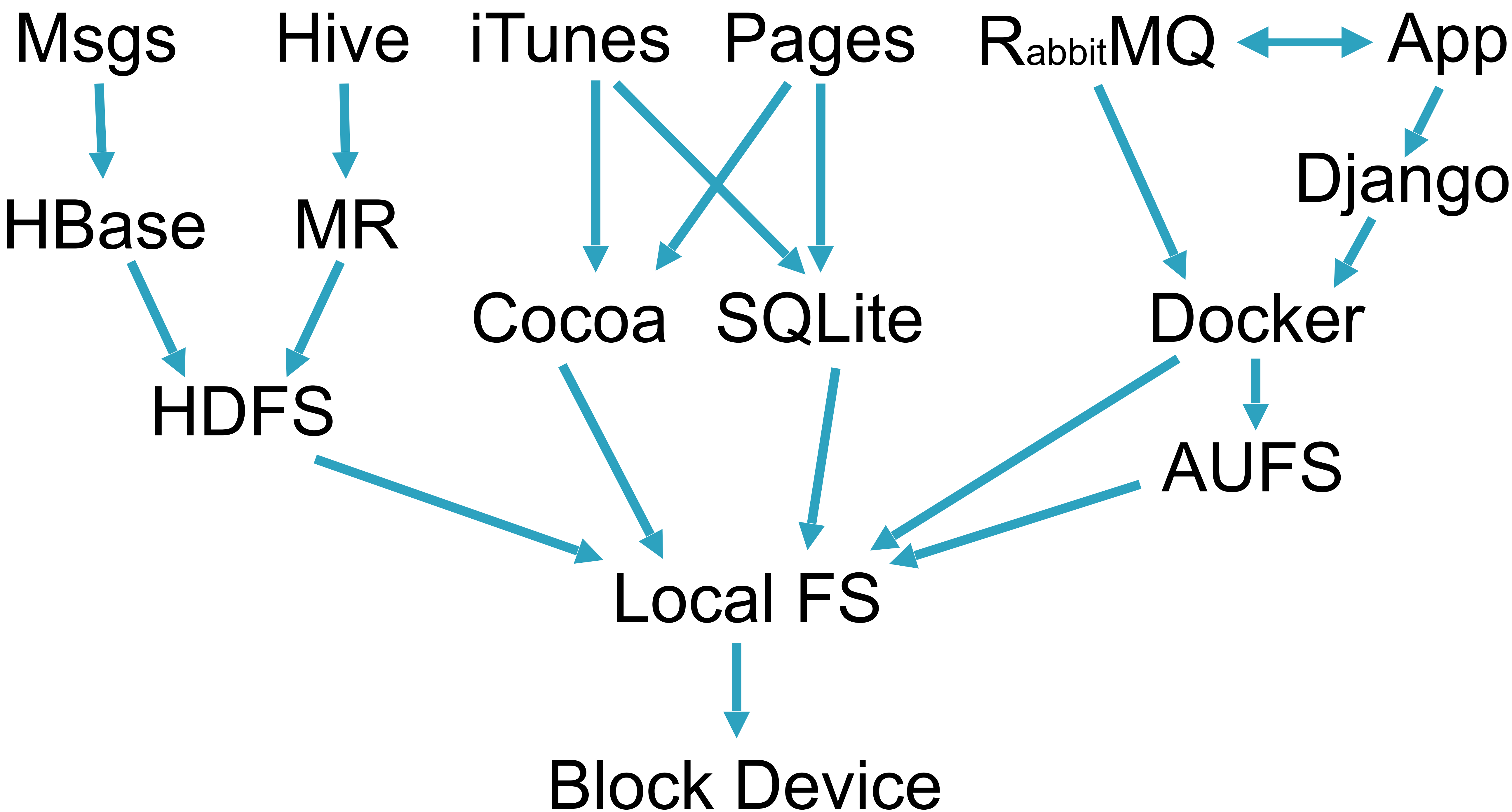
Layer Study: Facebook Messages

Microservice Study: Docker Containers

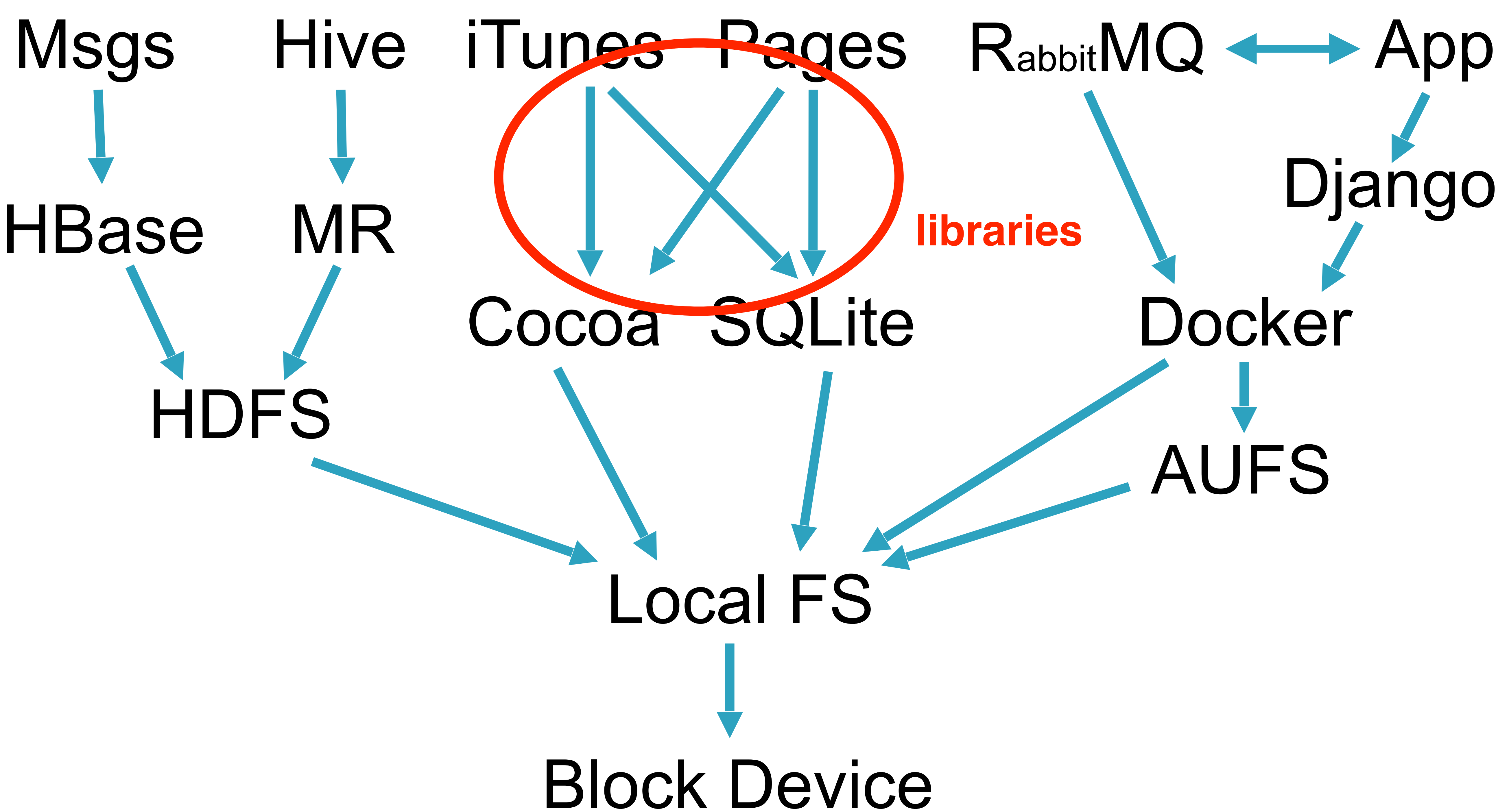
Slacker: a Lazy Docker Storage Driver

Conclusions

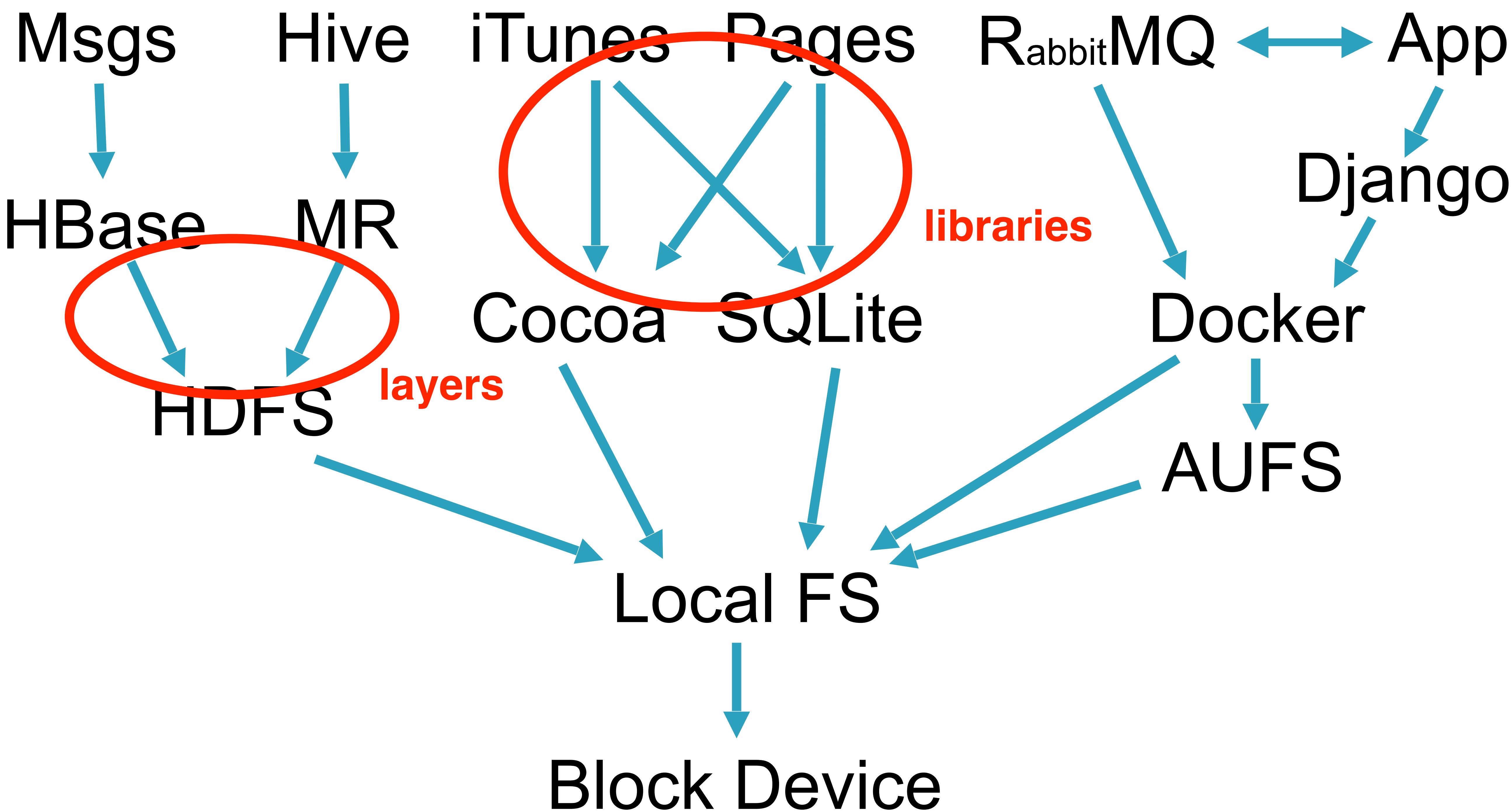
Many Types of Reuse



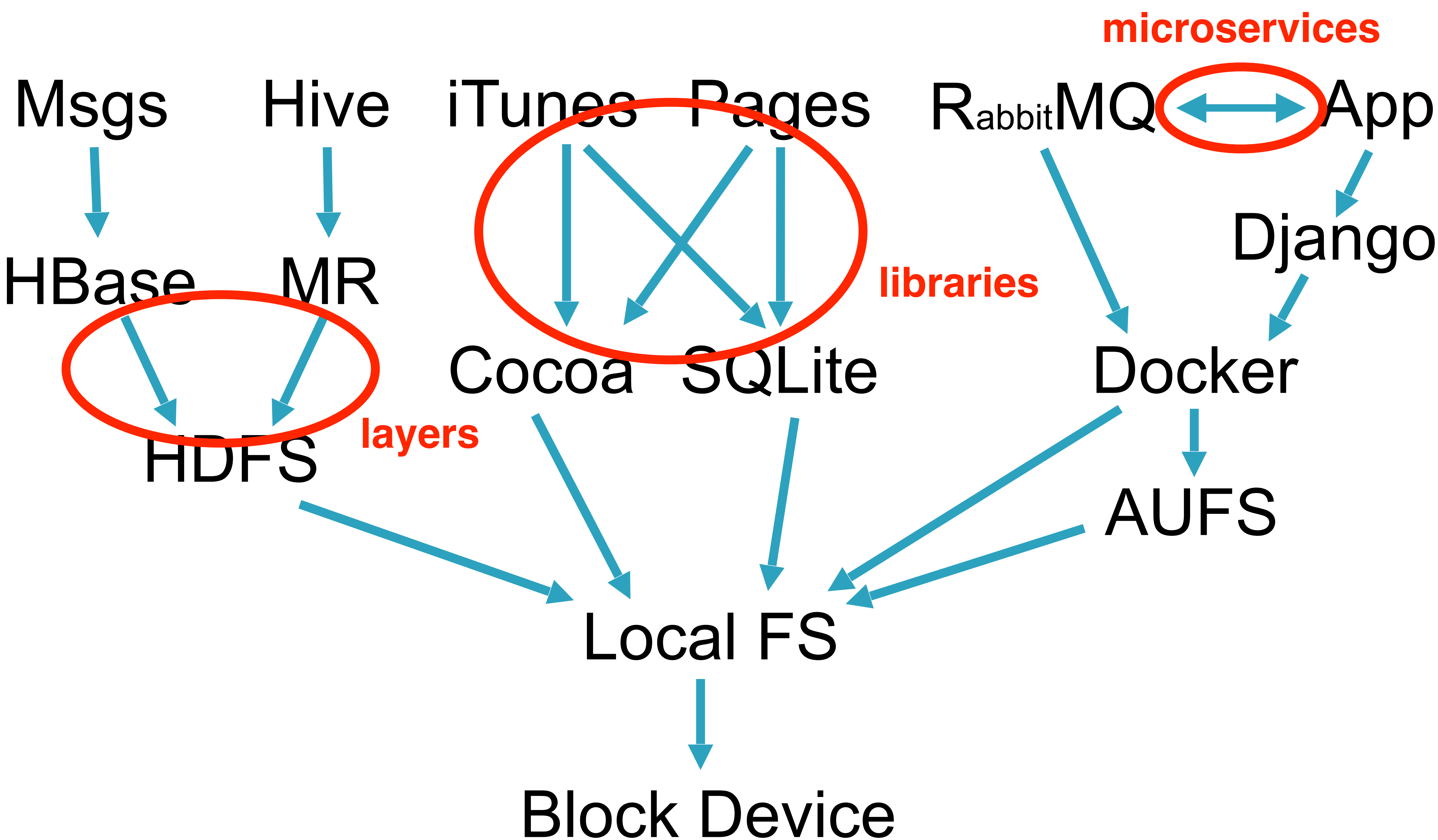
Many Types of Reuse



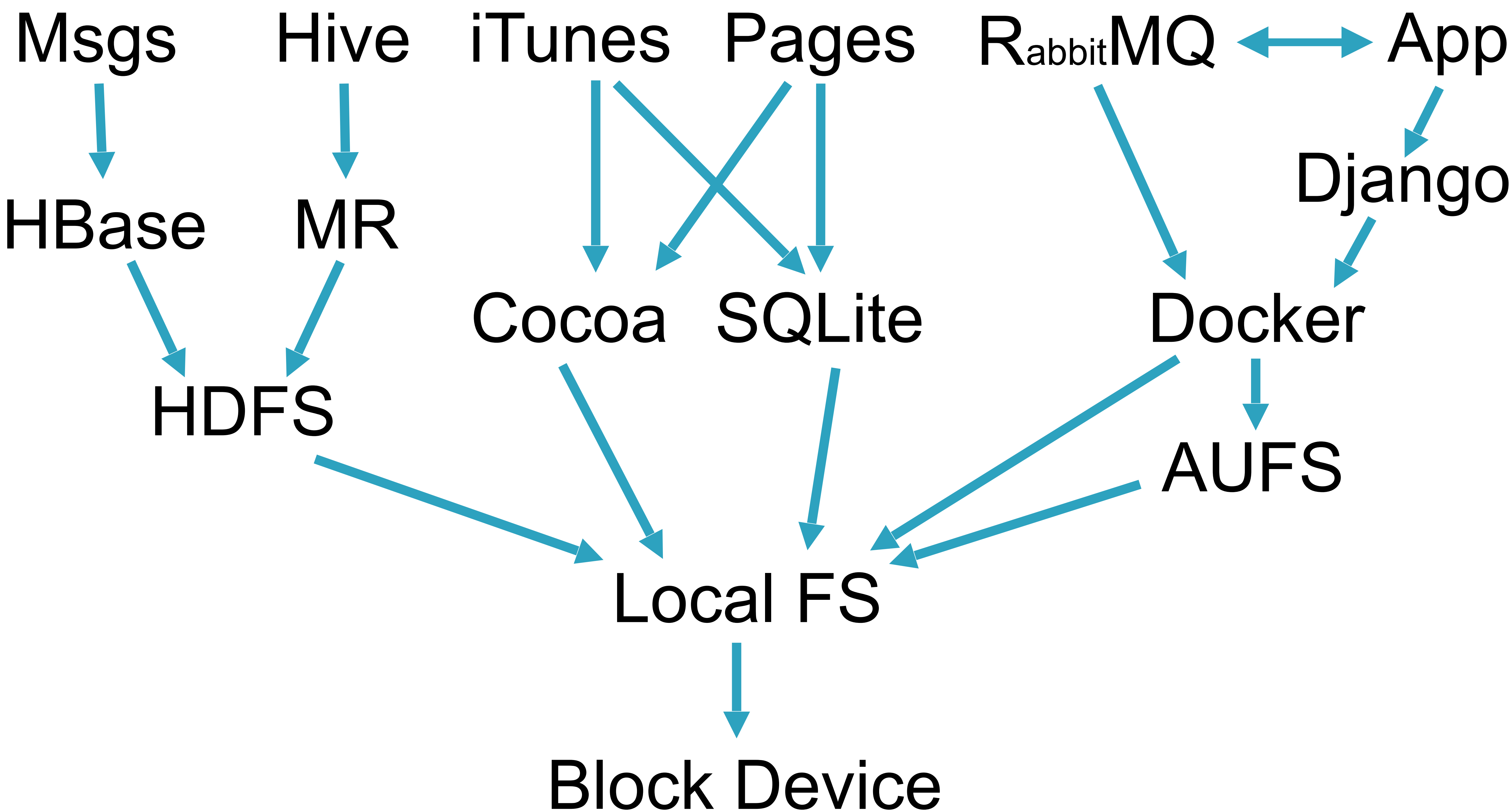
Many Types of Reuse



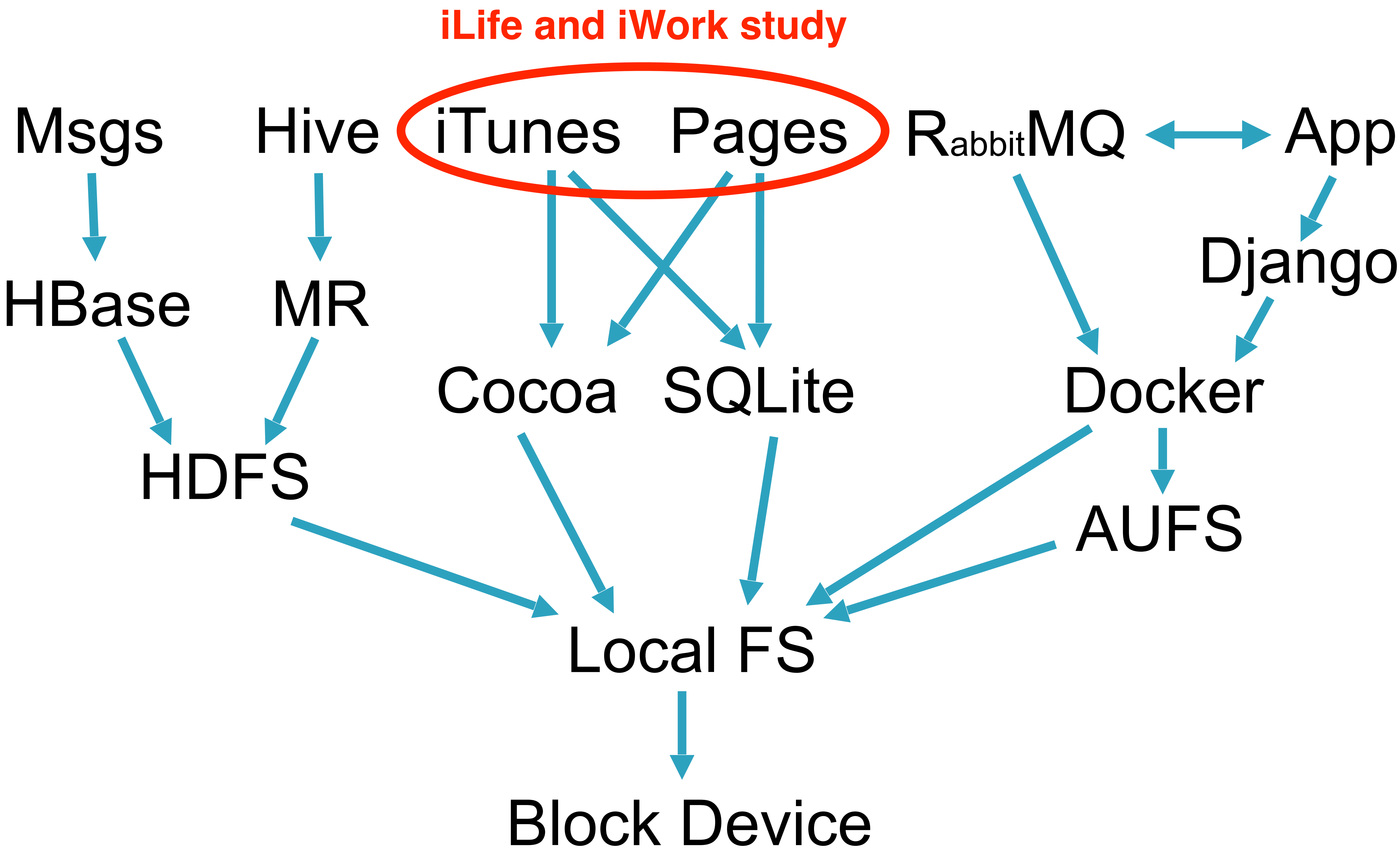
Many Types of Reuse



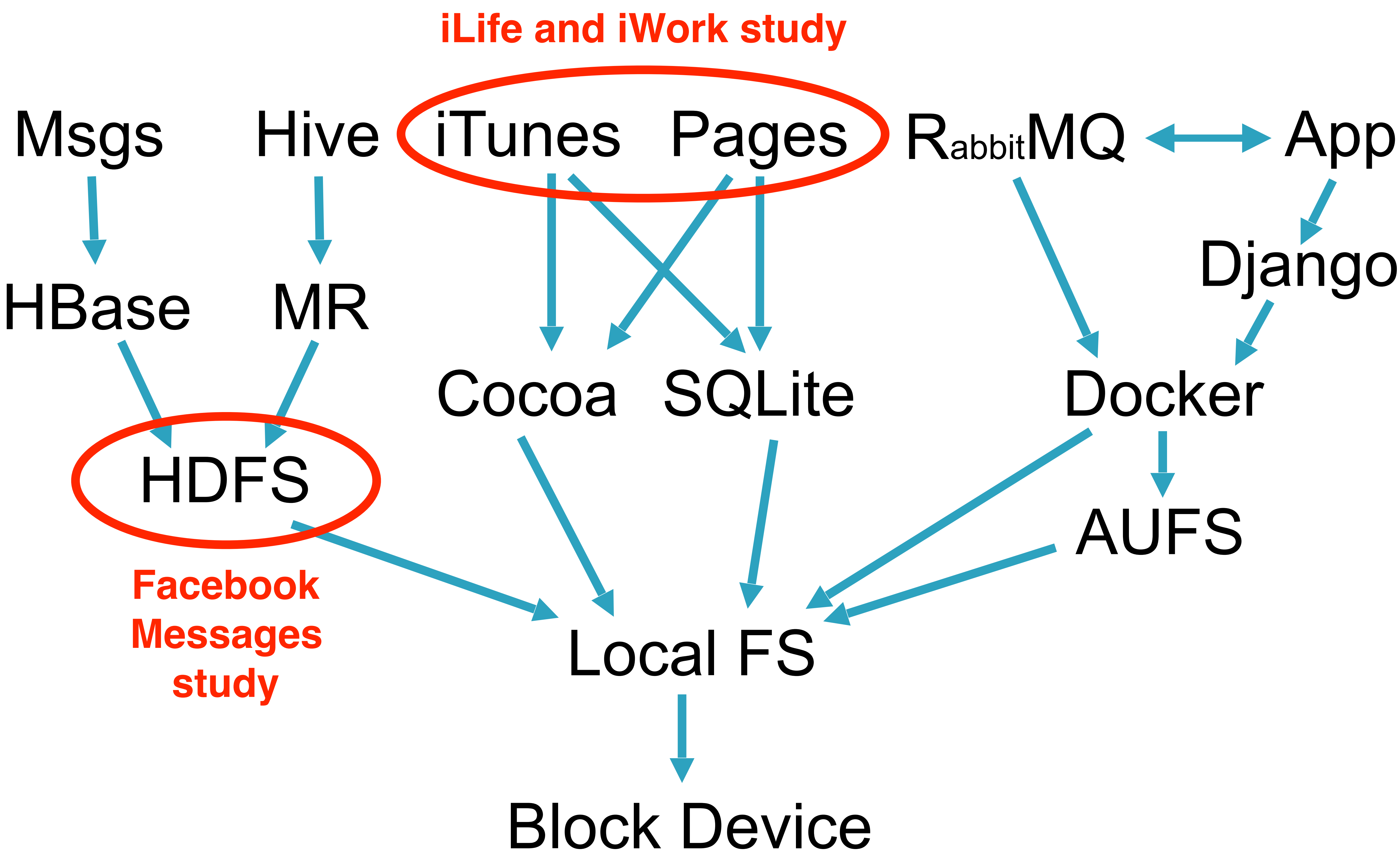
This Dissertation



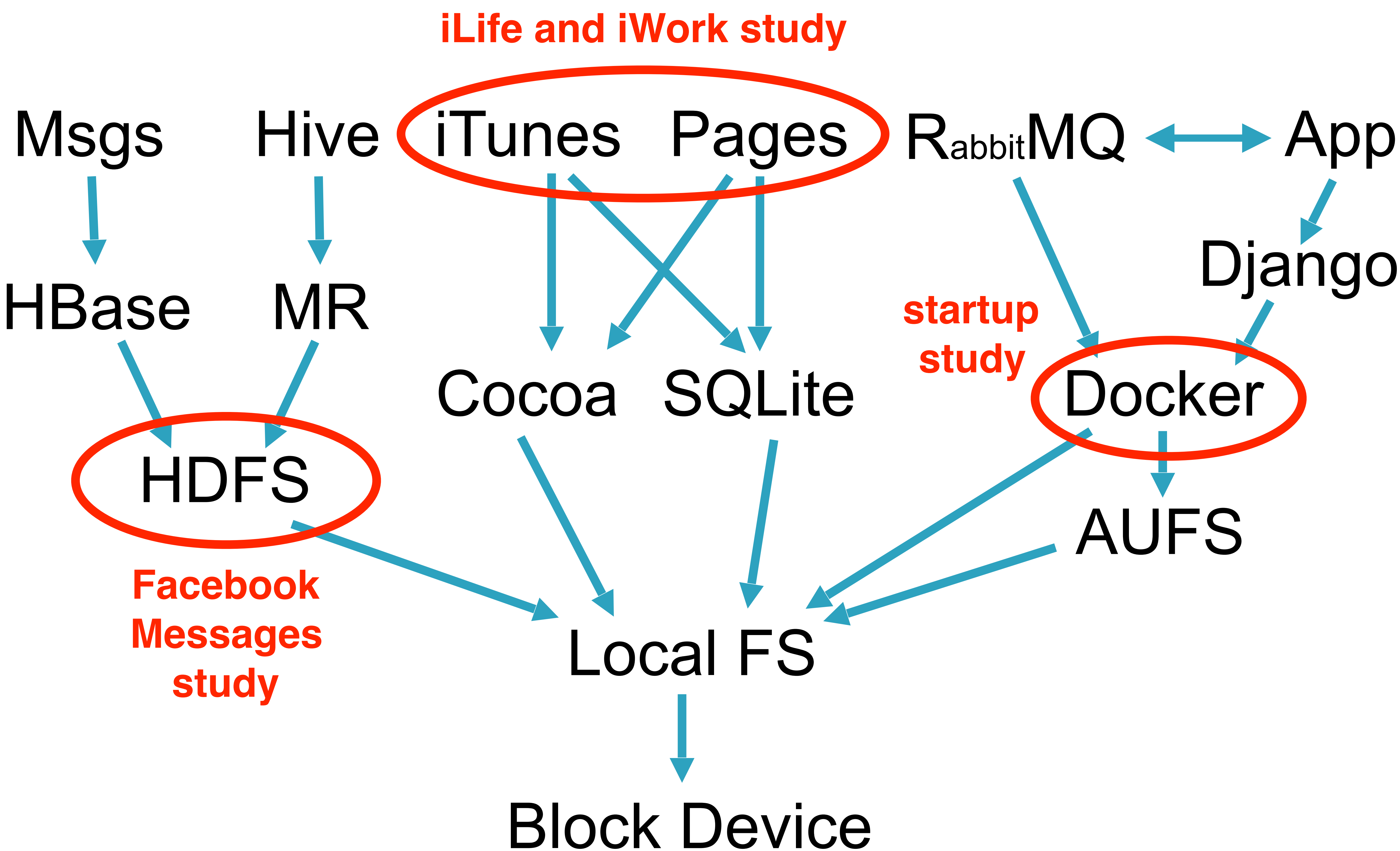
This Dissertation



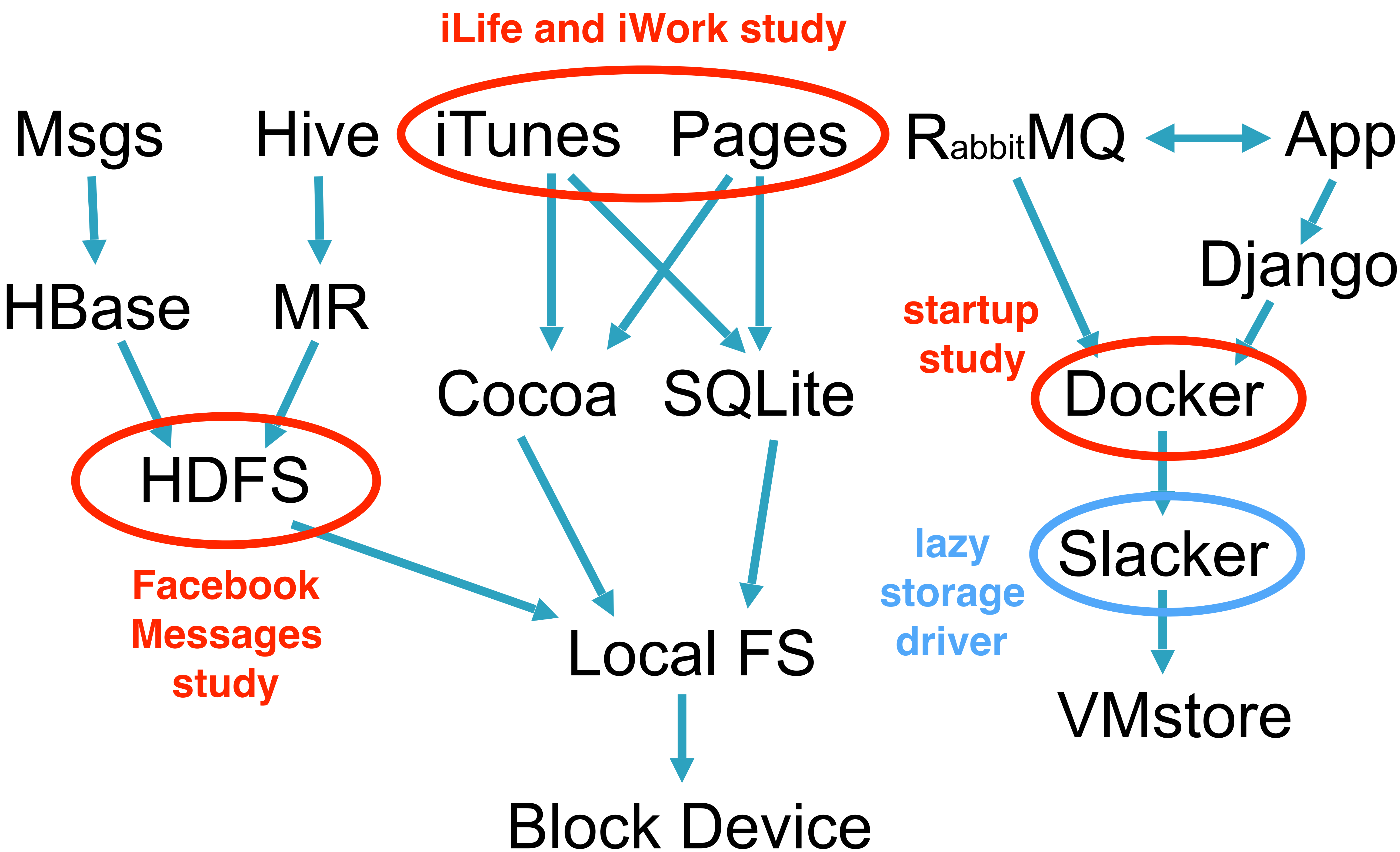
This Dissertation



This Dissertation



This Dissertation



Publications

SOSP ’11: A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau.

TOCS ’12: A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau.

FAST ’14: Analysis of HDFS Under HBase: A Facebook Messages Case Study. Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

;login ’14: Analysis of HDFS Under HBase: A Facebook Messages Case Study. Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

FAST ’16: Slacker: Fast Distribution with Lazy Docker Containers. Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

Outline

Motivation: Modularity in Modern Storage

Overview: Types of Modularity

Library Study: Apple Desktop Applications

Layer Study: Facebook Messages

Microservice Study: Docker Containers

Slacker: a Lazy Docker Storage Driver

Conclusions

Modern Desktop Applications and Libraries

In 1974:

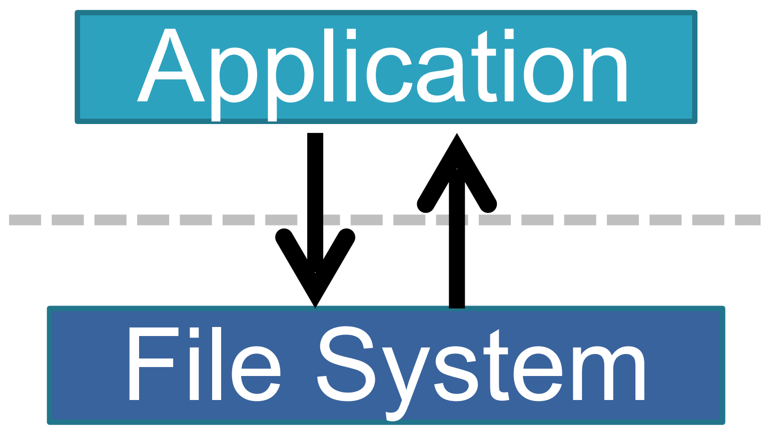
“No large ‘access method’ routines are required to insulate the programmer from the system calls; in fact, all user programs either call the system directly or use a small library program, only tens of instructions long…”

~ Ritchie and Thompson. The UNIX Time-Sharing System.

Modern Desktop Applications and Libraries

In the past, applications:

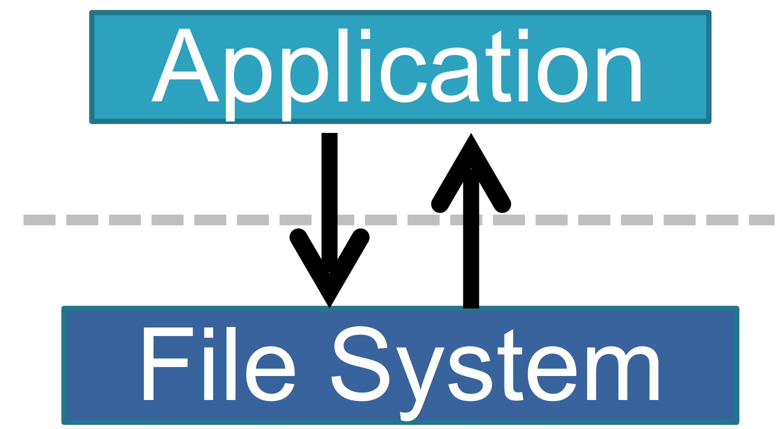
- Used the file-system API directly
- Performed simple tasks well
- Chained together for more complex actions



Modern Desktop Applications and Libraries

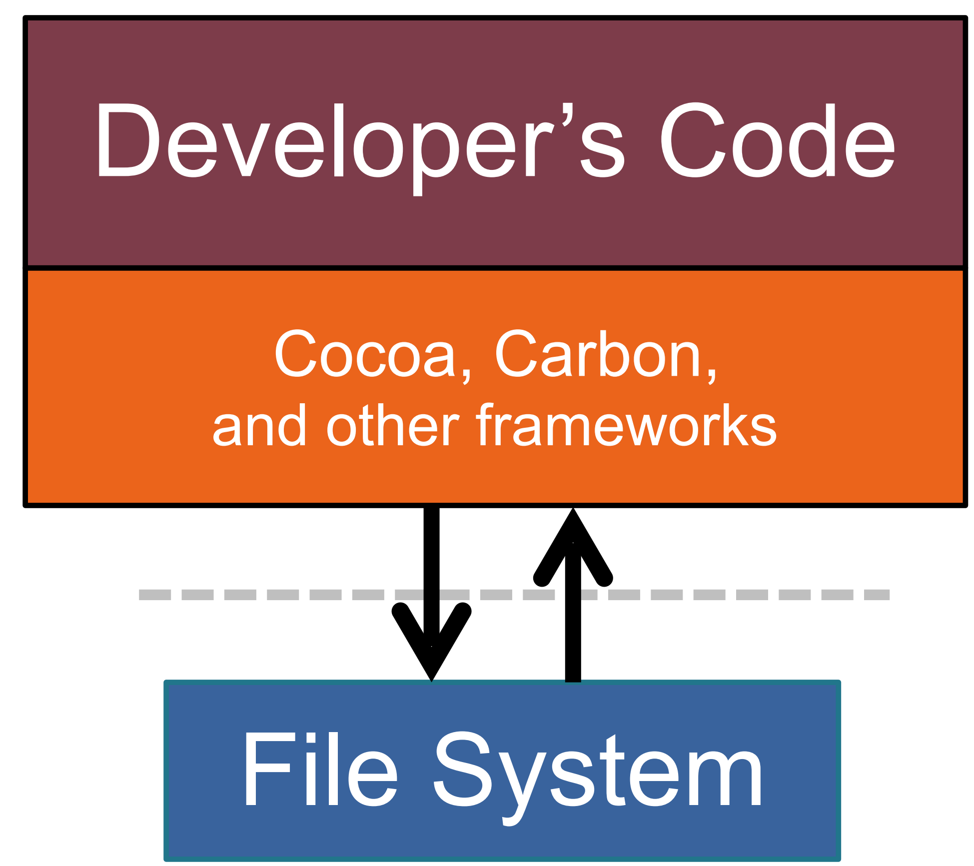
In the past, applications:

- Used the file-system API directly
- Performed simple tasks well
- Chained together for more complex actions



Today, we see:

- Applications are graphically rich, multifunctional monoliths
- *“#include <Cocoa/Cocoa.h>*
  reads 112,047 lines from 689 files”
  ~ Rob Pike ‘10
- They rely heavily on I/O libraries



Our Study

Measure 34 tasks from popular home-user applications

- **iLife suite (multimedia)**
  - iPhoto 8.1.1
  - iTunes 9.0.3
  - iMovie 8.0.5
- **iWork (like MS Office)**
  - Pages 4.0.3 (*Word*)
  - Numbers 2.0.3 (*Excel*)
  - Keynote 5.0.3 (*PowerPoint*)

Goal: understand I/O patterns and impact of libraries

Our Study

Measure 34 tasks from popular home-user applications

- **iLife suite (multimedia)**
  - iPhoto 8.1.1
  - iTunes 9.0.3
  - iMovie 8.0.5
- **iWork (like MS Office)**
  - Pages 4.0.3 (*Word*)
  - Numbers 2.0.3 (*Excel*)
  - Keynote 5.0.3 (*PowerPoint*)

This talk: look at one task from Pages in detail as case study

A Case Study: Saving a Document

Application: Pages 4.0.3

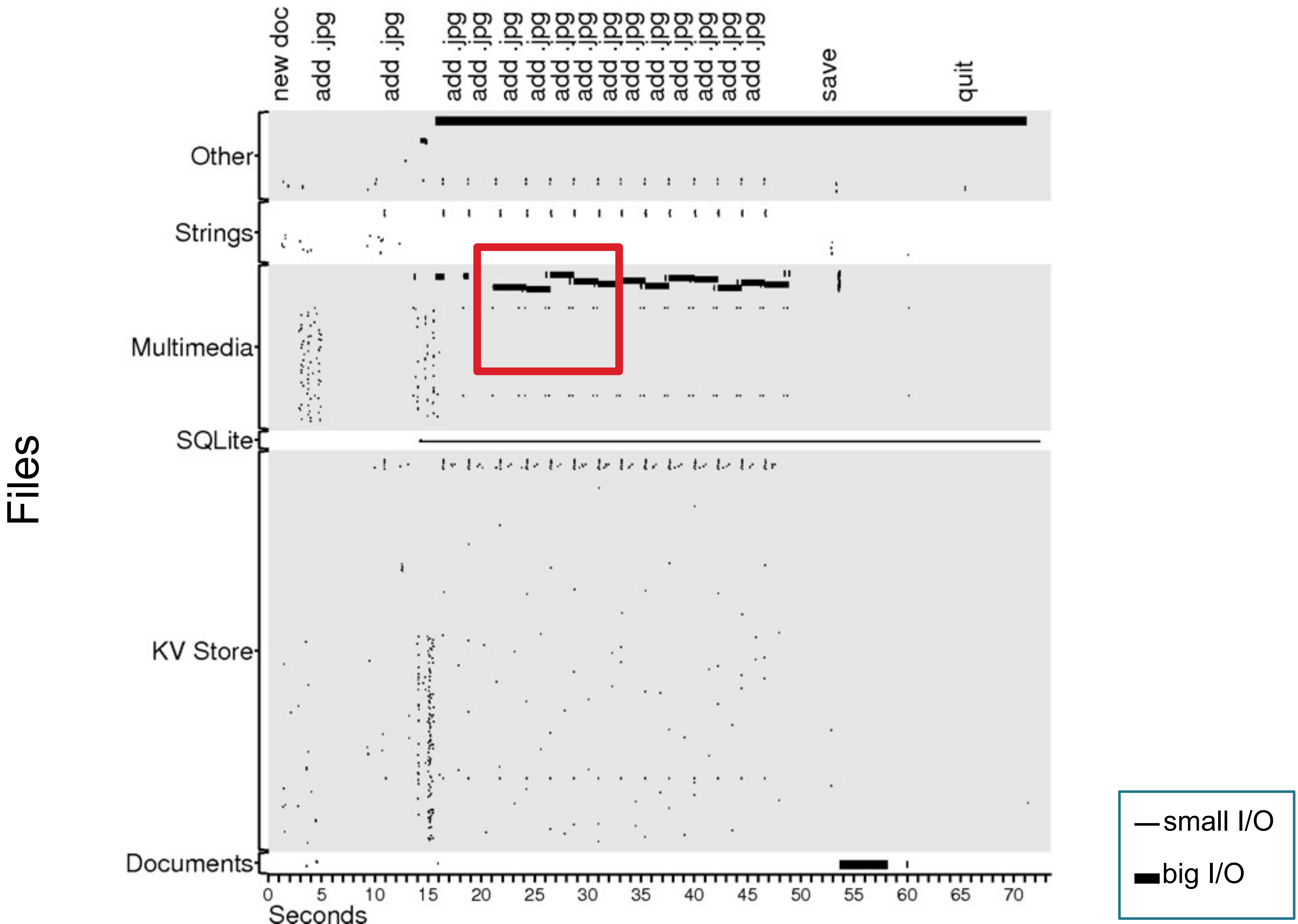
- From Apple's iWork suite
- Document processor (like Microsoft Word)

One simple task (from user's perspective):

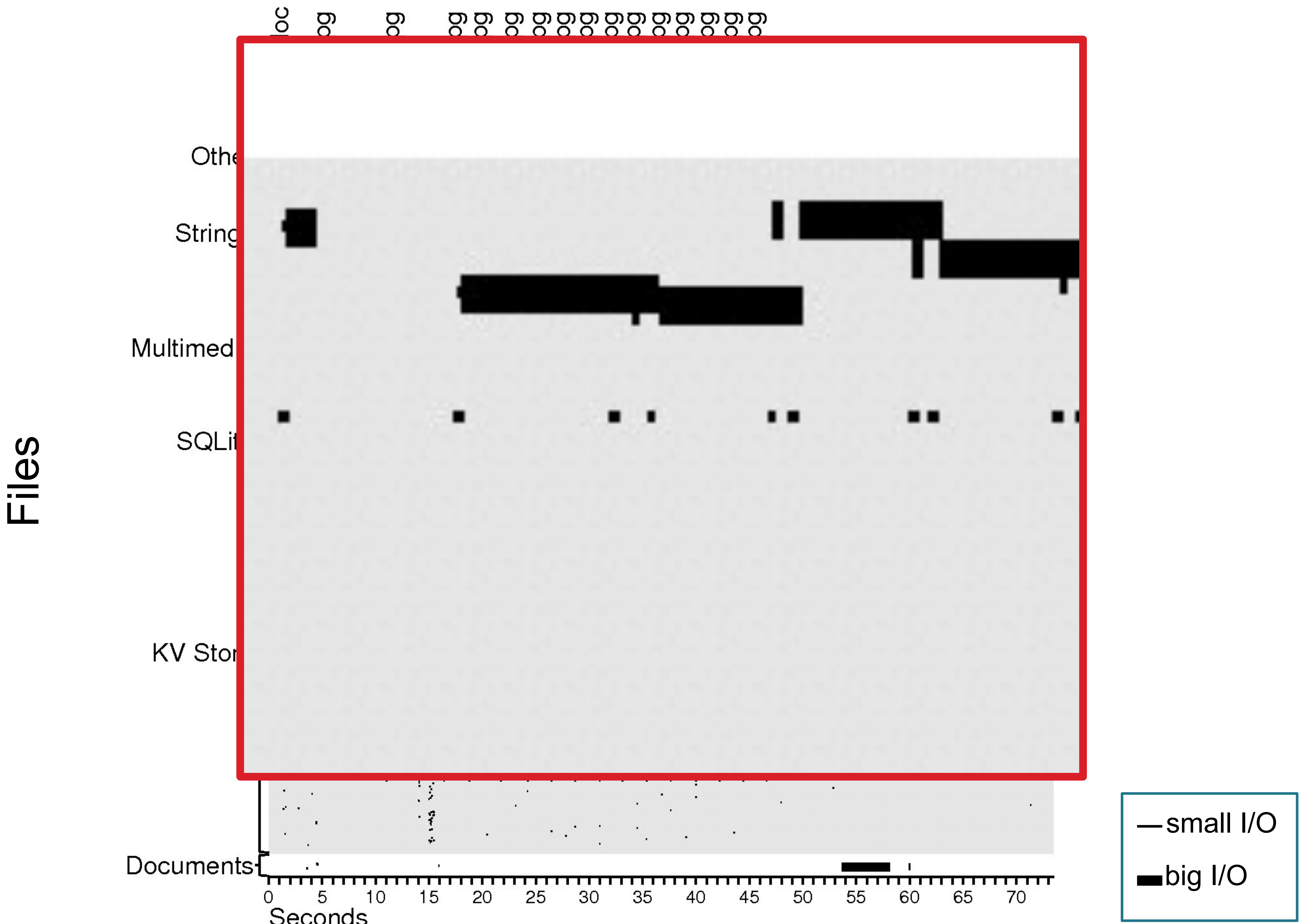
1. Create a new document
2. Insert 15 JPEG images (each ~2.5MB)
3. Save to the Microsoft DOC format

Trace I/O System Calls

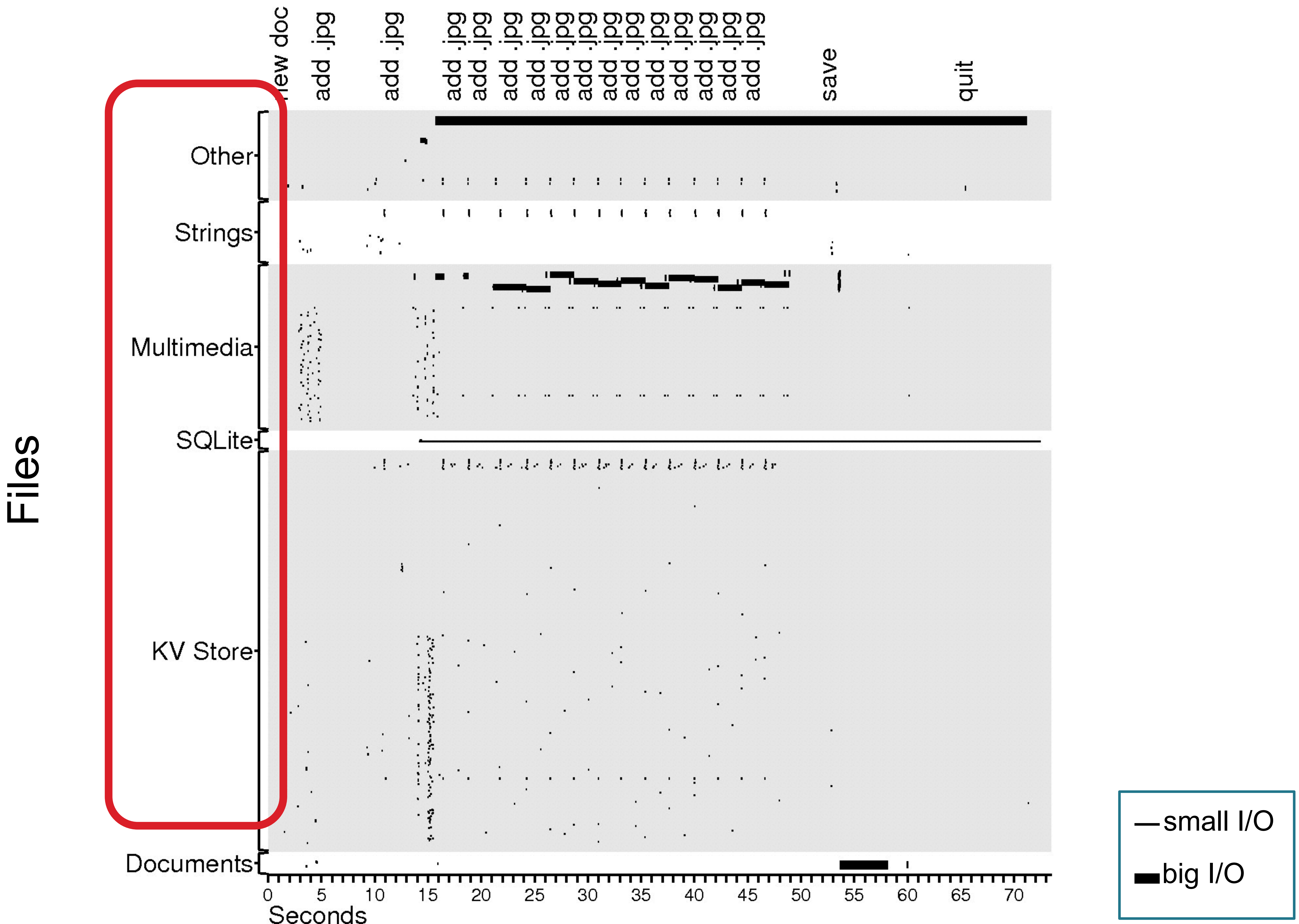
- Instrument with DTrace, record user-space stack traces
- Relatively little paging from mmap I/O



new doc
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
save
quit
Other
Strings
Multimedia
SQLite
KV Store
Documents
Files
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
Seconds
small I/O
big I/O

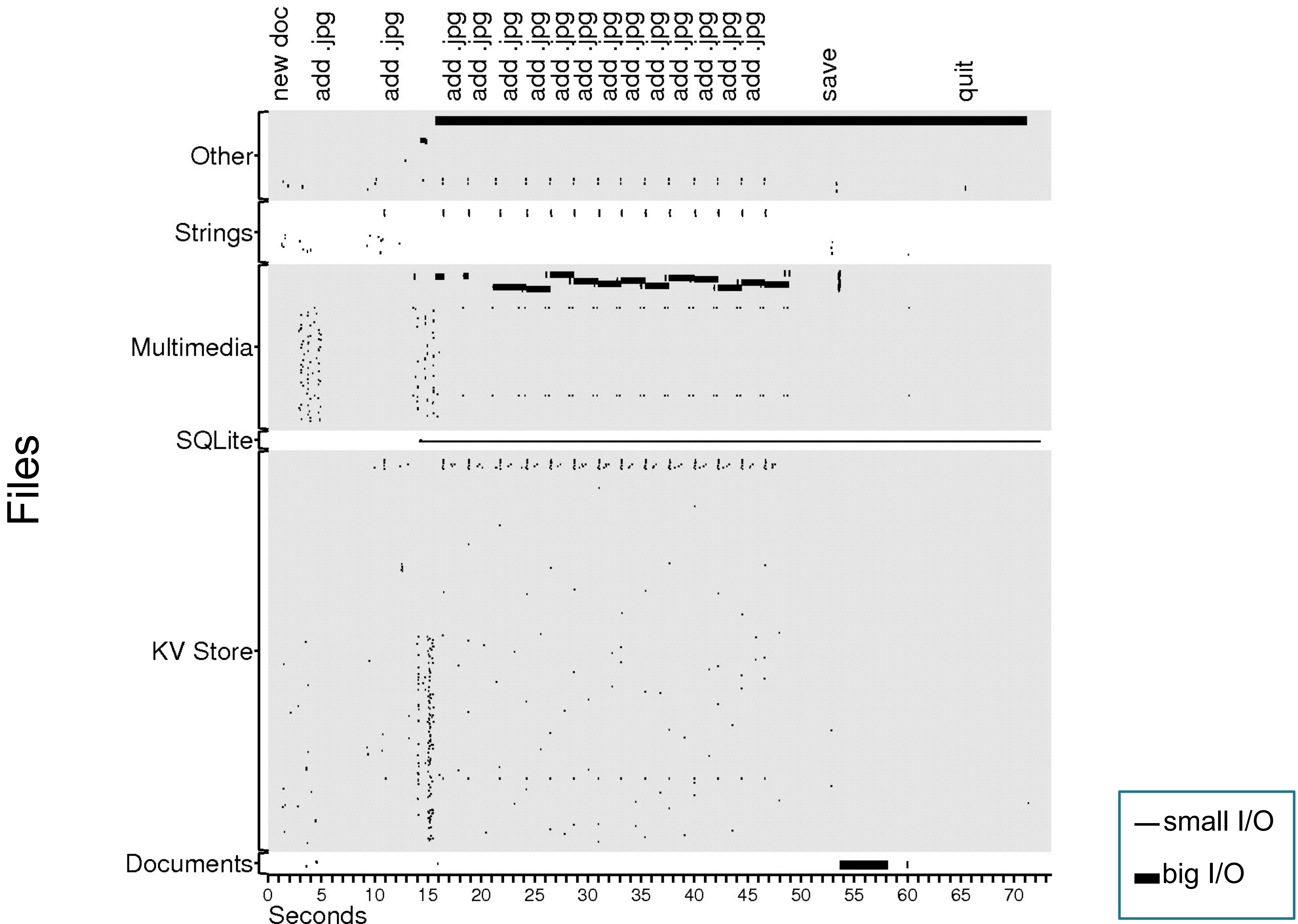


Files
Othe
String
Multimed
SQLit
KV Stor
Documents
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70
Seconds
small I/O
big I/O

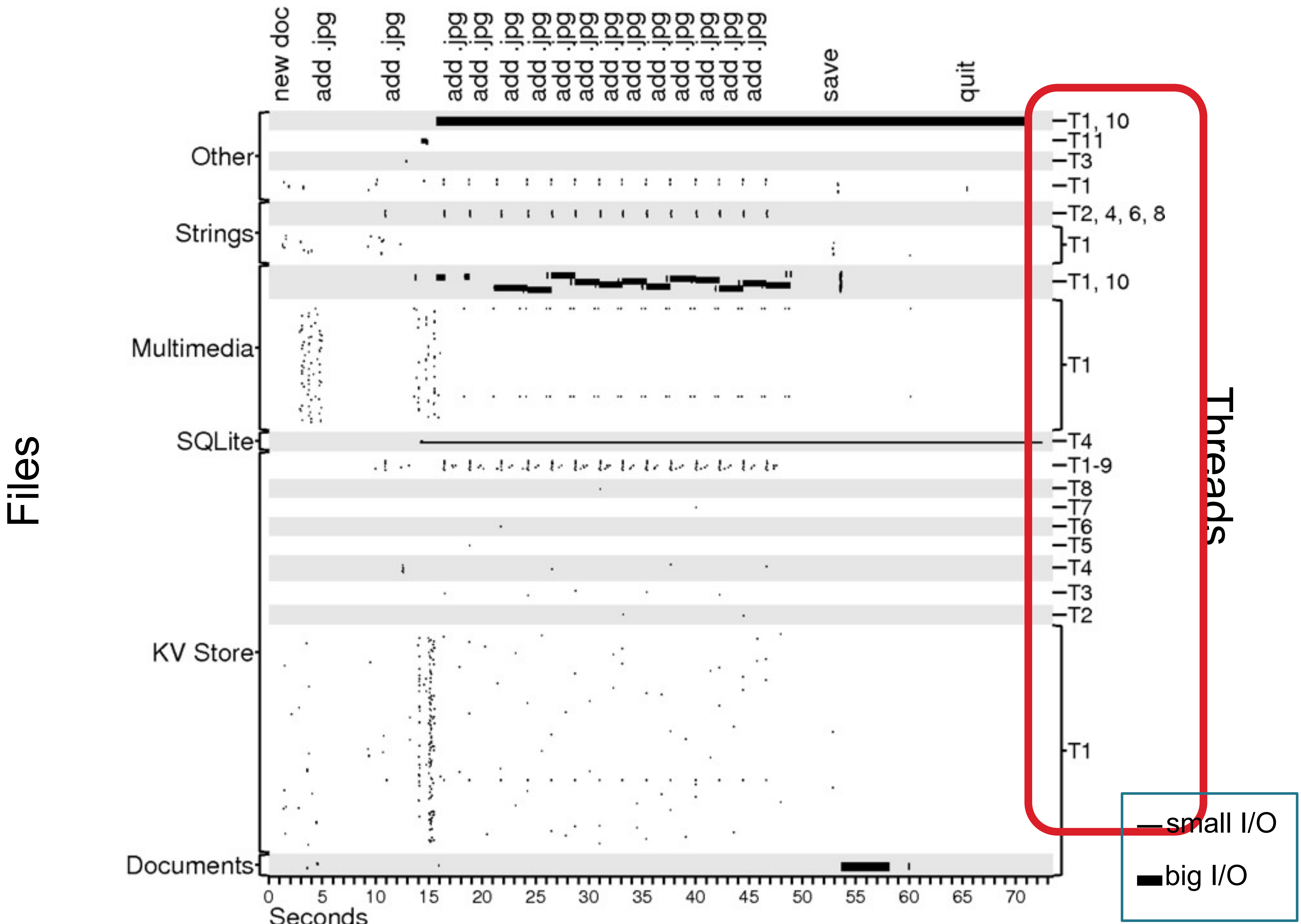

new doc
add .jpg
save
quit
Other
Strings
Multimedia
SQLite
KV Store
Documents
Files
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70
Seconds
small I/O
big I/O

Case Study Observations

- **Auxiliary files dominate**
  - Task's purpose: create **1** file; observed I/O: **385** files are touched
  - 218 KV store files + 2 SQLite files:
    - Personalized behavior (recently used lists, settings, etc)
  - 118 multimedia files:
    - Rich graphical experience
  - 25 Strings files:
    - Language localization
  - 17 Other files:
    - Auto-save file and others

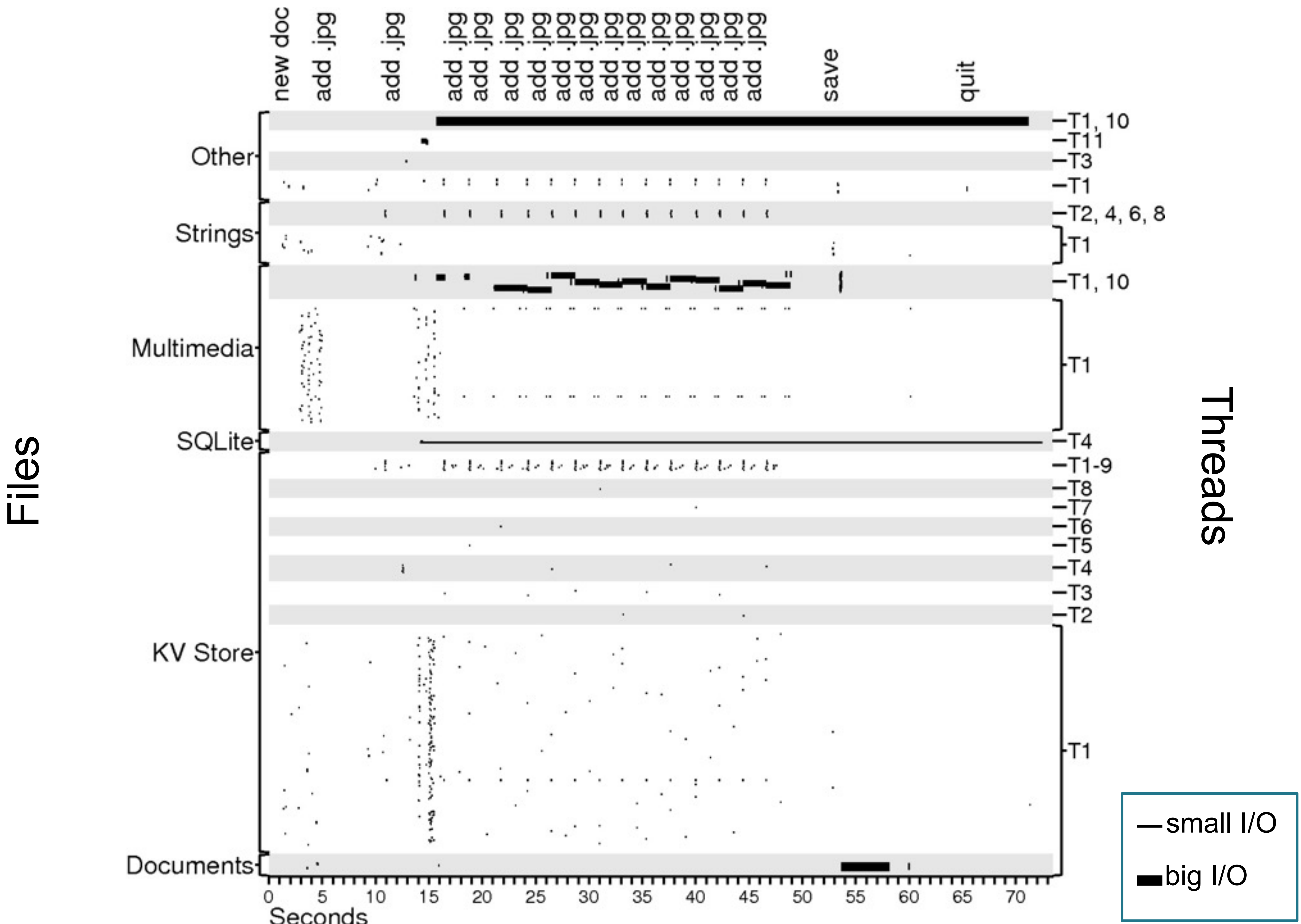


new doc
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
add .jpg
save
quit
Files
Other
Strings
Multimedia
SQLite
KV Store
Documents
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
Seconds
small I/O
big I/O

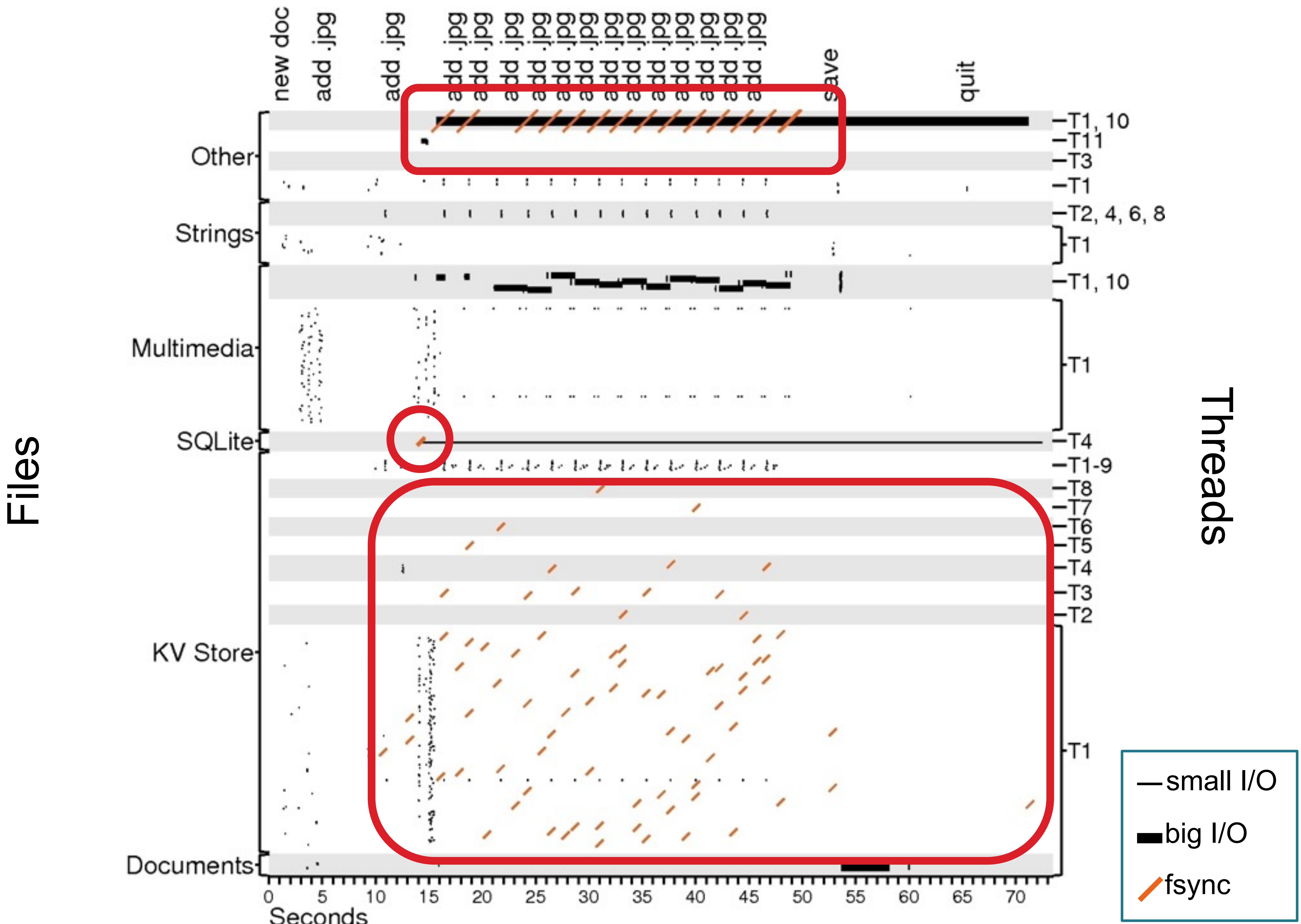

new doc
add .jpg
add .jpg
add .jpg
save
quit
Files
Other
Strings
Multimedia
SQLite
KV Store
Documents
Threads
T1, 10
T11
T3
T1
T2, 4, 6, 8
T1
T1, 10
T1
T4
T1-9
T8
T7
T6
T5
T4
T3
T2
T1
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
Seconds
small I/O
big I/O

Case Study Observations

- Auxiliary files dominate
- **Multiple threads perform I/O**
  - Interactive programs must avoid blocking



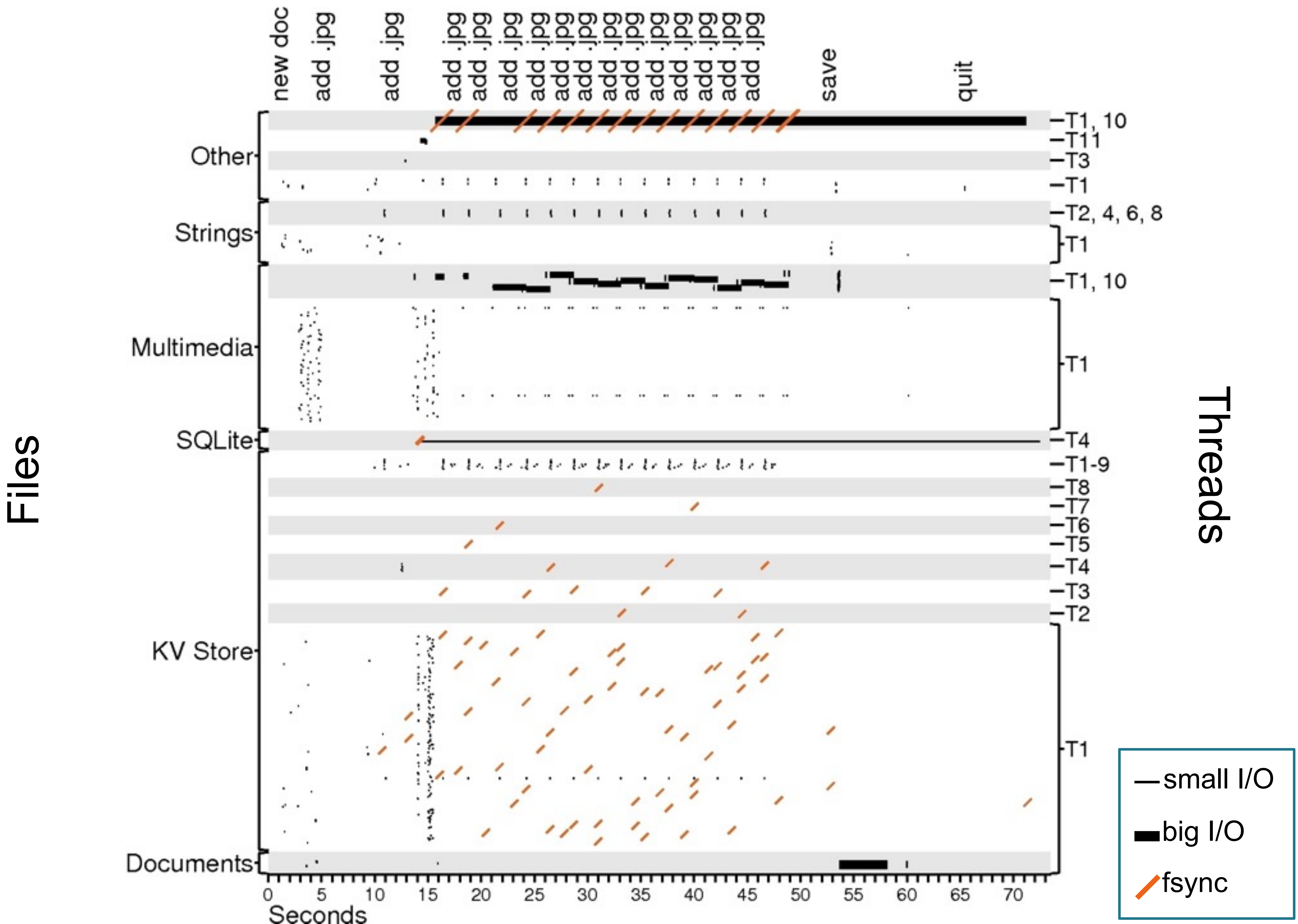
new doc
add .jpg
add .jpg
add .jpg
save
quit
Other
Strings
Multimedia
SQLite
KV Store
Documents
Files
Threads
T1, 10
T11
T3
T1
T2, 4, 6, 8
T1
T1, 10
T1
T4
T1-9
T8
T7
T6
T5
T4
T3
T2
T1
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
Seconds
small I/O
big I/O



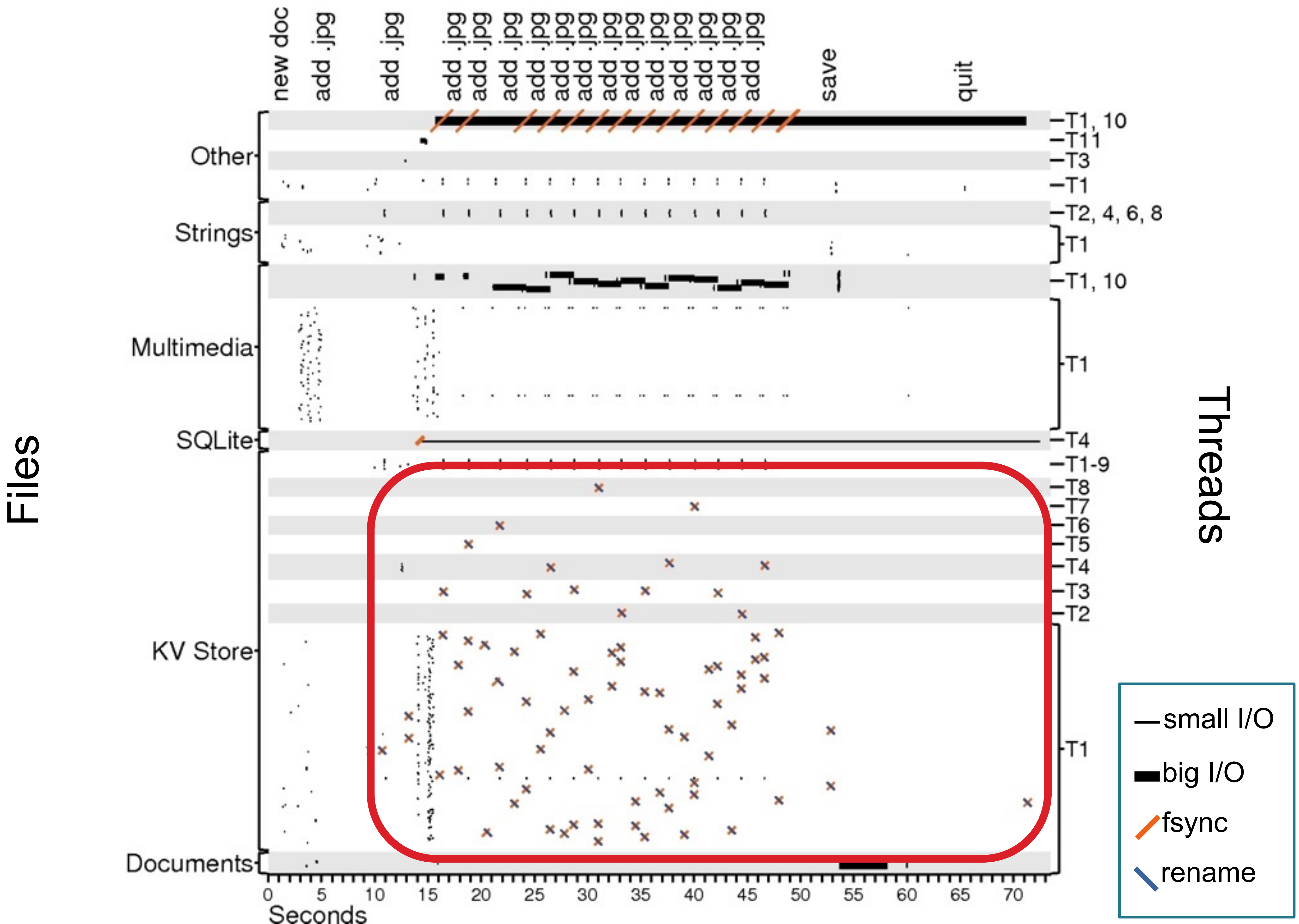
new doc
add .jpg
add .jpg
add .jpg
save
quit
Other
Strings
Multimedia
SQLite
KV Store
Documents
Files
Threads
T1, 10
T11
T3
T1
T2, 4, 6, 8
T1
T1, 10
T1
T4
T1-9
T8
T7
T6
T5
T4
T3
T2
T1
Seconds
small I/O
big I/O
fsync

Case Study Observations

- Auxiliary files dominate
- Multiple threads perform I/O
- **Writes are often forced**
  - KV-store + SQLite durability
  - Auto-save file



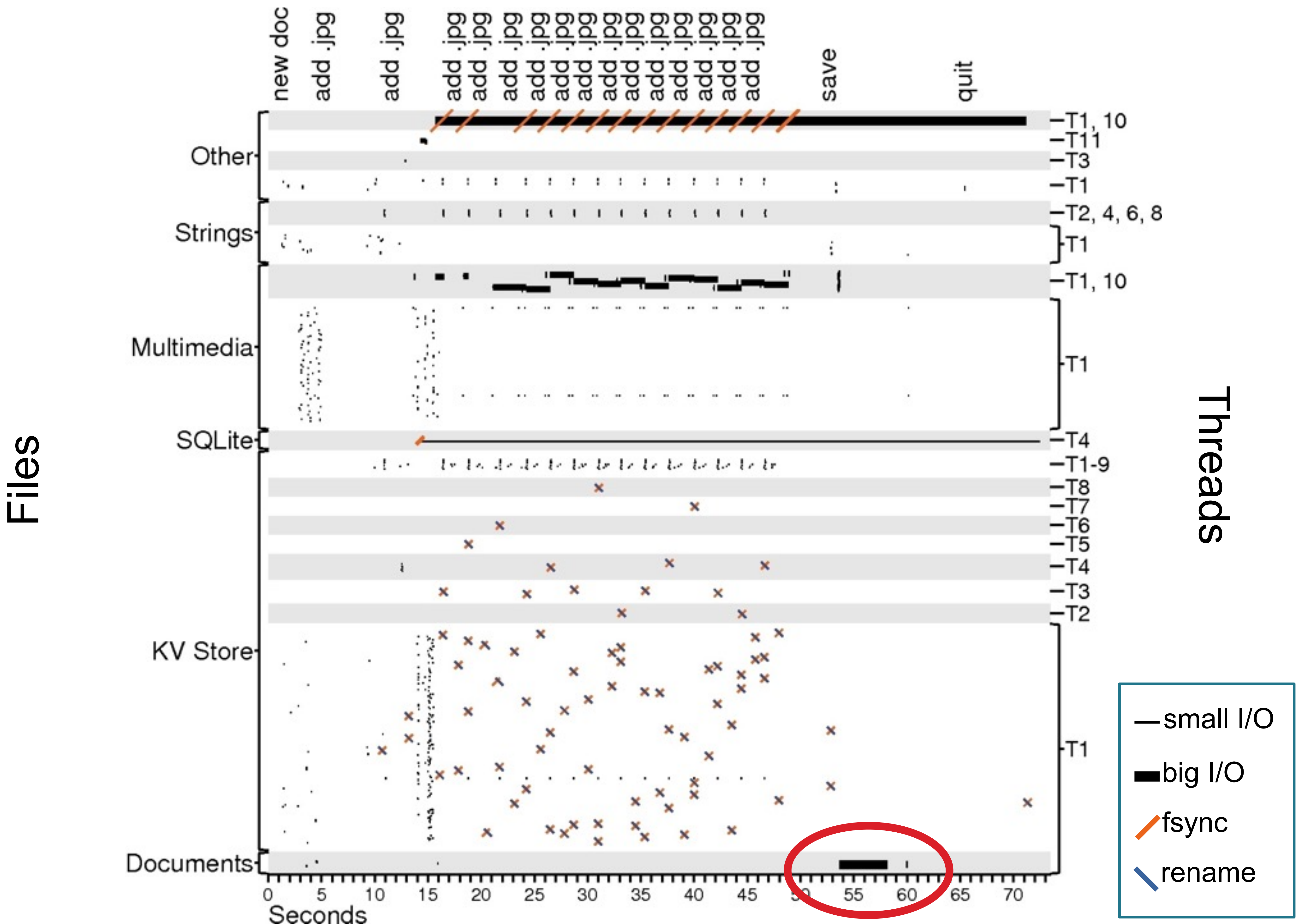
new doc
add .jpg
save
quit
Other
Strings
Multimedia
SQLite
KV Store
Documents
Files
Threads
T1, 10
T11
T3
T1
T2, 4, 6, 8
T1-9
T8
T7
T6
T5
T4
T2
Seconds
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
small I/O
big I/O
fsync



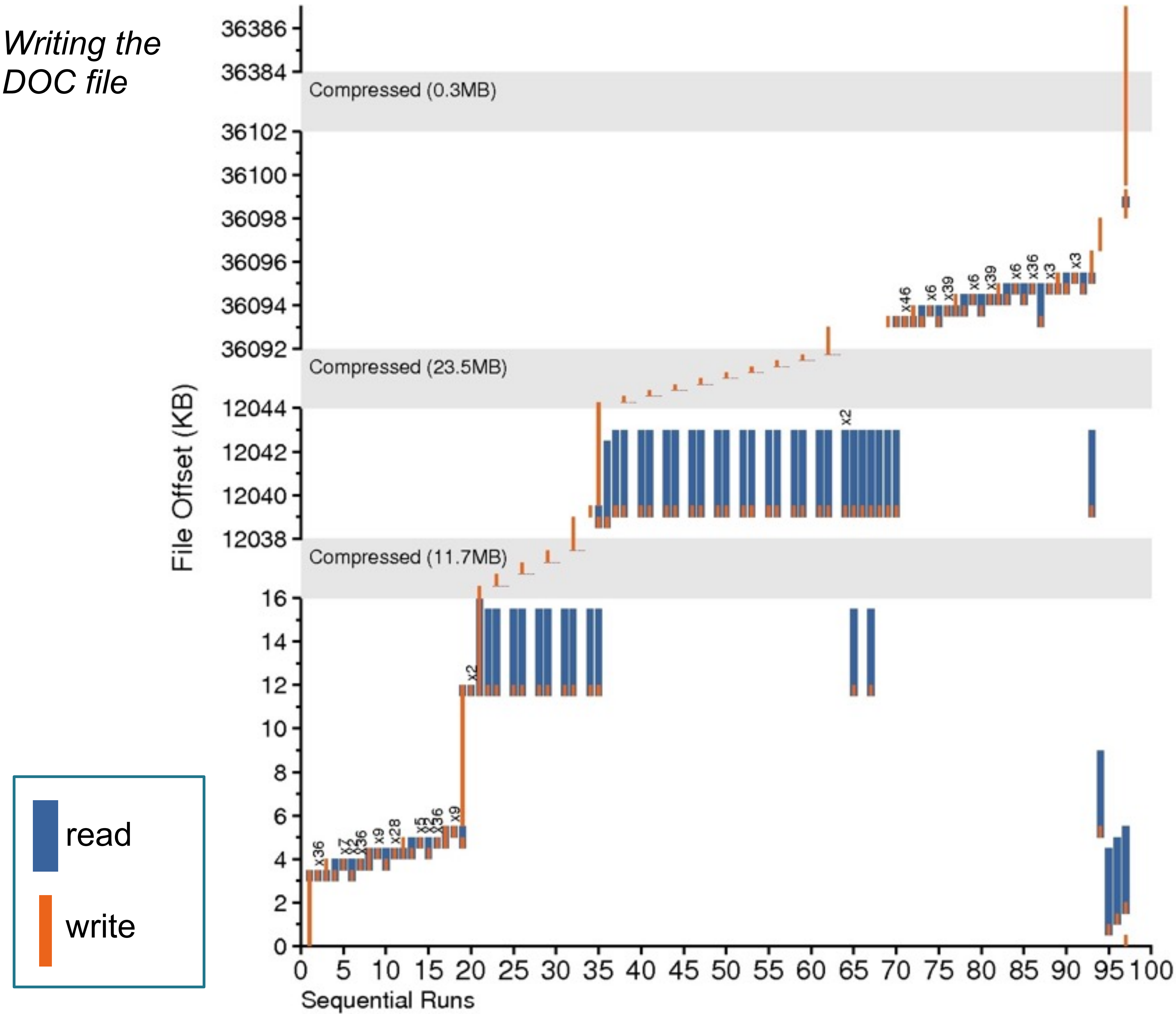
new doc
add .jpg
add .jpg
add .jpg
save
quit
Files
Other
Strings
Multimedia
SQLite
KV Store
Documents
Threads
T1, 10
T11
T3
T1
T2, 4, 6, 8
T1
T1, 10
T1
T4
T1-9
T8
T7
T6
T5
T4
T3
T2
T1
Seconds
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
small I/O
big I/O
fsync
rename

Case Study Observations

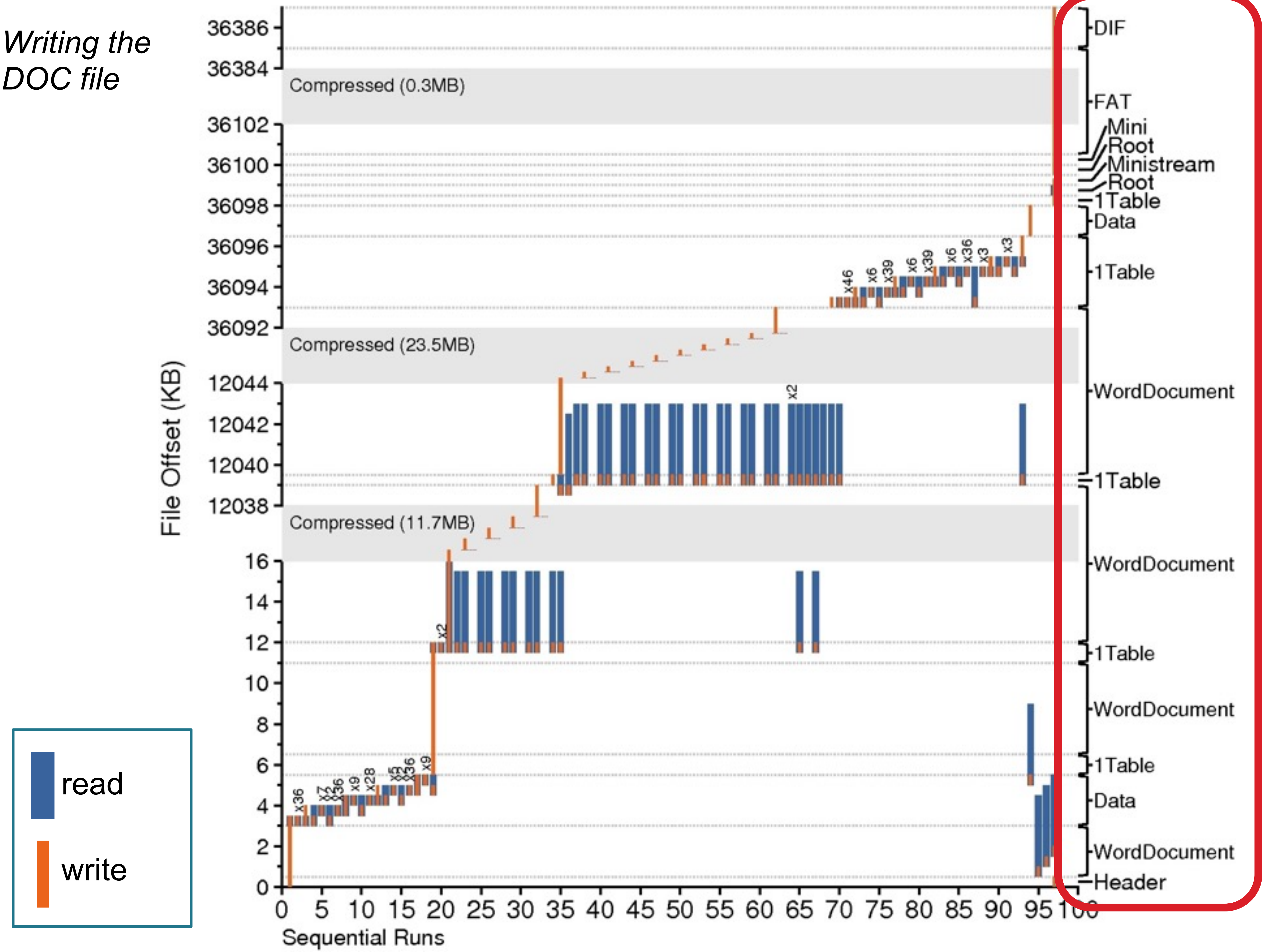
- Auxiliary files dominate
- Multiple threads perform I/O
- Writes are often forced
- **Renaming is popular**
  - Often used for key-value store
  - Makes updates atomic


new doc
add .jpg
add .jpg
add .jpg
save
quit
Other
Strings
Multimedia
SQLite
KV Store
Documents
Files
Threads
T1, 10
T11
T3
T1
T2, 4, 6, 8
T1
T1, 10
T1
T4
T1-9
T8
T7
T6
T5
T4
T3
T2
T1
Seconds
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
small I/O
big I/O
fsync
rename

Writing the DOC file



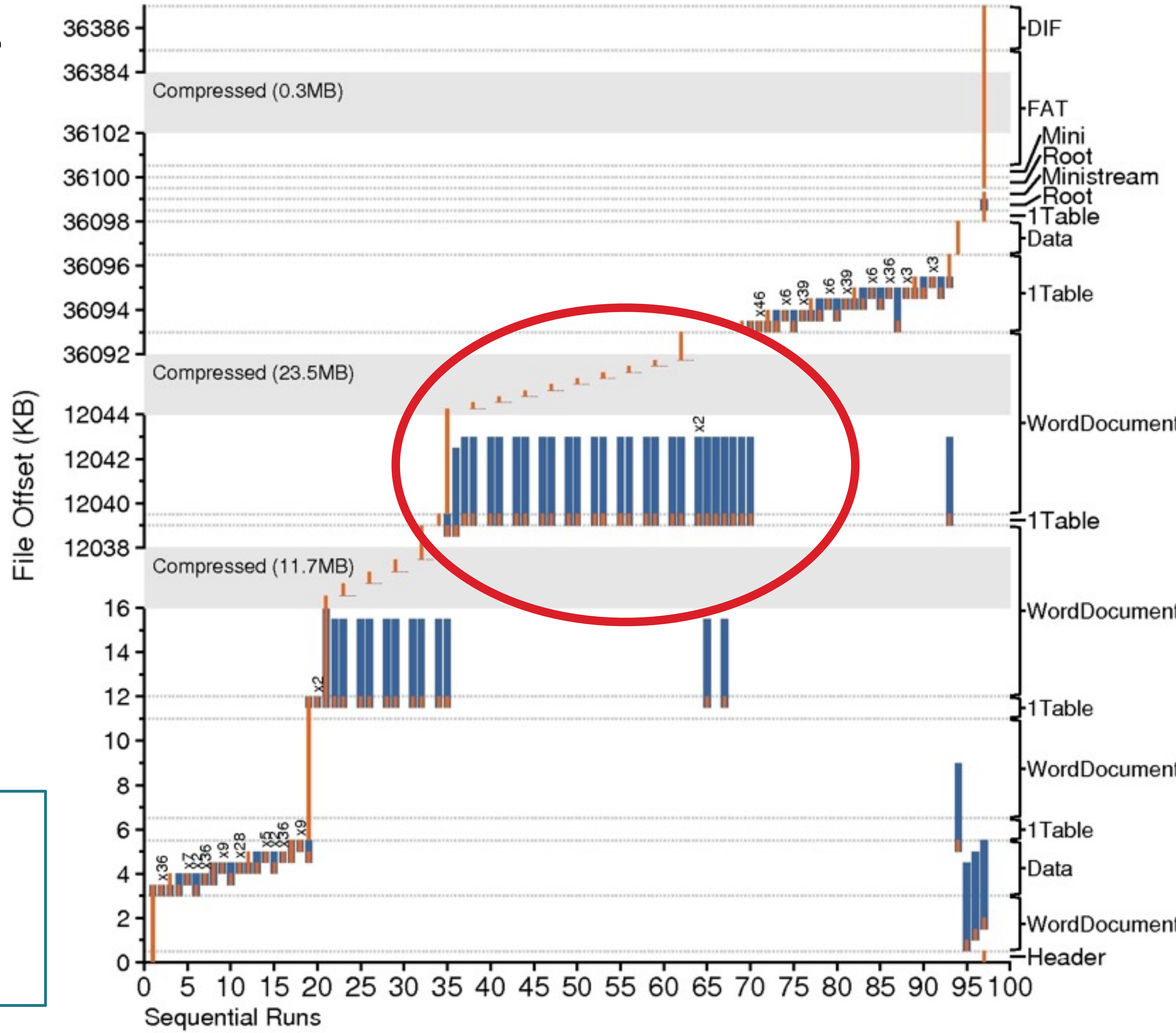
Writing the DOC file



Case Study Observations

- Auxiliary files dominate
- Multiple threads perform I/O
- Writes are often forced
- Renaming is popular
- **A file is not a file**
  - DOC format is modeled after a FAT file system
    - Multiple "sub-files"
    - Application manages space allocation

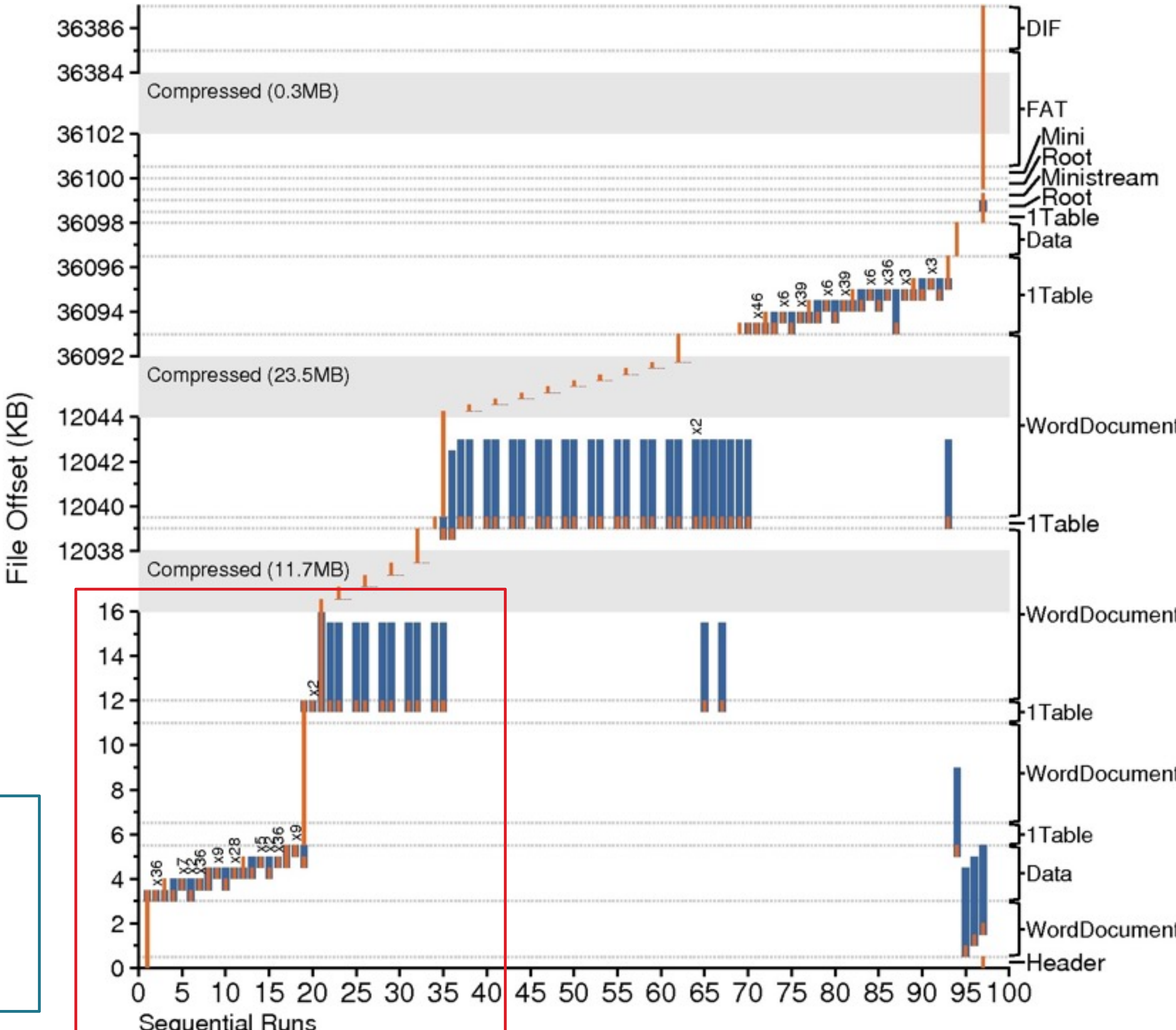
Writing the DOC file

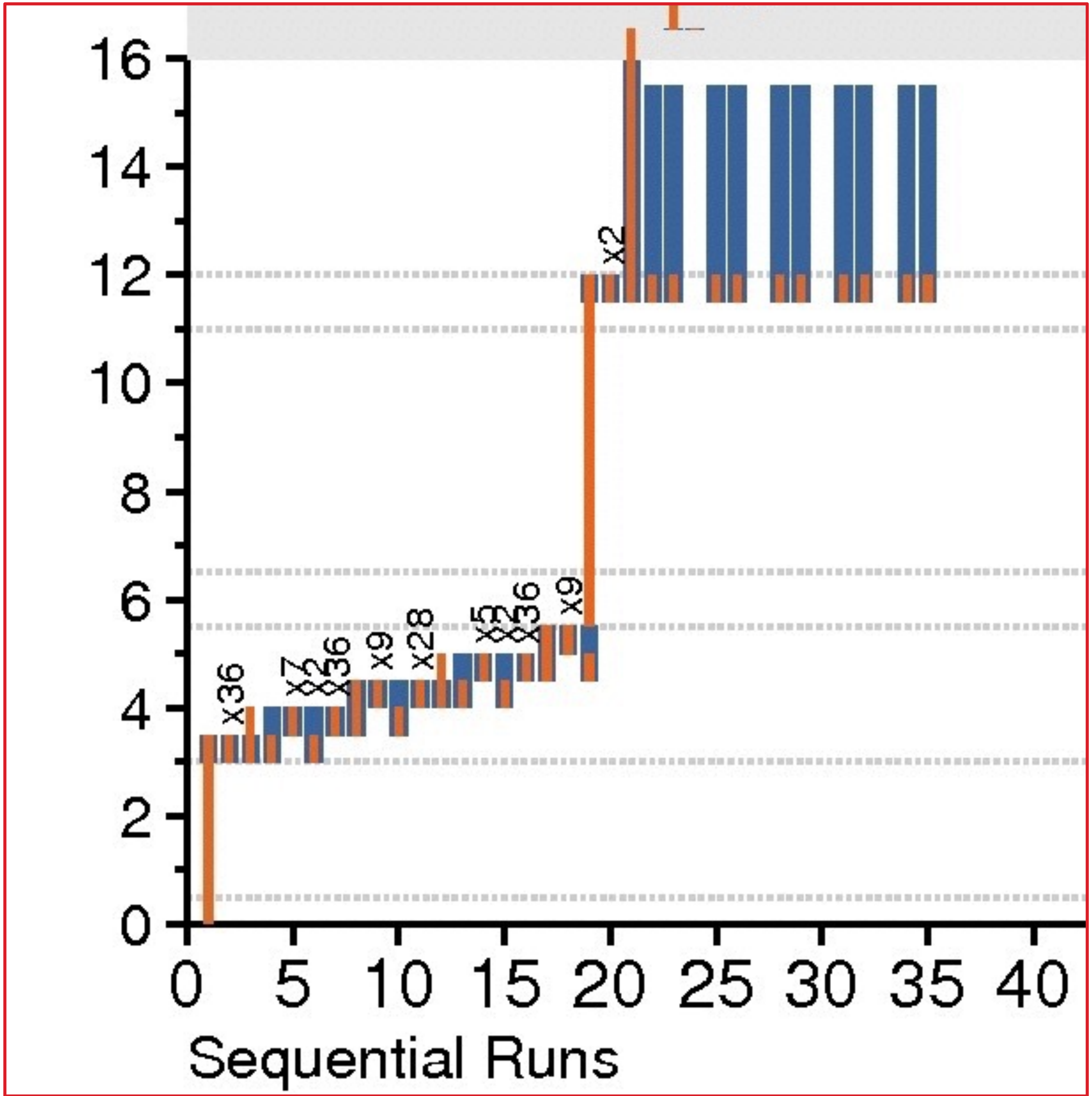


Case Study Observations

- Auxiliary files dominate
- Multiple threads perform I/O
- Writes are often forced
- Renaming is popular
- A file is not a file
- **Sequential access is not sequential**
  - Multiple sequential runs in a complex file => random accesses

Writing the DOC file





16
14
12
10
8
6
4
2
0
x36
x7
x2
x36
x9
x28
x5
x2
x36
x9
x2
0
5
10
15
20
25
30
35
40
Sequential Runs

Case Study Observations

- Auxiliary files dominate
- Multiple threads perform I/O
- Writes are often forced
- Renaming is popular
- A file is not a file
- Sequential access is not sequential
- **Frameworks influence I/O**
  - Example: update value in page function
  - Cocoa, Carbon are a substantial part of application

Case Study Observations

- Auxiliary files dominate
- Multiple threads perform I/O
- Writes are often forced
- Renaming is popular
- A file is not a file
- Sequential access is not sequential
- Frameworks influence I/O

all findings are general trends across multiple tasks
(more details in dissertation)

Case Study Observations

- Auxiliary files dominate
- Multiple threads perform I/O
- Writes are often forced
- Renaming is popular
- A file is not a file
- Sequential access is not sequential
- Frameworks influence I/O

all findings are general trends across multiple tasks
(more details in dissertation)

Noted Effects of Modularity

Described in dissertation:

- Mismatch between .doc page size and STDIO block size
- Repeated read-copy-update to same page
- Open flags are meaningless (O_RDWR overused)
- Preallocation hints not meaningful
- Copy abstraction prevents combined use of source
- Coarse-grained exclusion make fine-grained locks useless
- Atomicity/durability required for unimportant data

Noted Effects of Modularity

Described in dissertation:

- Mismatch between .doc page size and STDIO block size
- Repeated read-copy-update to same page
- Open flags are meaningless (O_RDWR overused)
- Preallocation hints not meaningful
- Copy abstraction prevents combined use of source
- Coarse-grained exclusion make fine-grained locks useless
- Atomicity/durability required for unimportant data

Use of Fsync

Older studies

- Baker *et al.*: 16% of data flushed by app. request (1991)
- Vogels: *"In 1.4% of file opens that had write operations posted to them, caching was disabled at open time. Of the files that were opened with write caching enabled, 4% actively controlled their caching by using the flush requests."* (1999)

Newer study

- Kim *et al.*: SQLite write traffic itself is quite random with plenty of synchronous overwrites … apps use the Android interfaces oblivious to performance. A particularly striking example is the heavy-handed management of application caches through SQLite." (2012)

Outline

Motivation: Modularity in Modern Storage

Overview: Types of Modularity

Library Study: Apple Desktop Applications

Layer Study: Facebook Messages

Microservice Study: Docker Containers

Slacker: a Lazy Docker Storage Driver

Conclusions

Why Study Facebook Messages?

Represents an important type of application. Universal backend for:

- Cellphone texts
- Chats
- Emails

Why Study Facebook Messages?

Represents an important type of application. Universal backend for:

- Cellphone texts
- Chats
- Emails

Why Study Facebook Messages?

Represents an important type of application. Universal backend for:

- Cellphone texts
- Chats
- Emails

Why Study Facebook Messages?

Represents an important type of application. Universal backend for:

- Cellphone texts
- Chats
- Emails

Why Study Facebook Messages?

Represents an important type of application. Universal backend for:

- Cellphone texts
- Chats
- Emails

Represents HBase over HDFS

- Common backend at Facebook and other companies
- Similar stack used at Google (BigTable over GFS)

Why Study Facebook Messages?

Represents an important type of application. Universal backend for:

- Cellphone texts
- Chats
- Emails

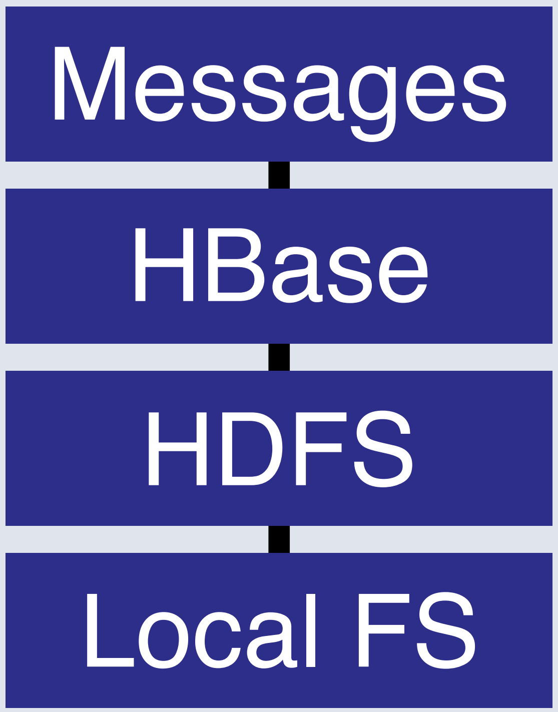
Represents HBase over HDFS

- Common backend at Facebook and other companies
- Similar stack used at Google (BigTable over GFS)

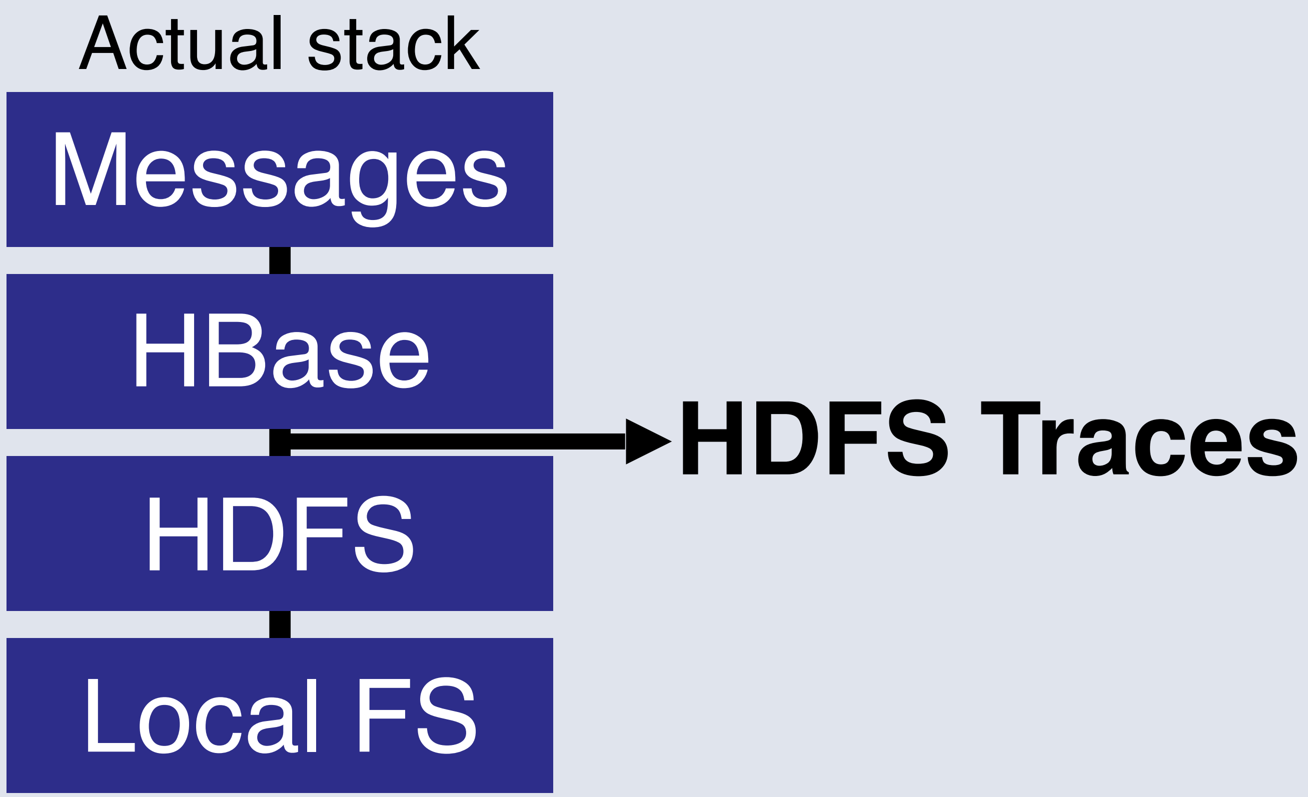
Represents layered storage

Methodology

Actual stack



Methodology



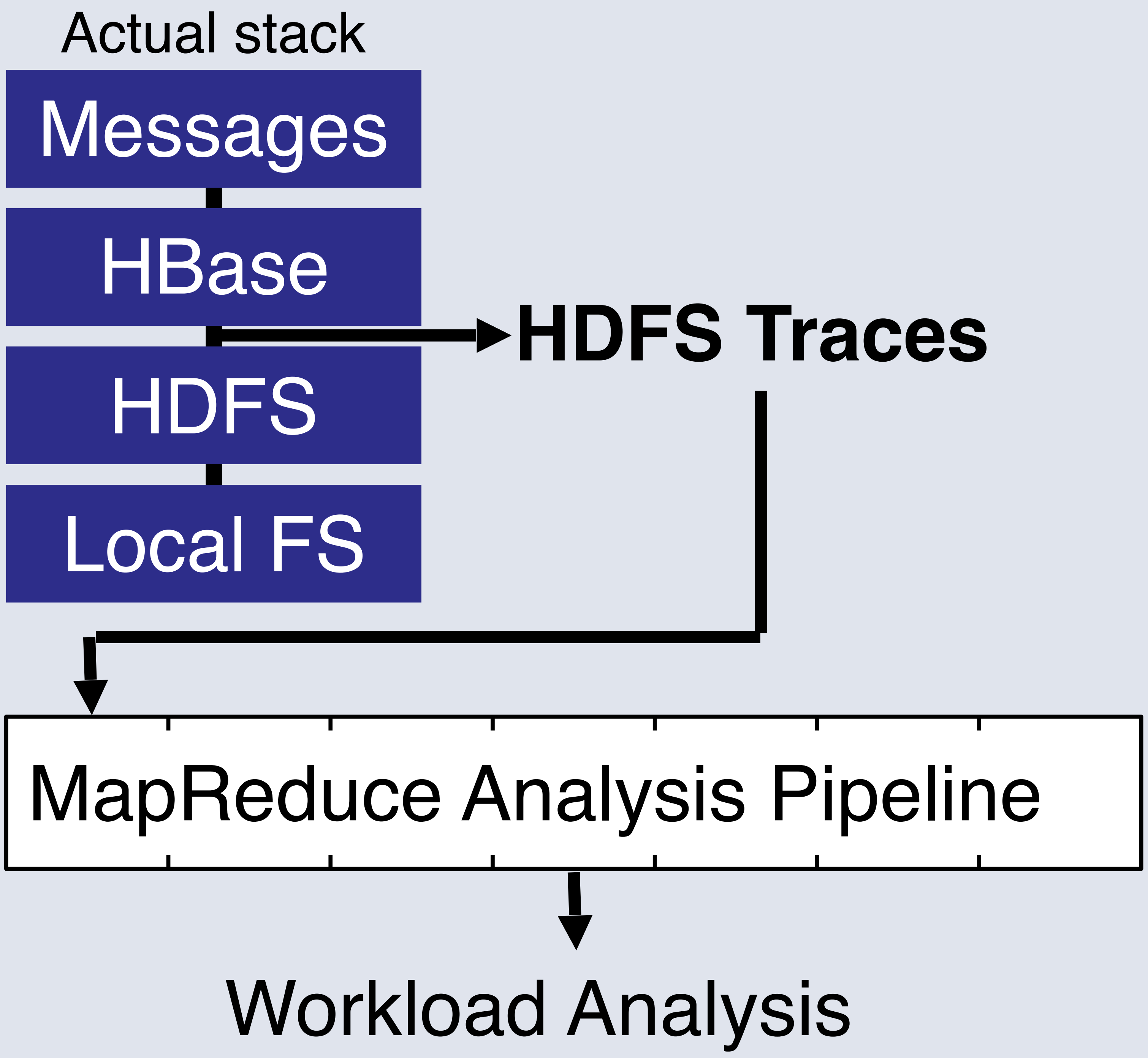
New tracing layer

- Hadoop Trace FS (HTFS)
- Collects request details
  - Reads/writes, offsets, sizes
  - Not contents

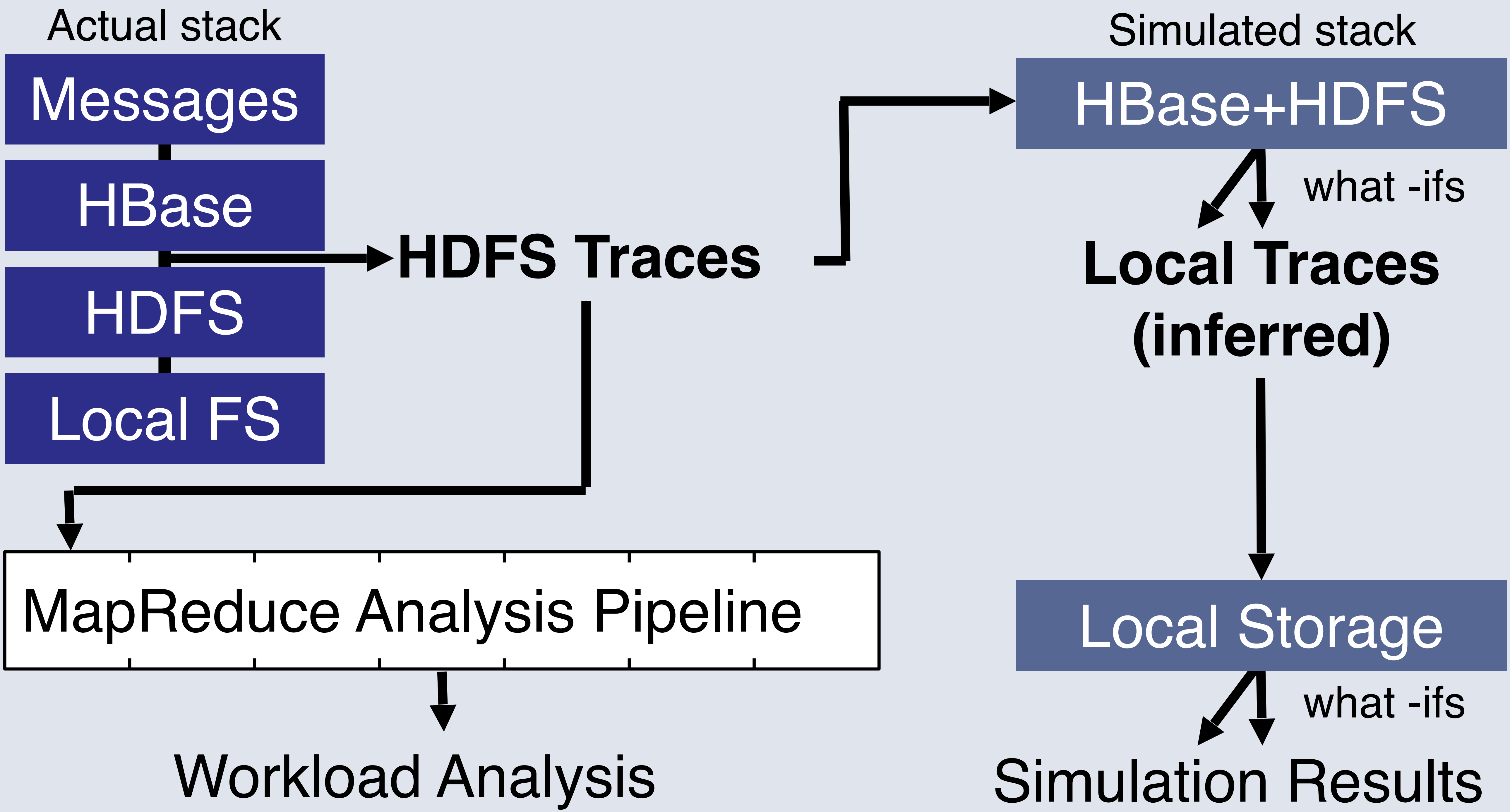
Trace results

- 9 shadow machines
- Production requests mirrored
- 8.3 days
- 71TB of HDFS I/O

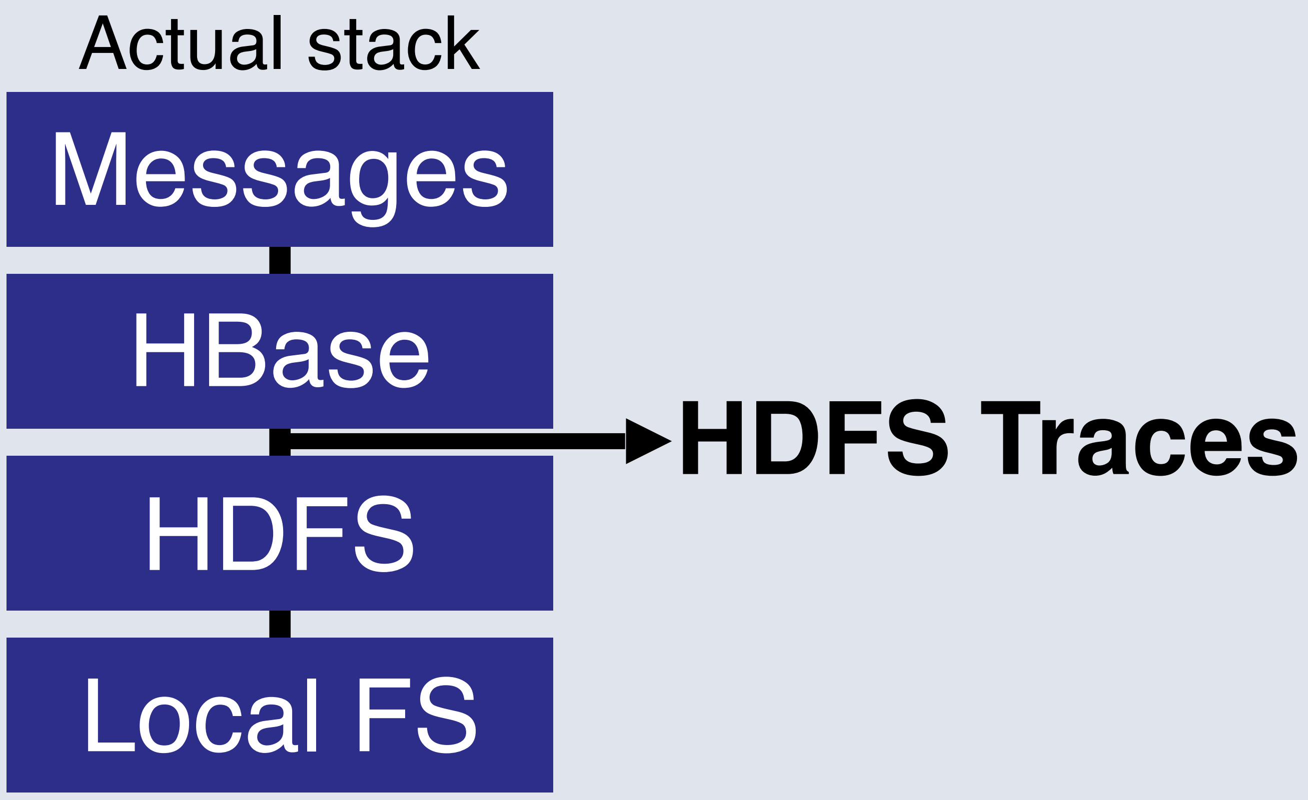
Methodology



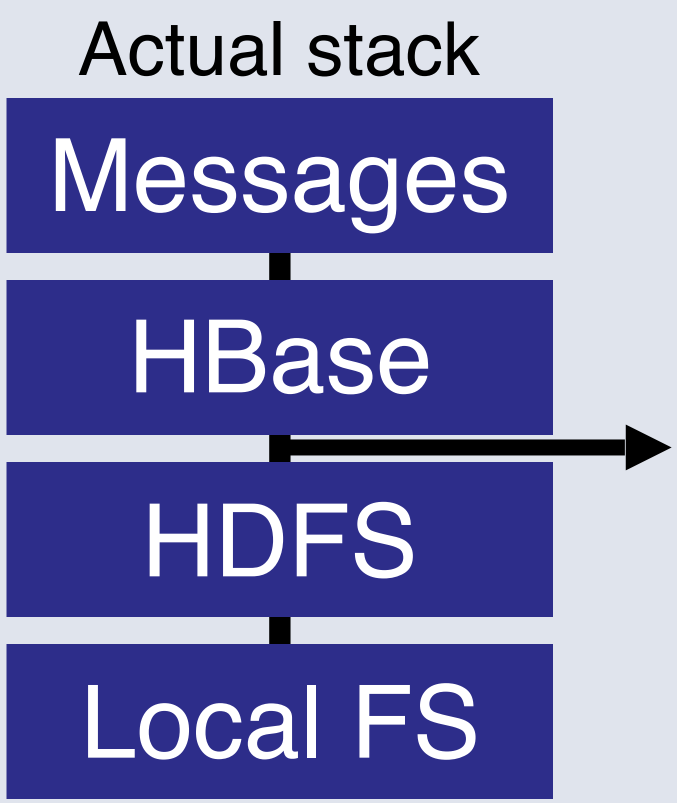
Methodology



Methodology



Methodology

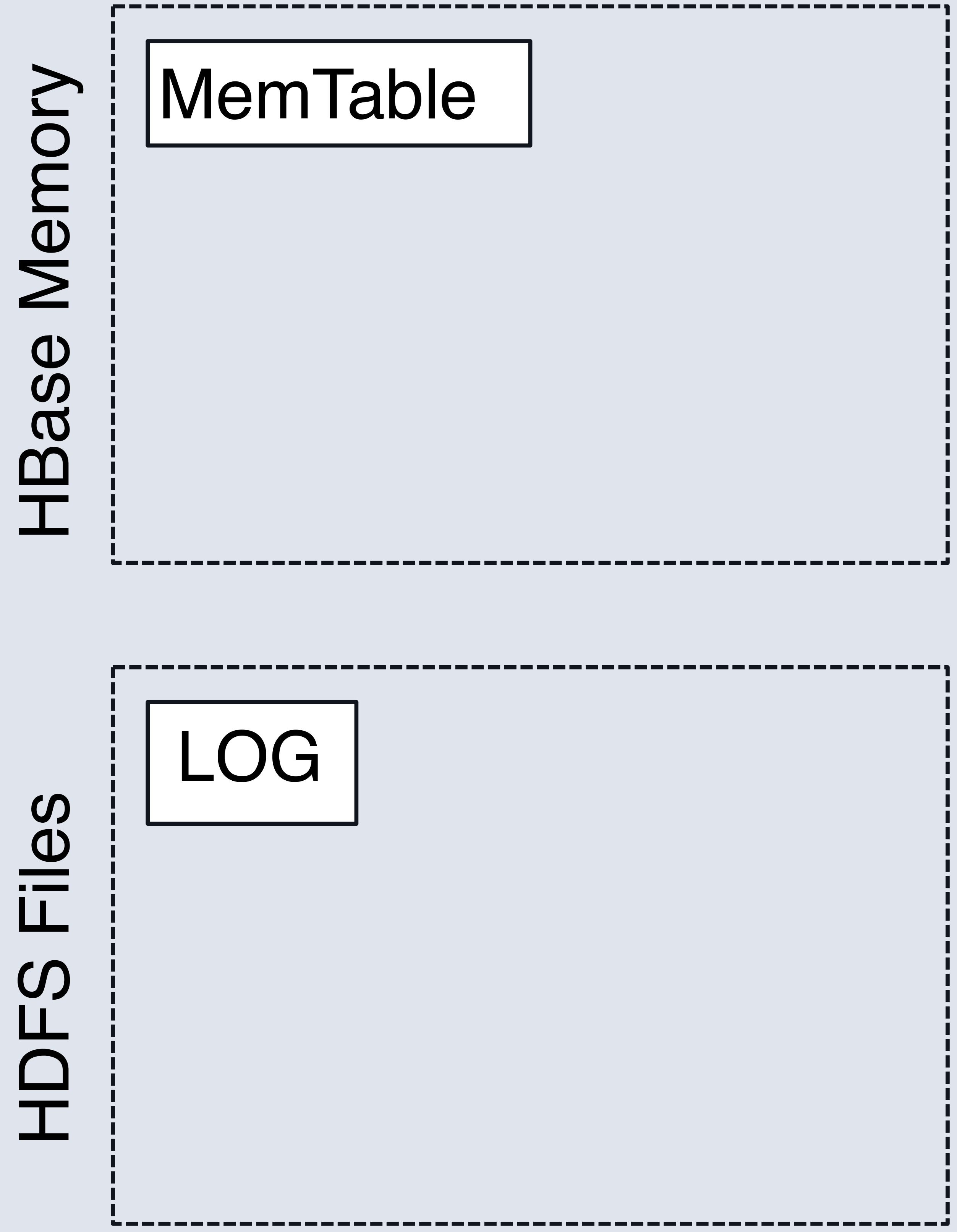


HDFS Traces

Background: how does HBase use HDFS?

HBase's HDFS Files

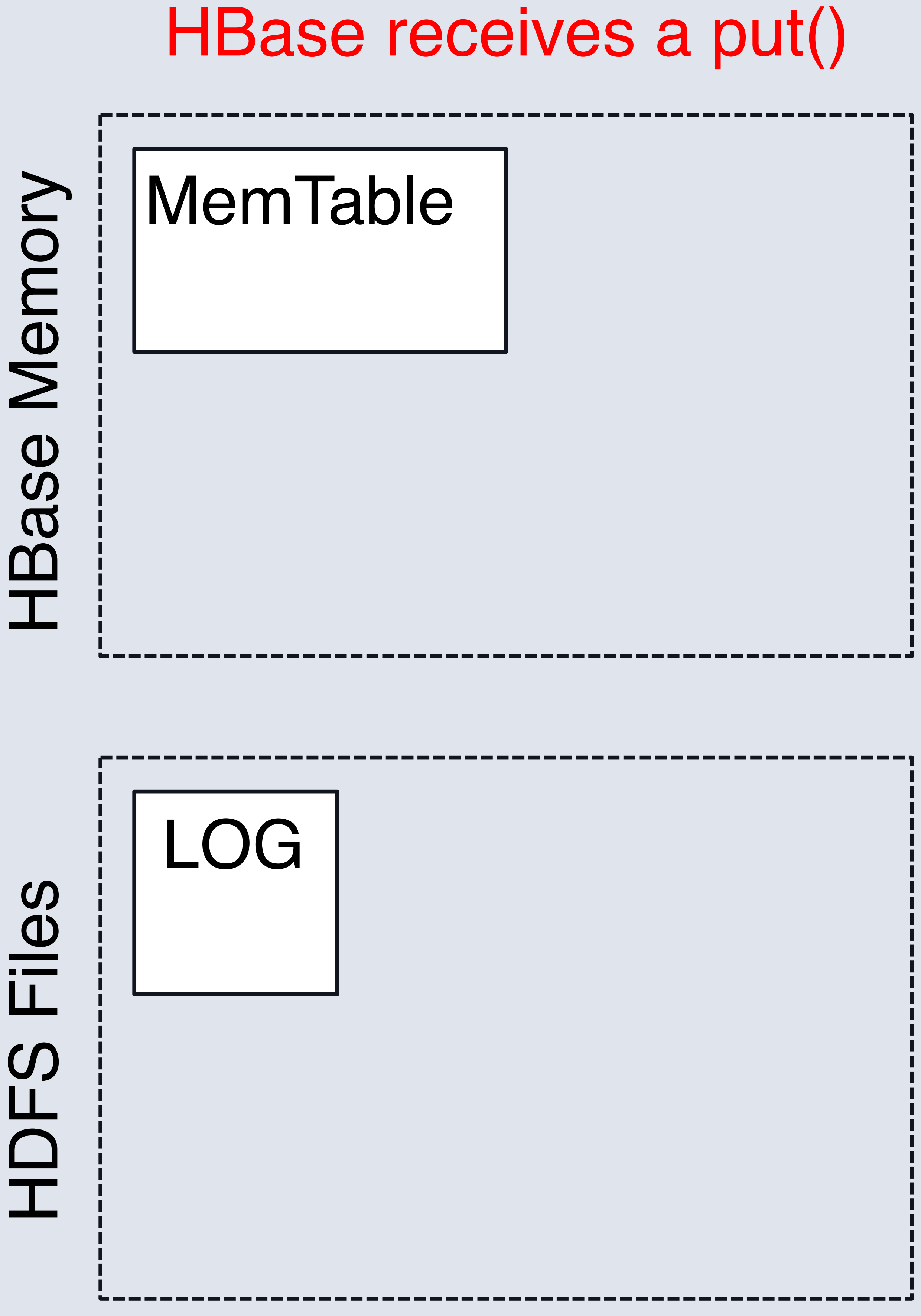
Four activities do HDFS I/O:



HBase's HDFS Files

Four activities do HDFS I/O:

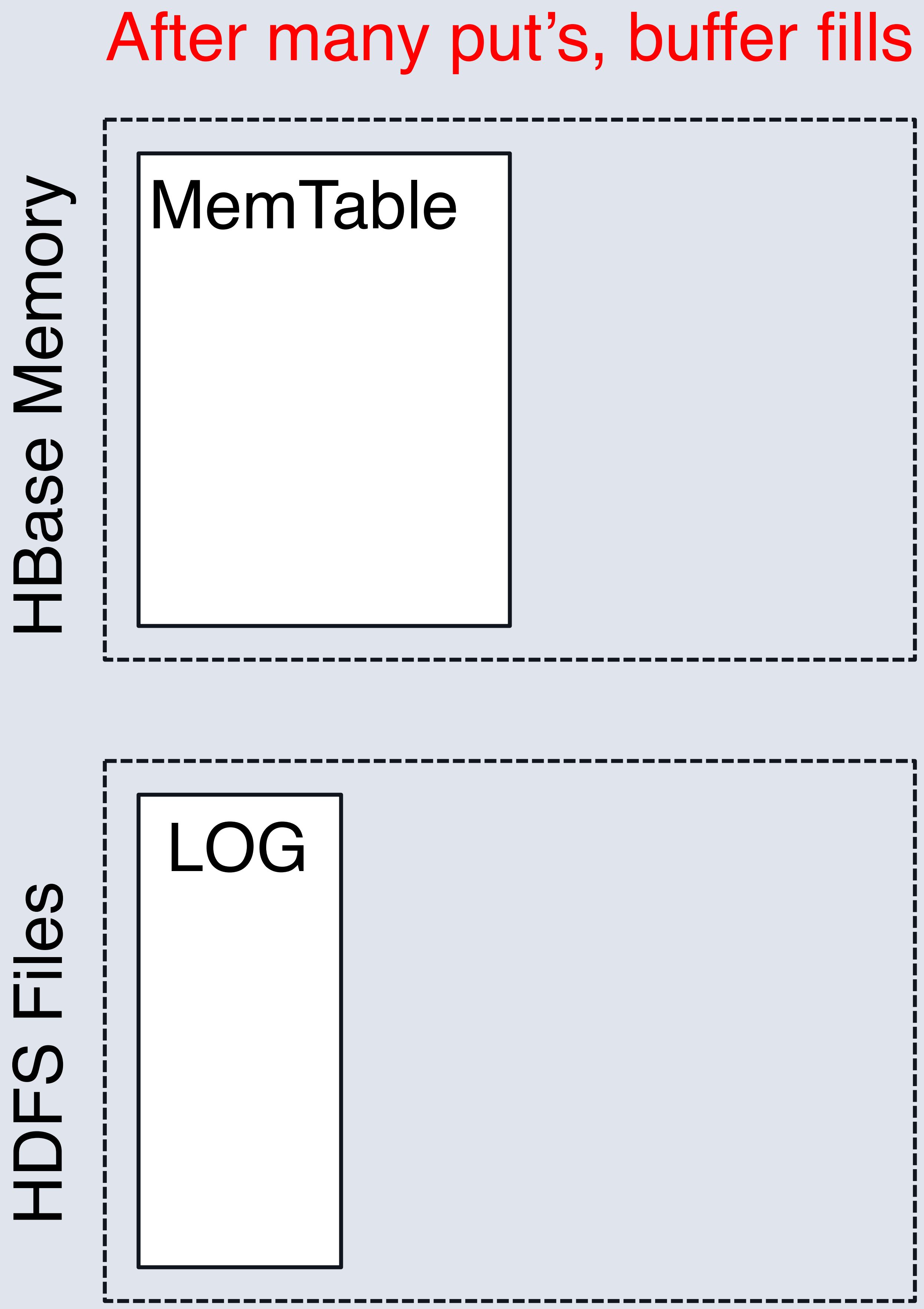
- Logging



HBase's HDFS Files

Four activities do HDFS I/O:

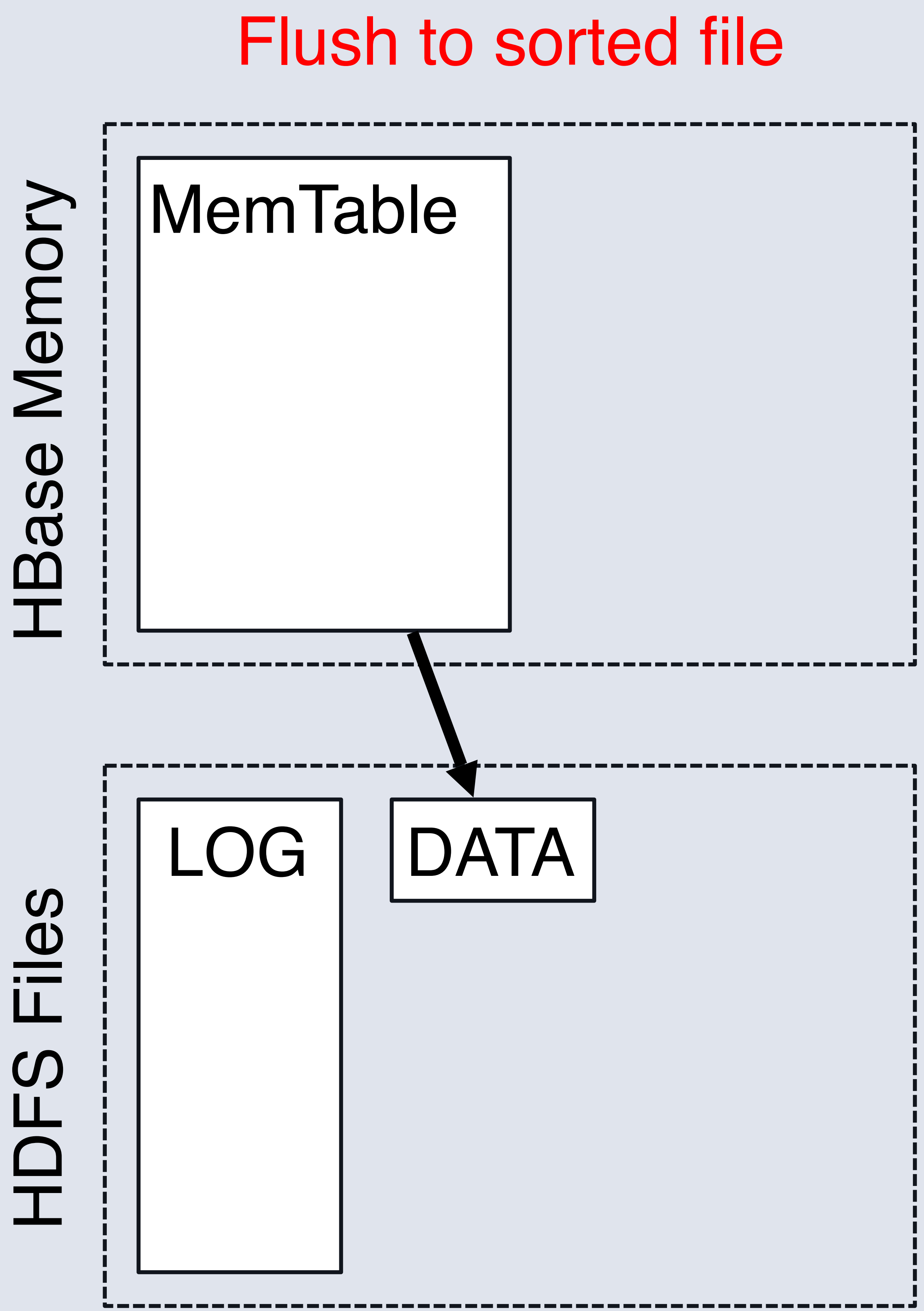
- Logging



HBase's HDFS Files

Four activities do HDFS I/O:

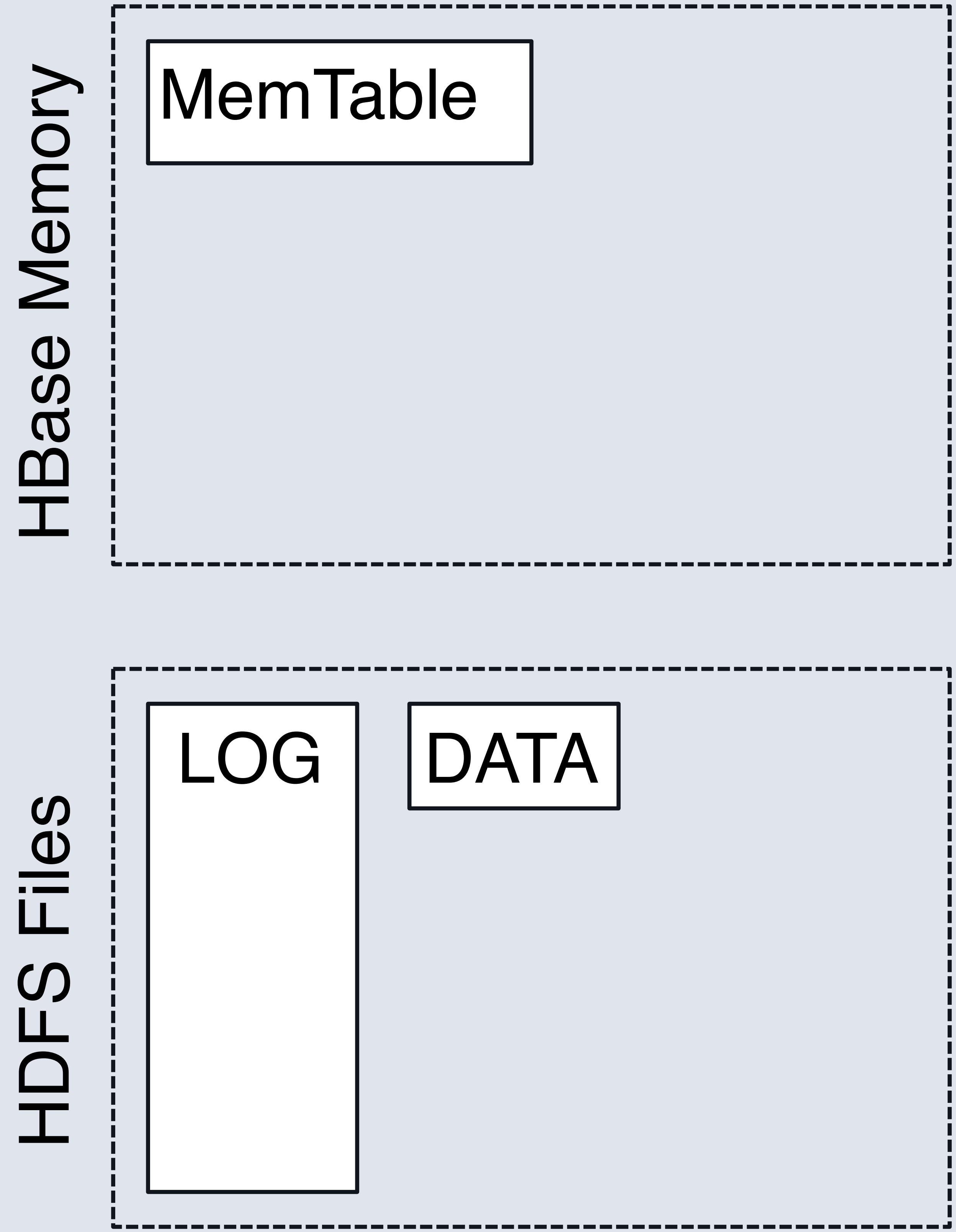
- Logging
- Flushing



HBase's HDFS Files

Four activities do HDFS I/O:

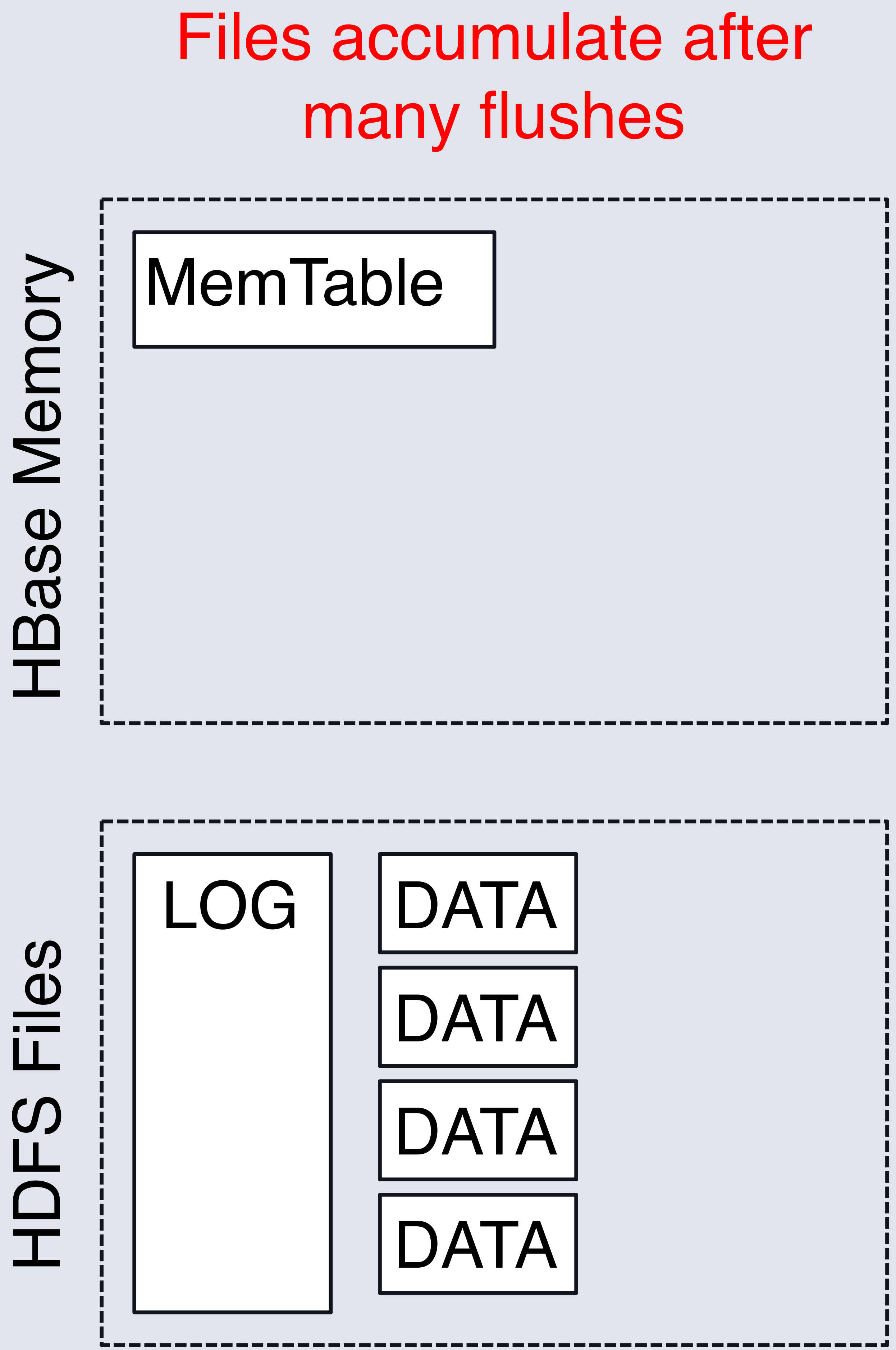
- Logging
- Flushing



HBase's HDFS Files

Four activities do HDFS I/O:

- Logging
- Flushing

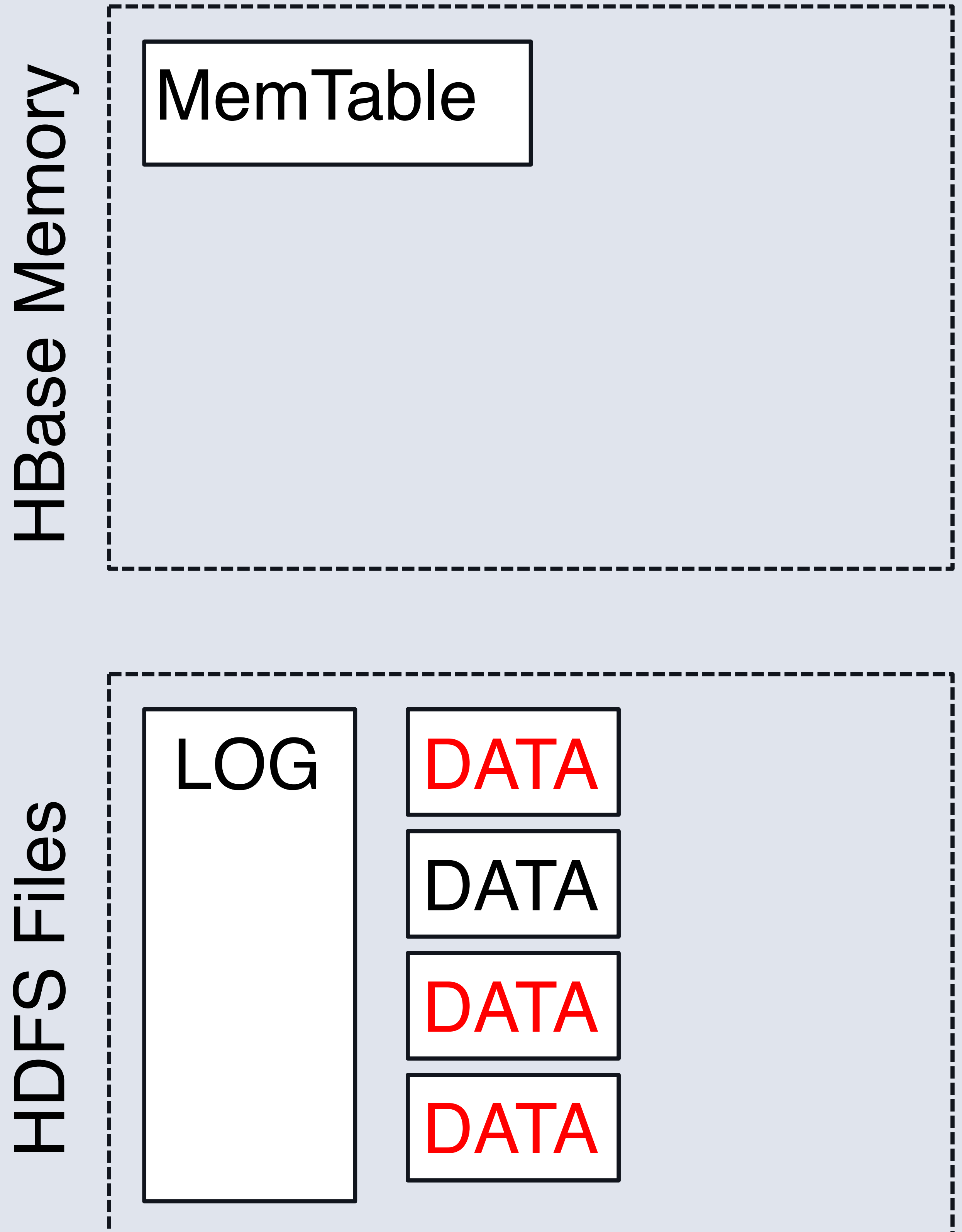


HBase's HDFS Files

Four activities do HDFS I/O:

- Logging
- Flushing
- Foreground reads

get's may check many files

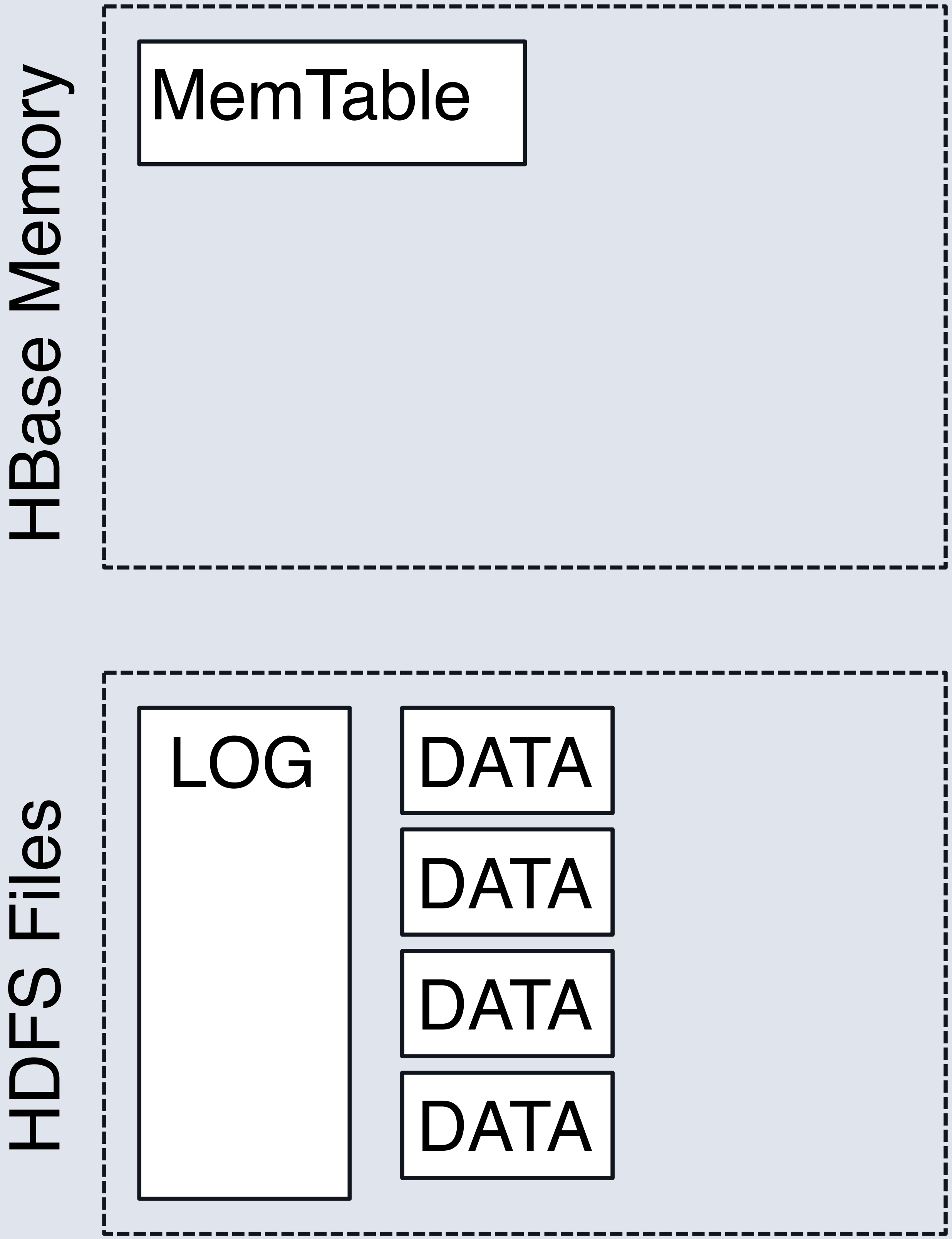


HBase's HDFS Files

Four activities do HDFS I/O:

- Logging
- Flushing
- Foreground reads

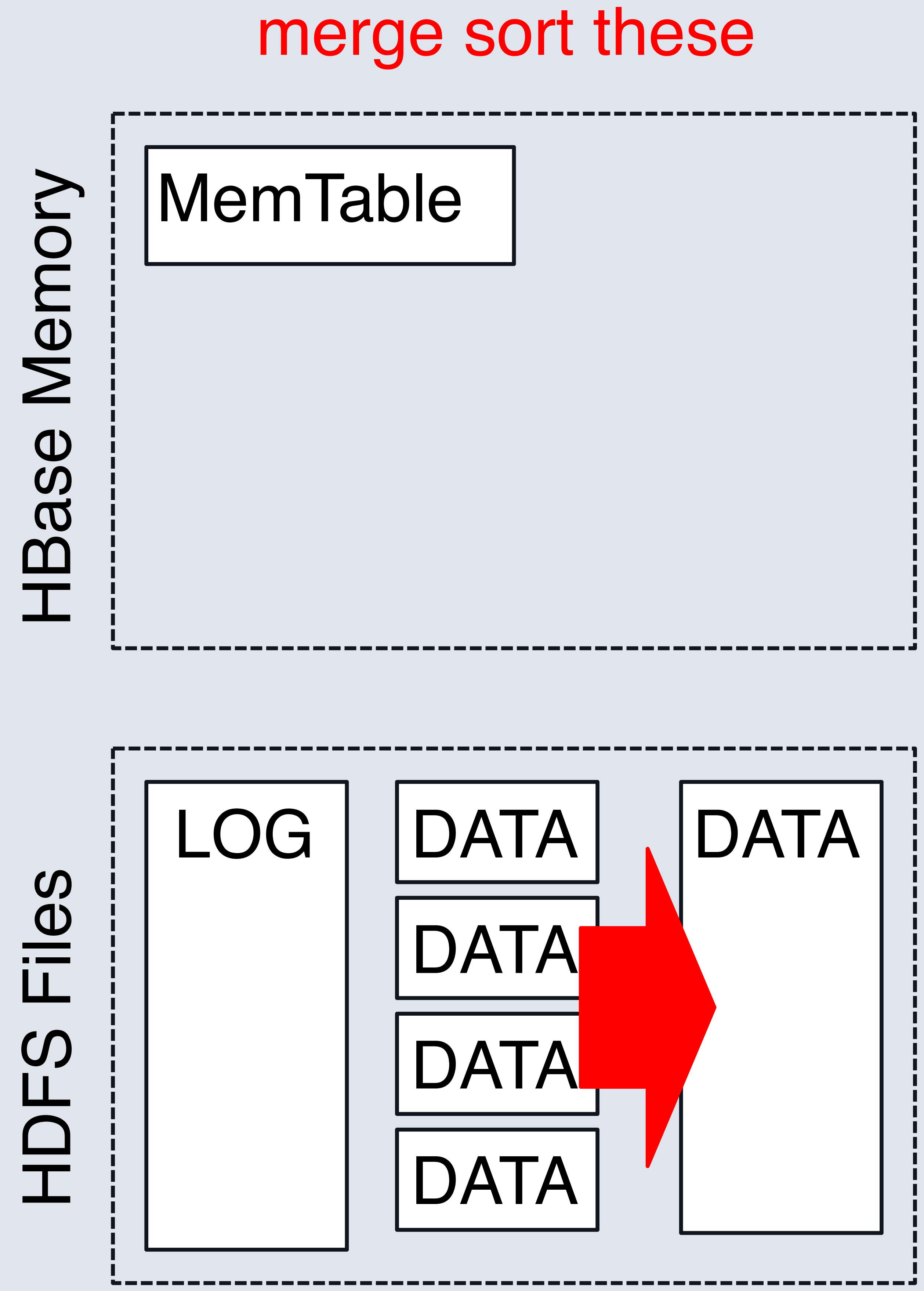
get's may check many files



HBase's HDFS Files

Four activities do HDFS I/O:

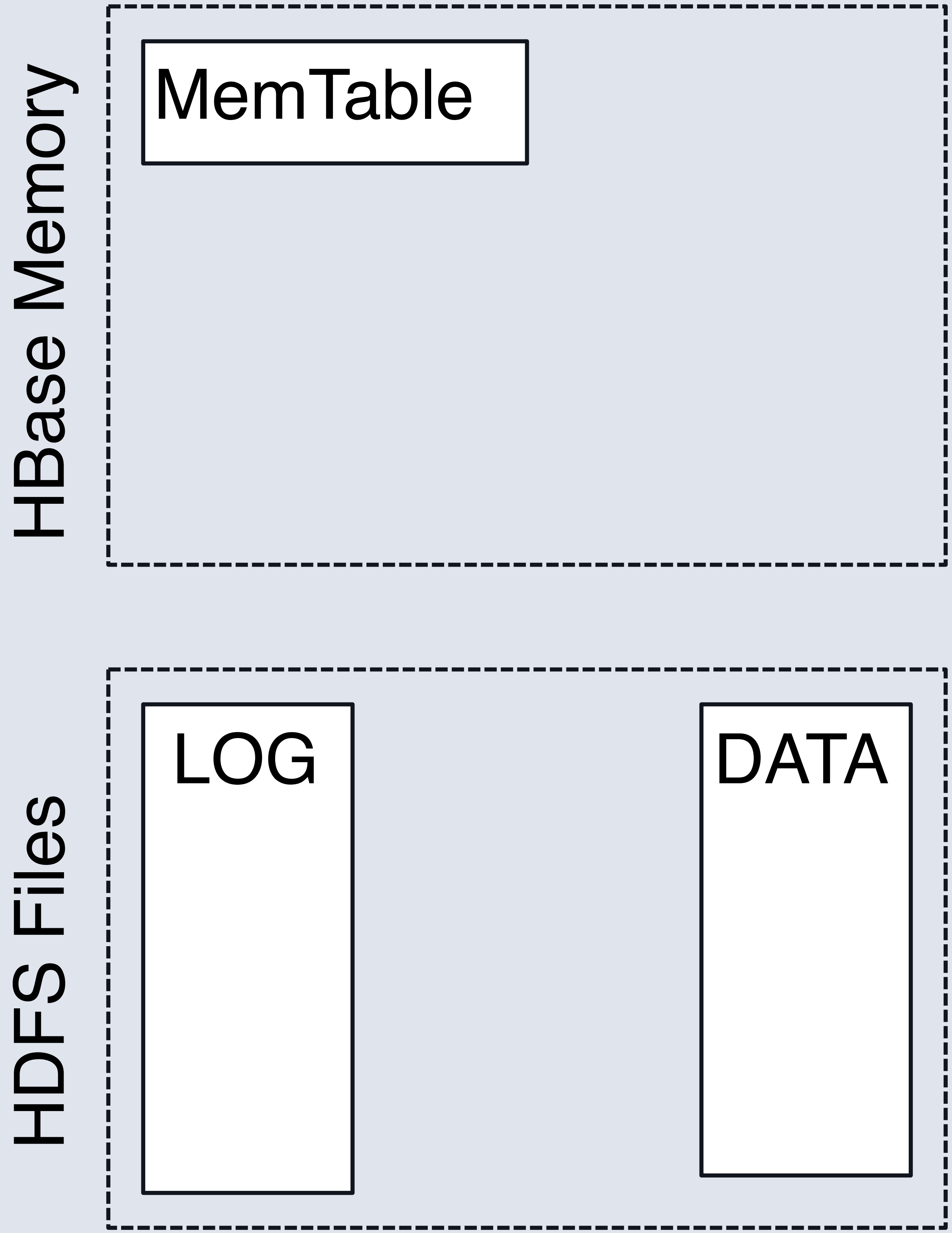
- Logging
- Flushing
- Foreground reads
- Compaction



HBase's HDFS Files

Four activities do HDFS I/O:

- Logging
- Flushing
- Foreground reads
- Compaction



HBase's HDFS Files

Four activities do HDFS I/O:

- Logging
- Flushing
- Foreground reads
- Compaction

Baseline I/O:

- <u>Flushing</u> and <u>foreground reads</u> are always required

HBase's HDFS Files

Four activities do HDFS I/O:

- Logging
- Flushing
- Foreground reads
- Compaction

Baseline I/O:

- Flushing and foreground reads are always required

HBase overheads:

- Logging: useful for crash recovery (not normal operation)
- Compaction: useful for performance (not correctness)

Facebook Messages Outline

Background

Workload Analysis

- I/O causes
- File size
- Sequentiality

Layer Integration

- Local compaction
- Combined logging

Discussion

Workload Analysis Questions

At each layer, what activities read or write?

How large are created files?

How sequential is I/O?

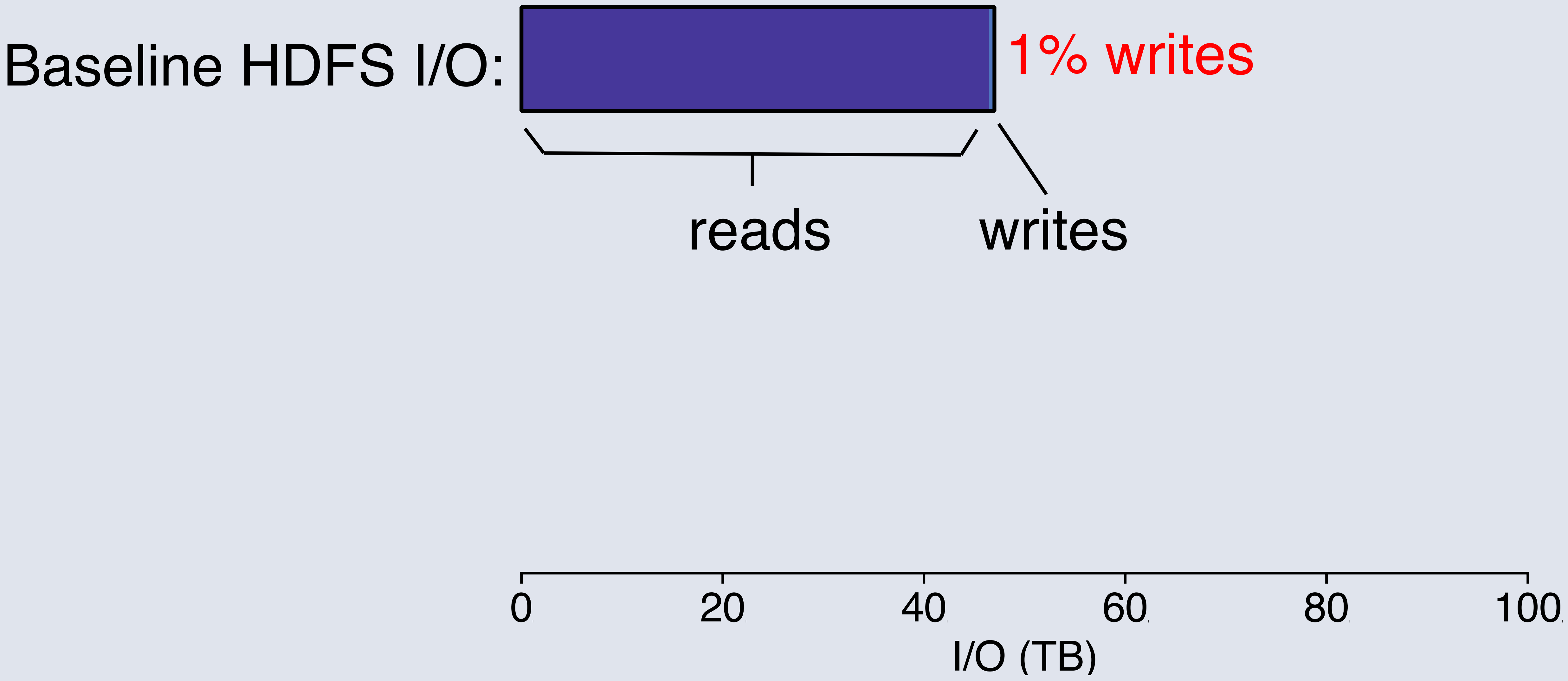
Workload Analysis Questions

At each layer, what activities read or write?

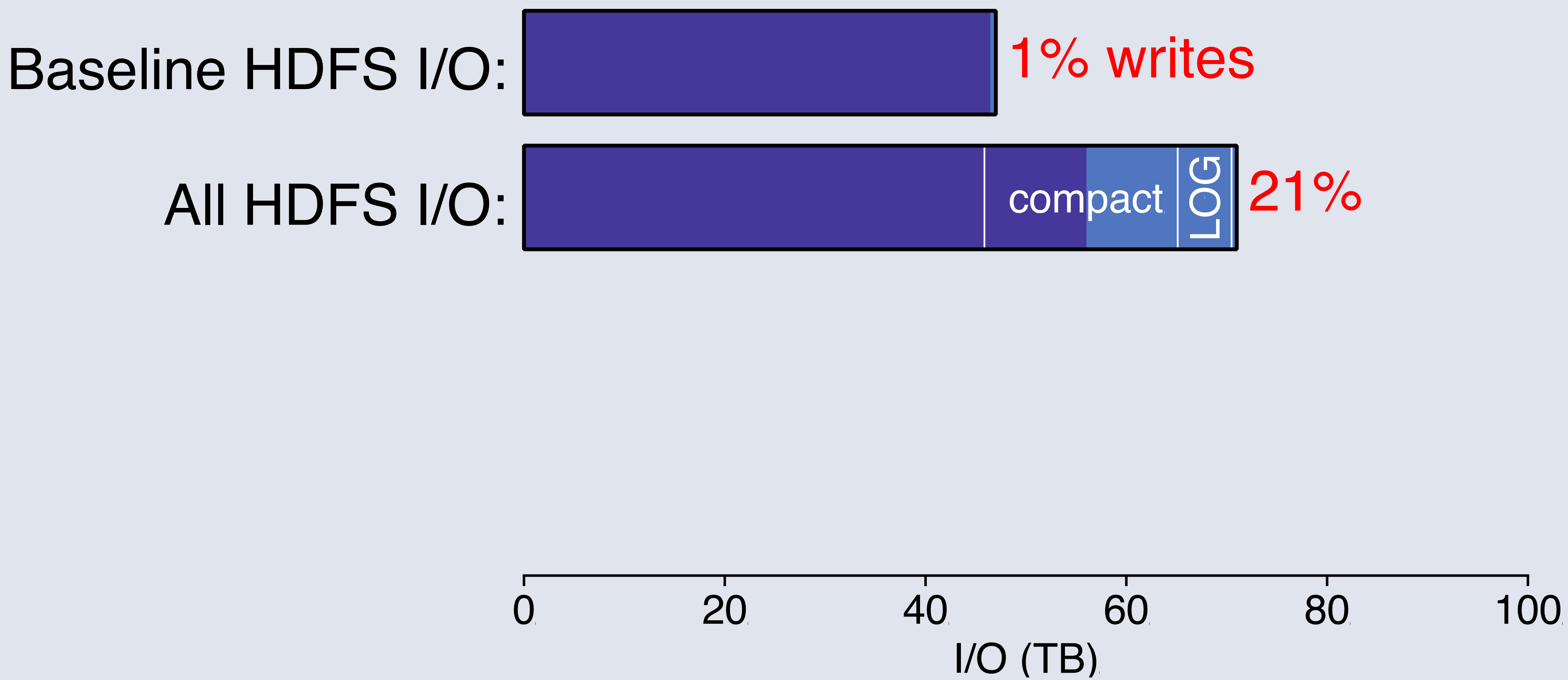
How large are created files?

How sequential is I/O?

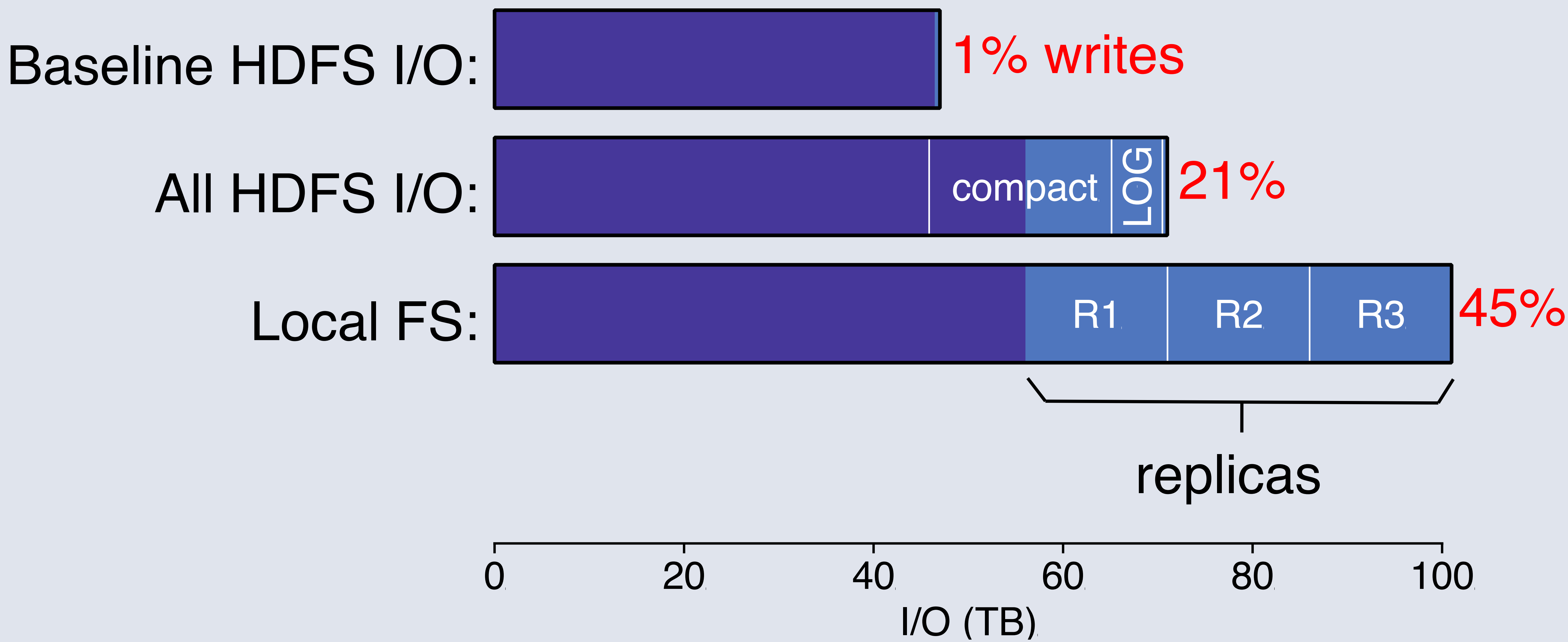
Cross-layer R/W Ratios



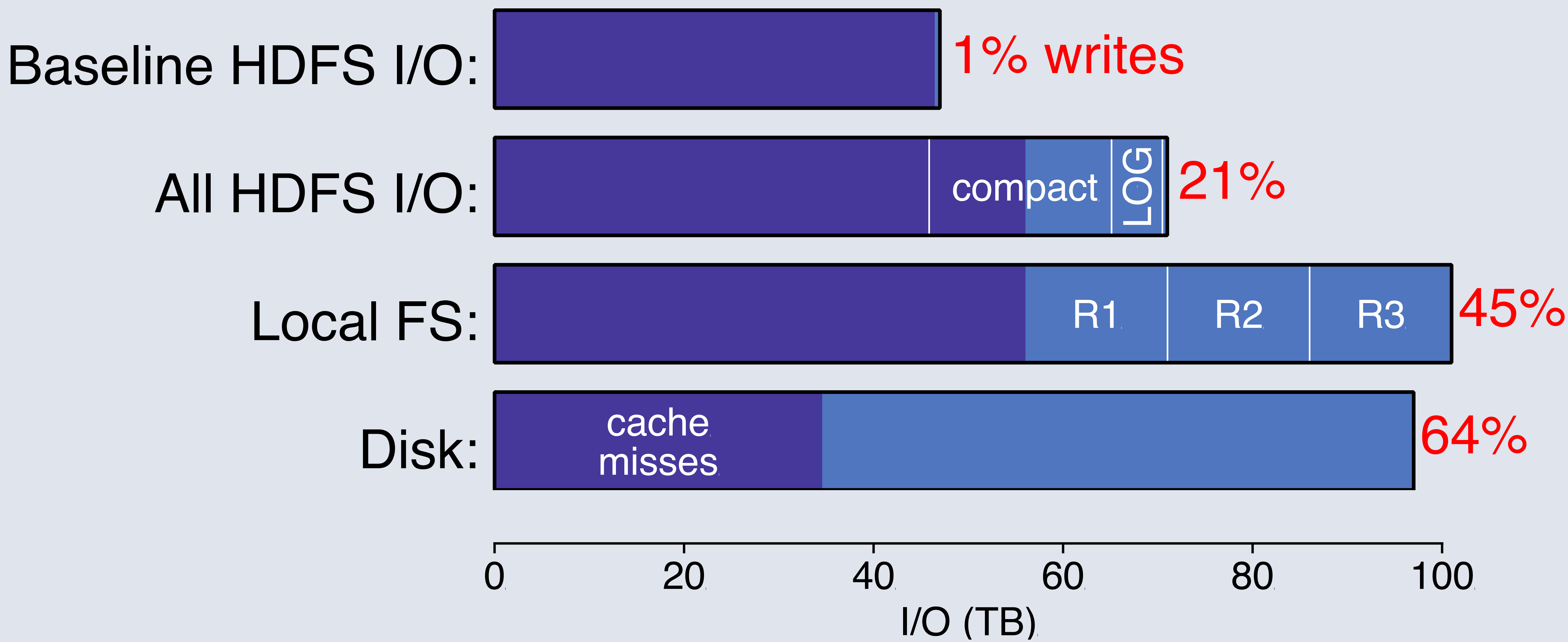
Cross-layer R/W Ratios



Cross-layer R/W Ratios



Cross-layer R/W Ratios



Workload Analysis Conclusions

① Layers amplify writes: 1% => 64%

- Logging, compaction, and replication increase writes
- Caching decreases reads

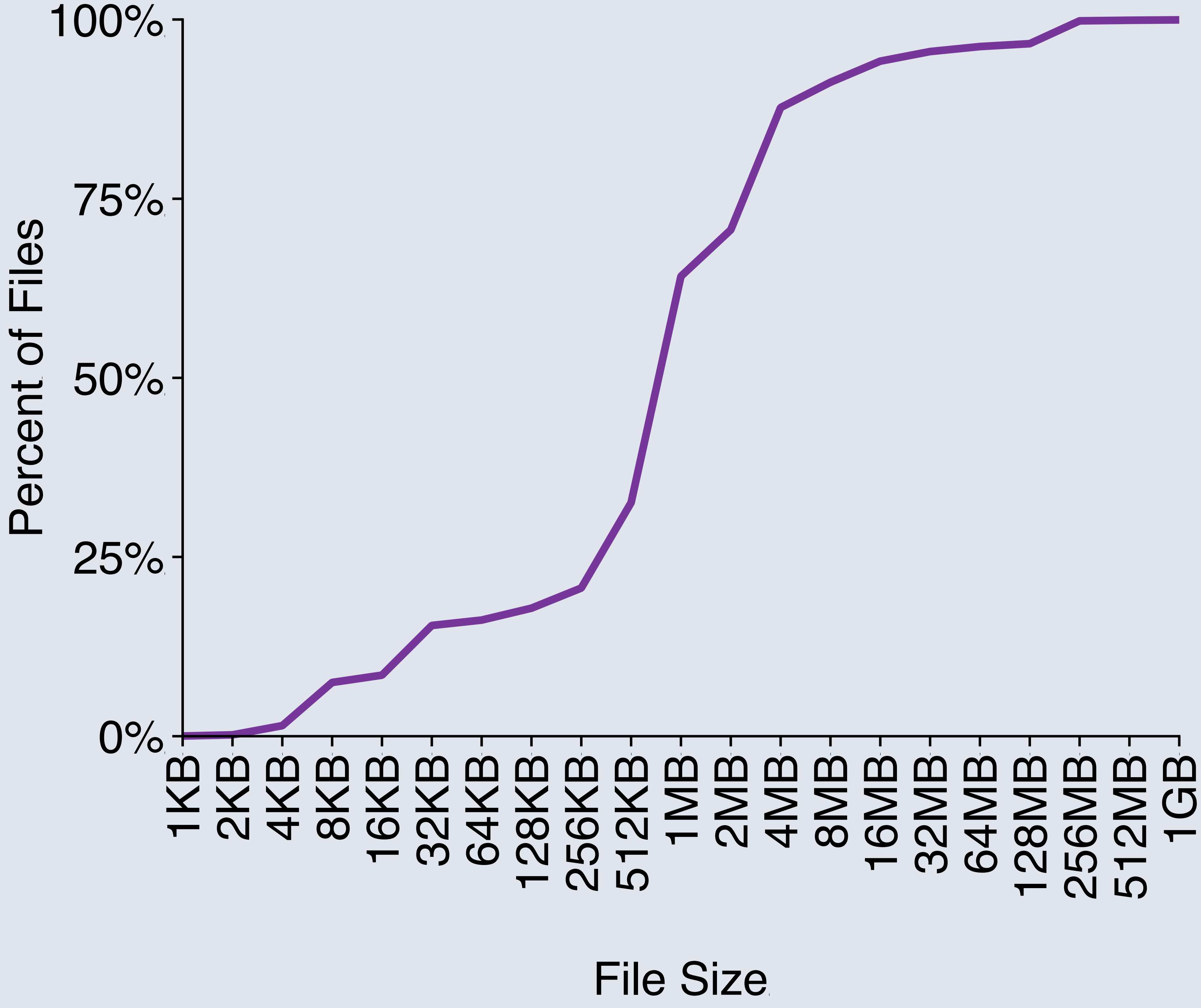
Workload Analysis Questions

At each layer, what activities read or write?

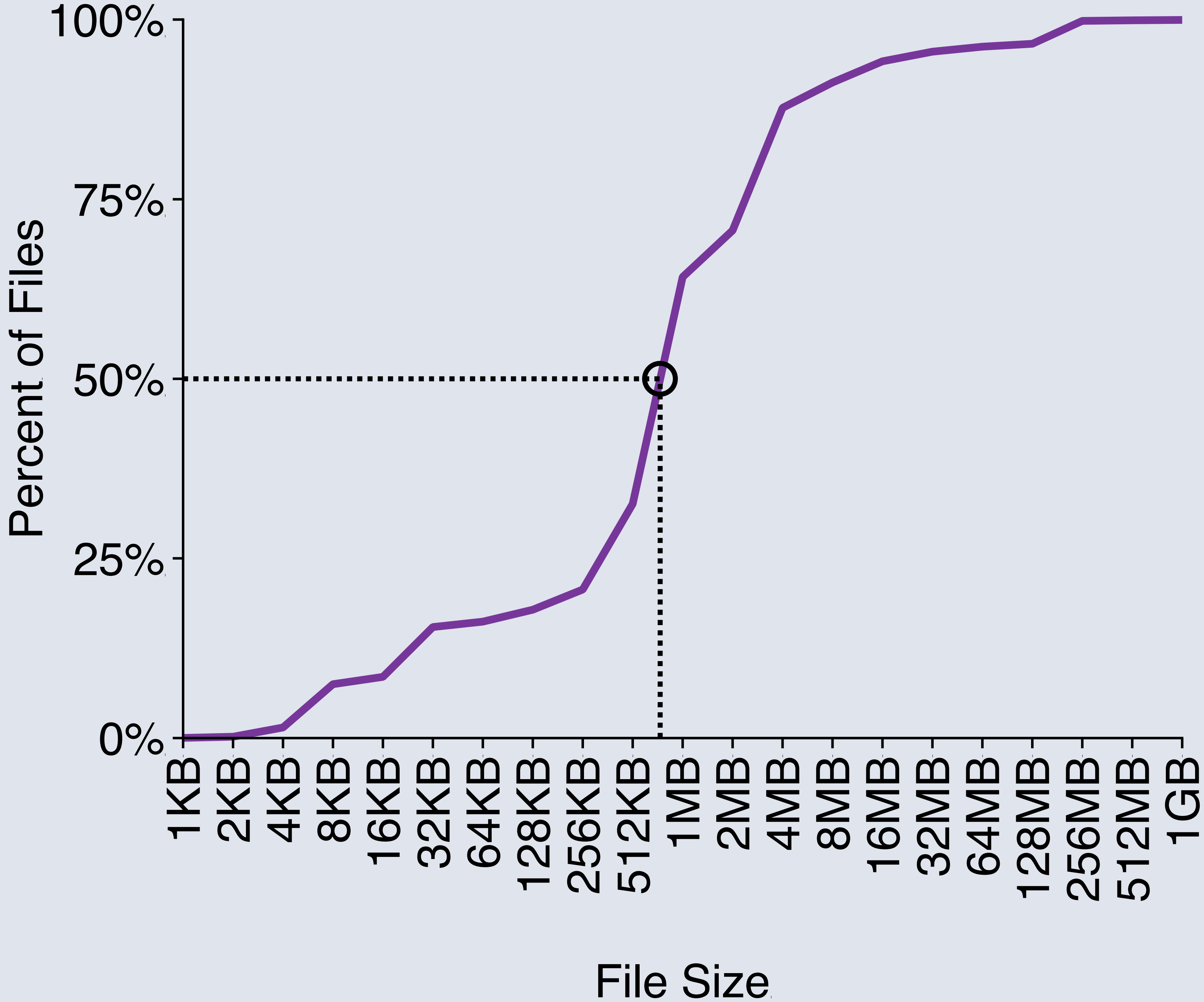
How large are created files?

How sequential is I/O?

Created Files: Size Distribution

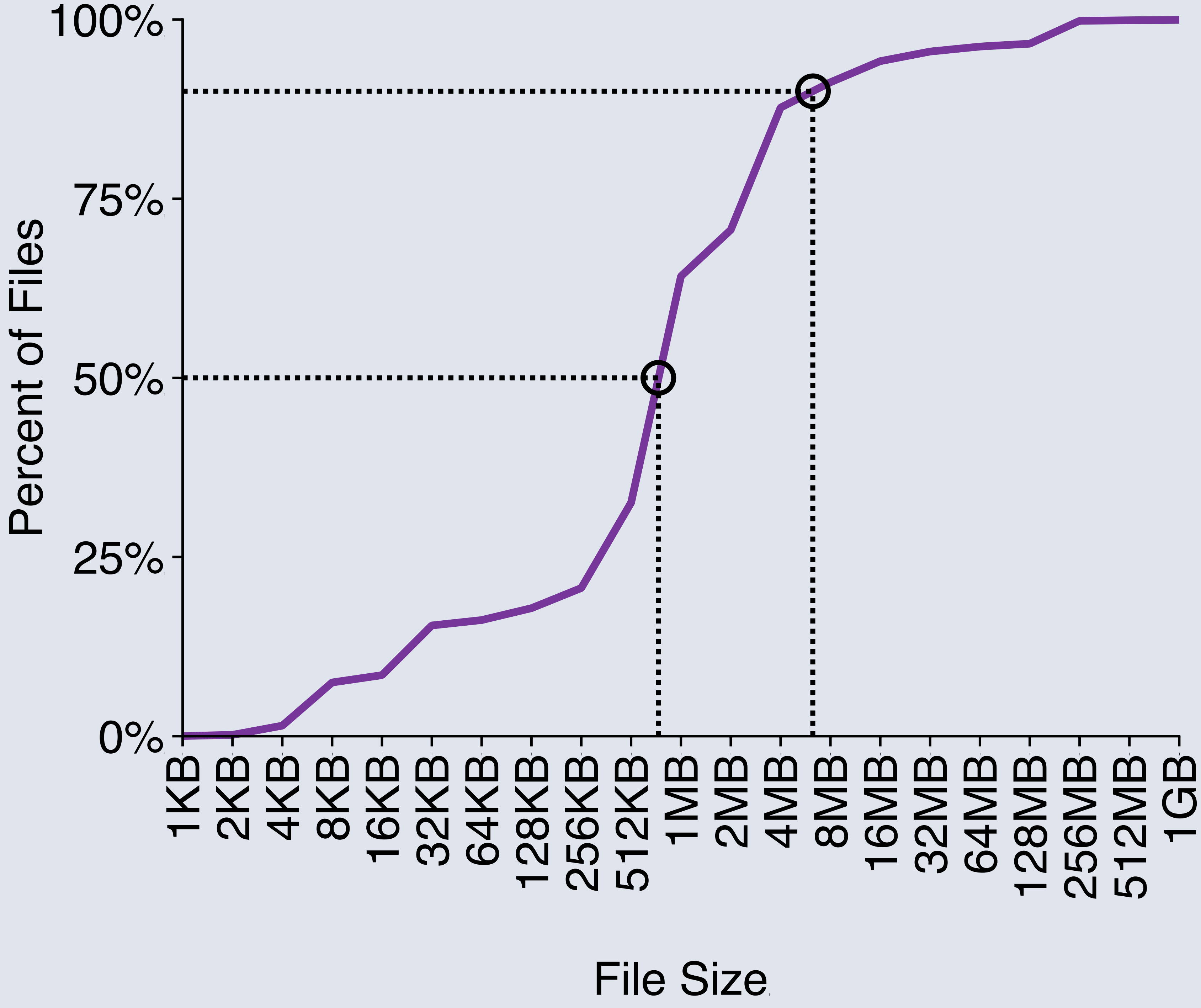


Created Files: Size Distribution



50% of files are <750KB

Created Files: Size Distribution



90% of files are <6.3MB

Workload Analysis Conclusions

① Layers amplify writes: 1% => 64%

② Files are very small: 90% smaller than 6.3MB

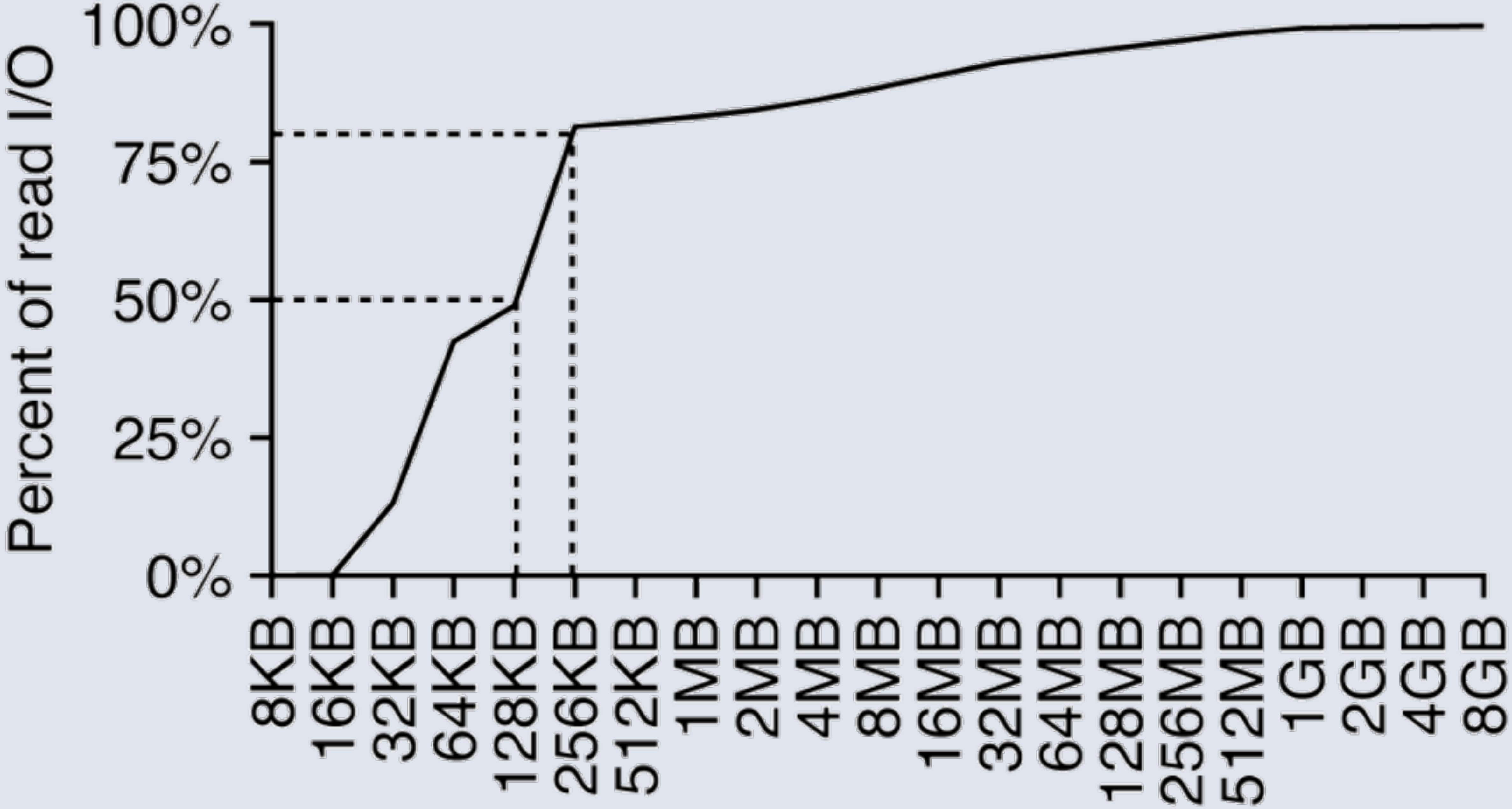
Workload Analysis Questions

At each layer, what activities read or write?

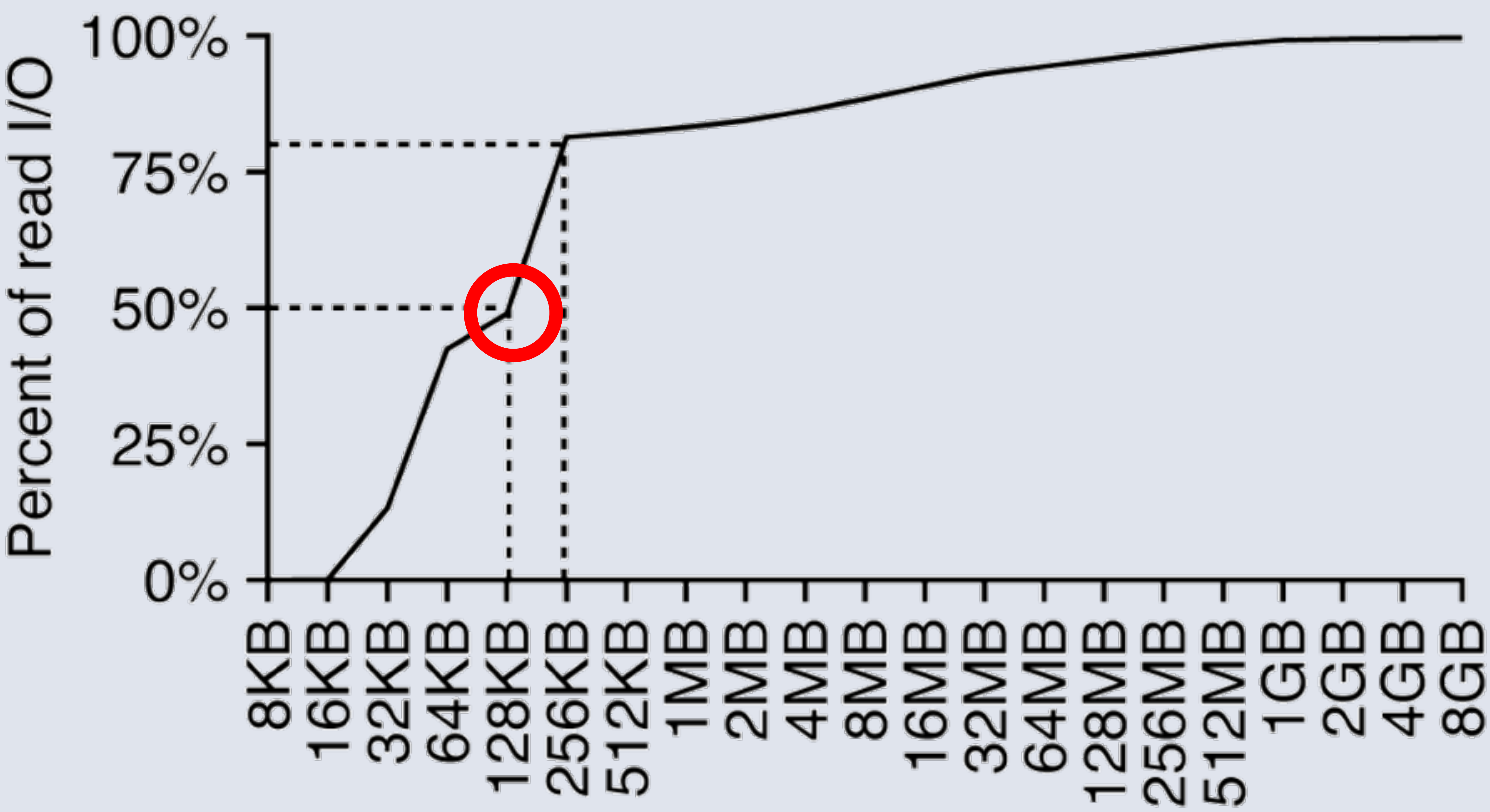
How large are created files?

How sequential is I/O?

Reads: Run Size

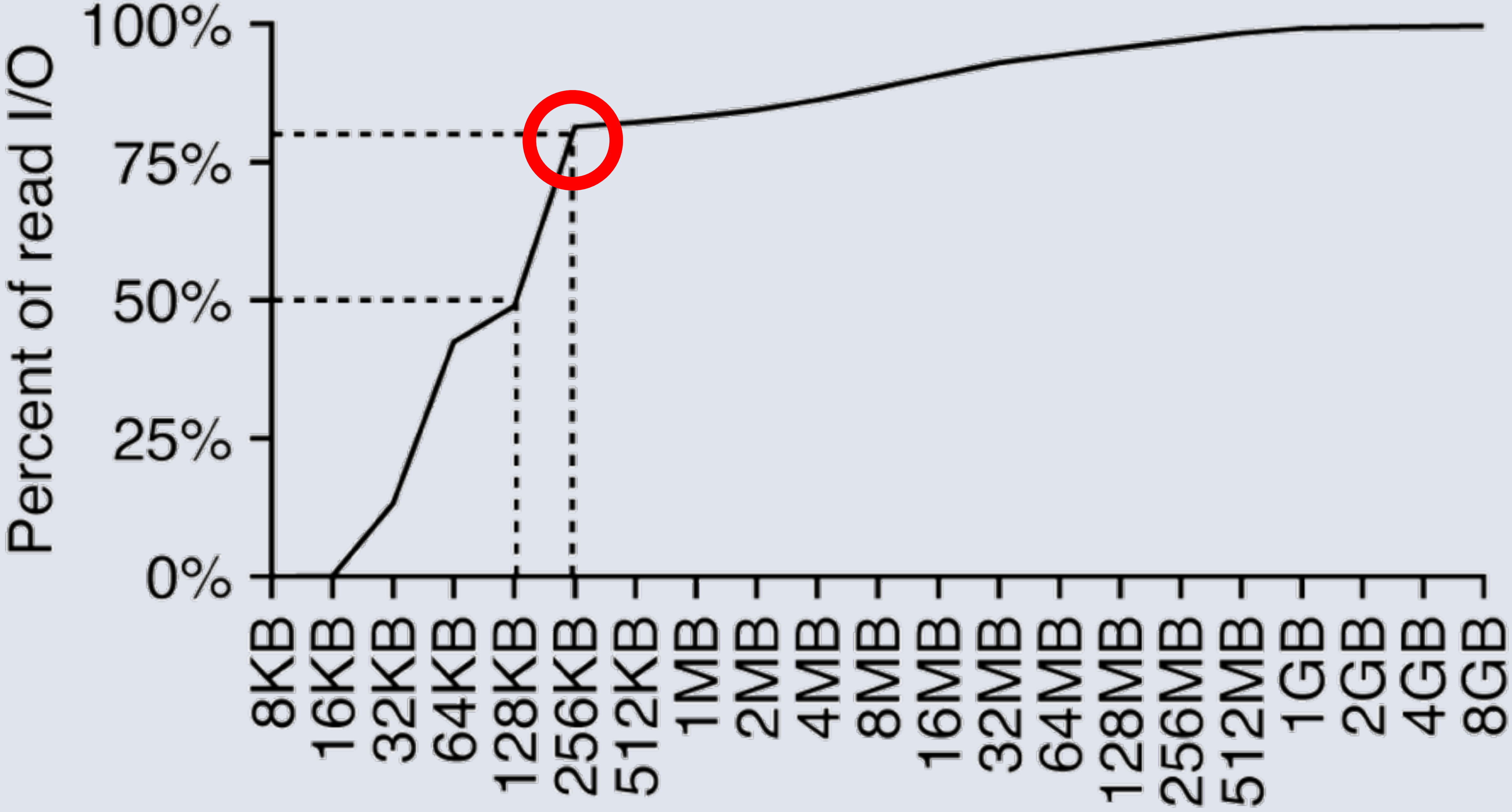


Reads: Run Size



50% of runs (weighted by I/O) <130KB

Reads: Run Size



80% of files are <256KB

Workload Analysis Conclusions

① Layers amplify writes: 1% => 64%

② Files are very small: 90% smaller than 6.3MB

③ Fairly random I/O: 130KB median read run

Facebook Messages Outline

Background

Workload Analysis

- I/O causes
- File size
- Sequentiality

Layer Integration

- Local compaction
- Combined logging

Discussion

Software Architecture: Workload Implications

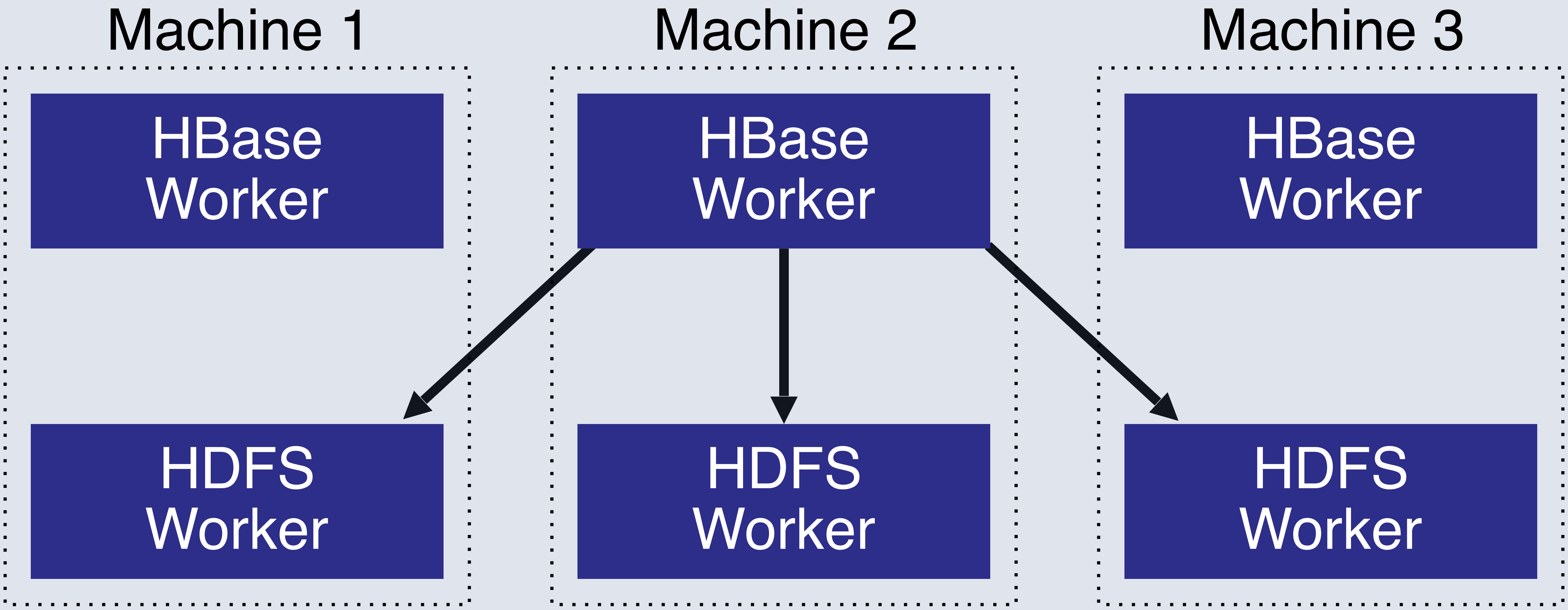
Writes are amplified

- 1% at HDFS (w/o overheads) to 64% at disk (30GB RAM)
- <u>We should optimize writes</u>

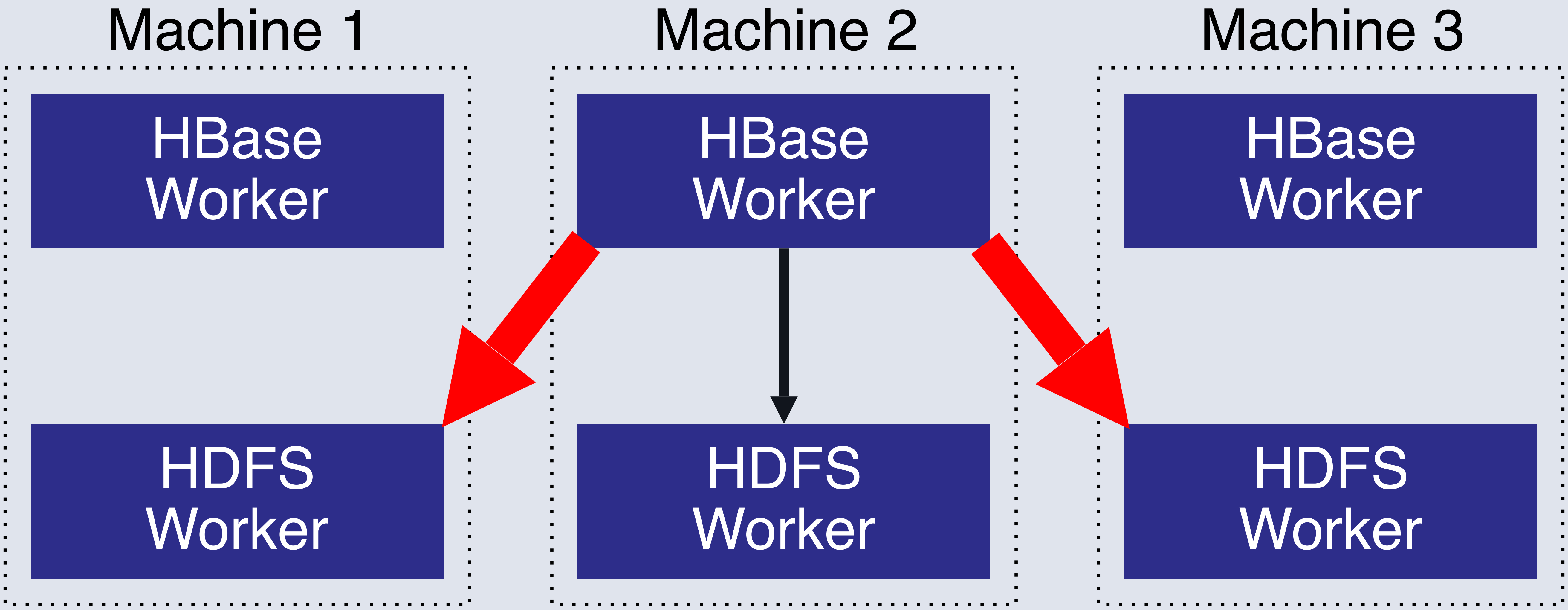
61% of writes are for compaction

36% of writes are for logging

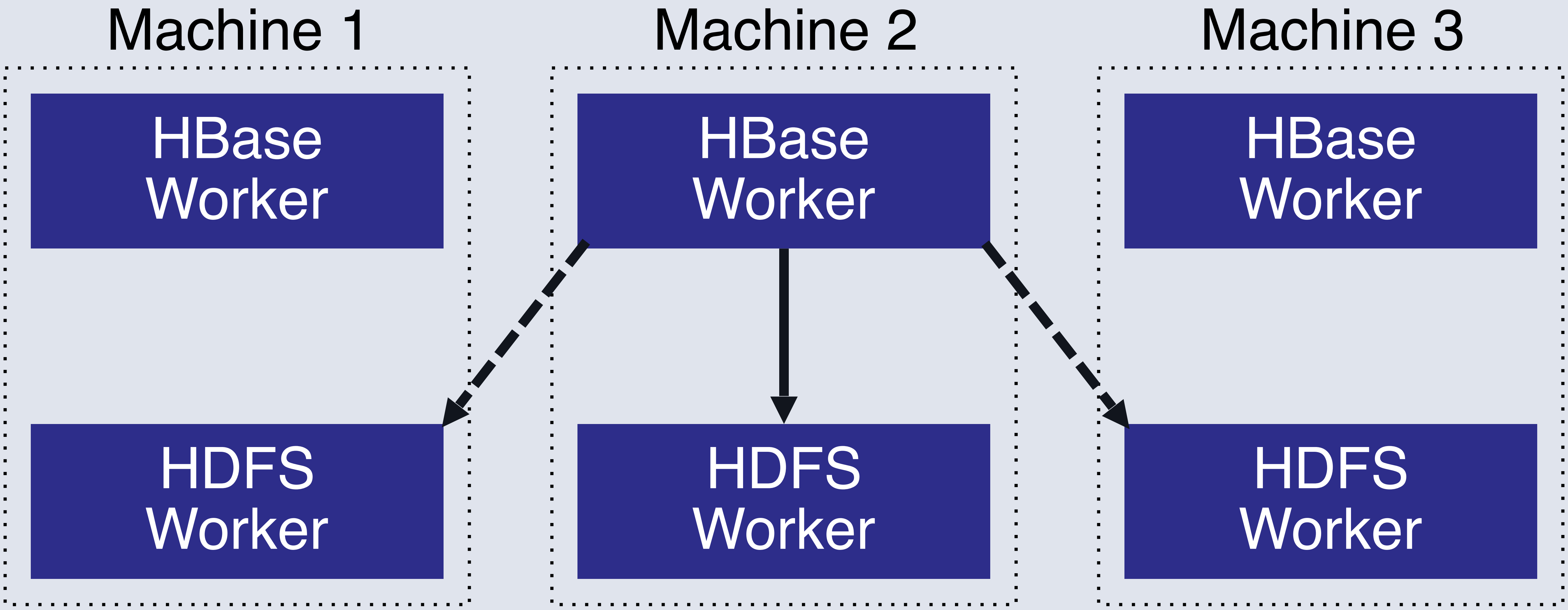
Replication Overview



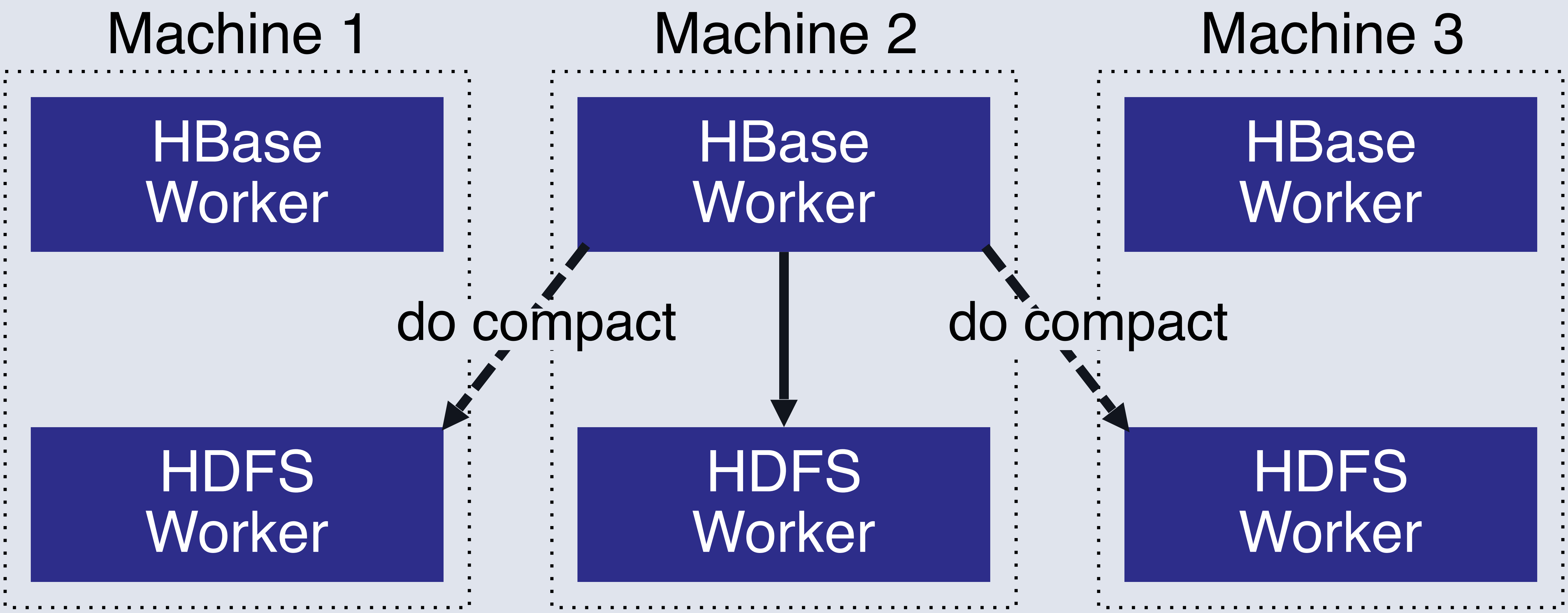
Problem: Network I/O (red lines)



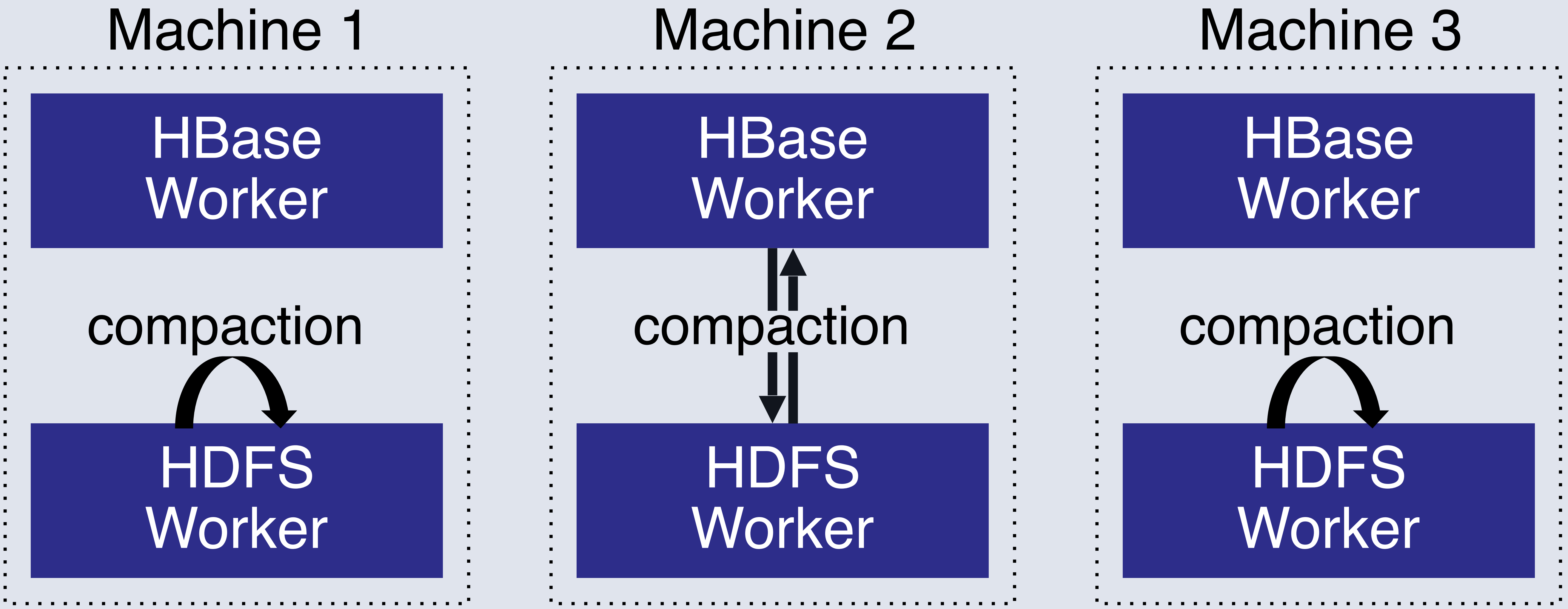
Solution: Ship Computation to Data



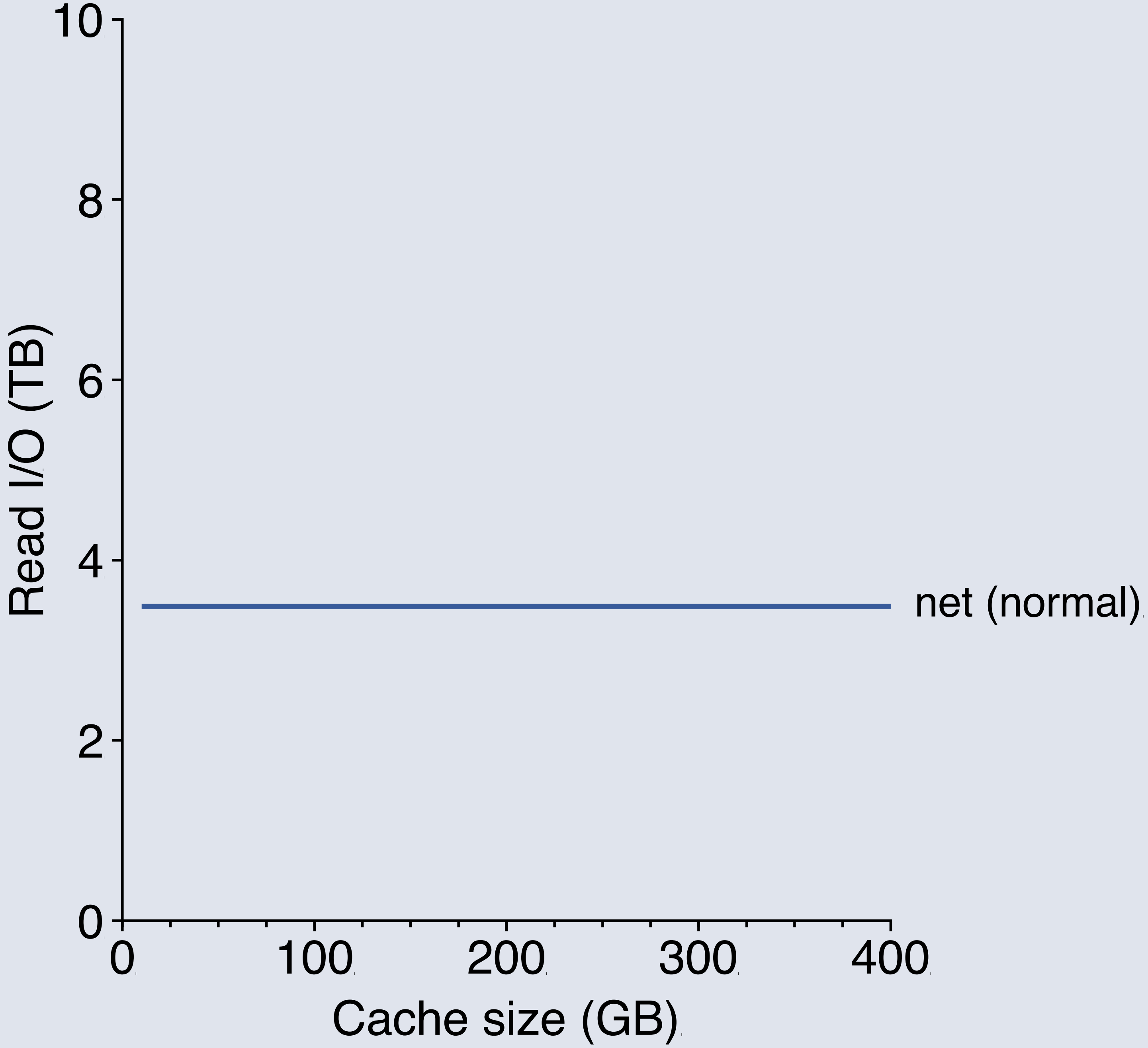
In Our Case, do *Local Compaction*



In Our Case, do *Local Compaction*

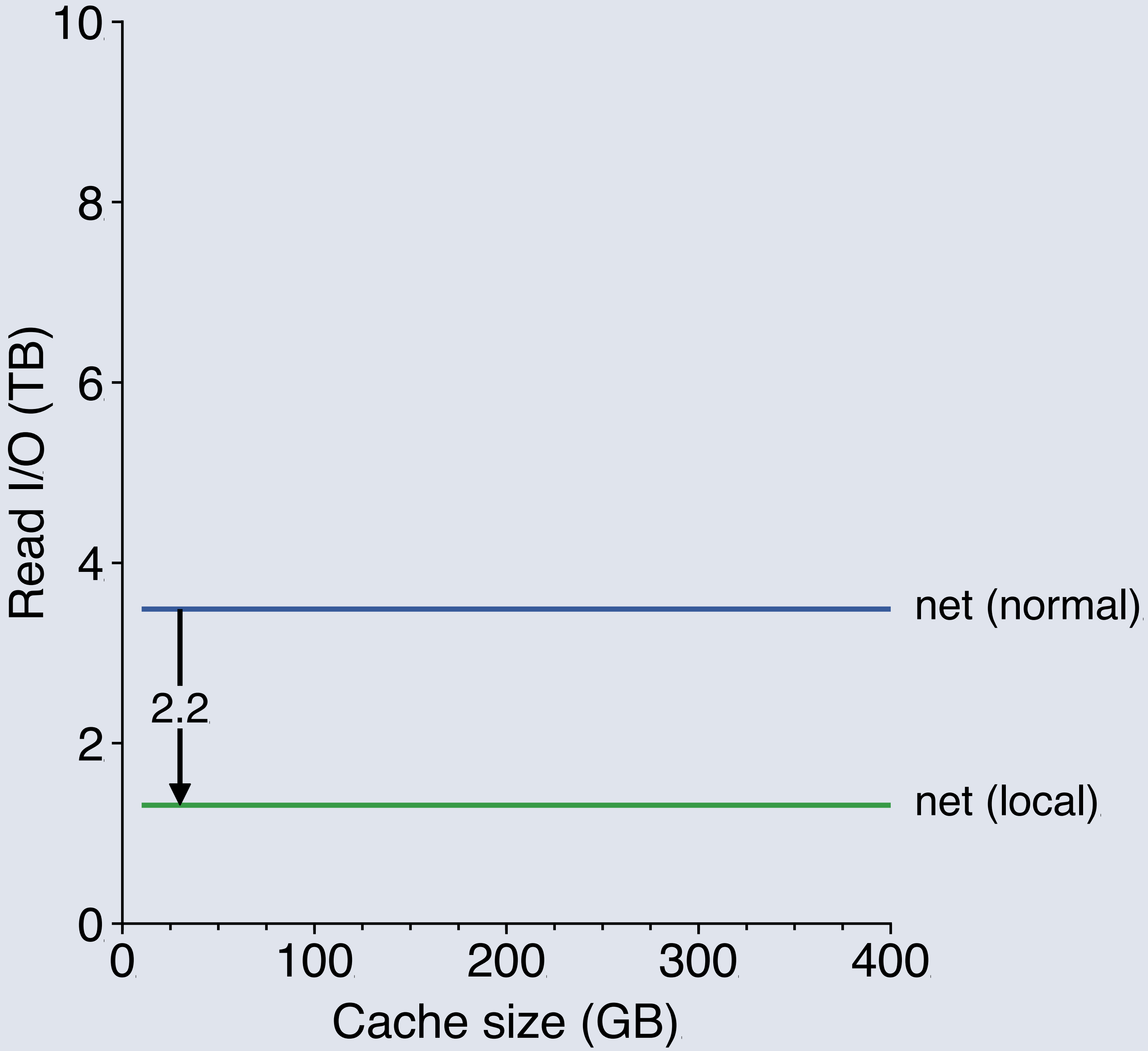


Local Compaction



Normally 3.5TB of network I/O

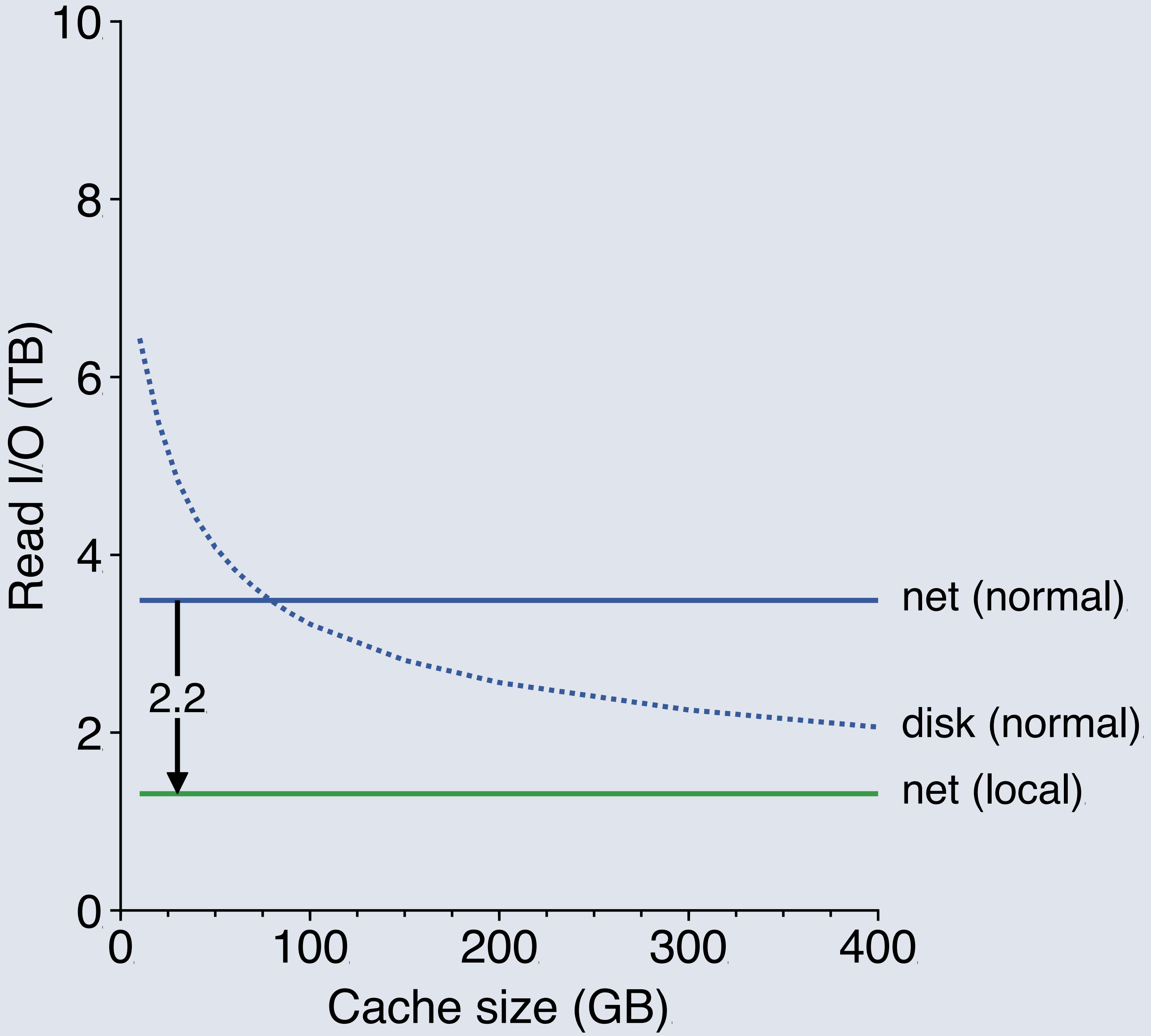
Local Compaction



Normally 3.5TB of network I/O

Local comp: 62% reduction

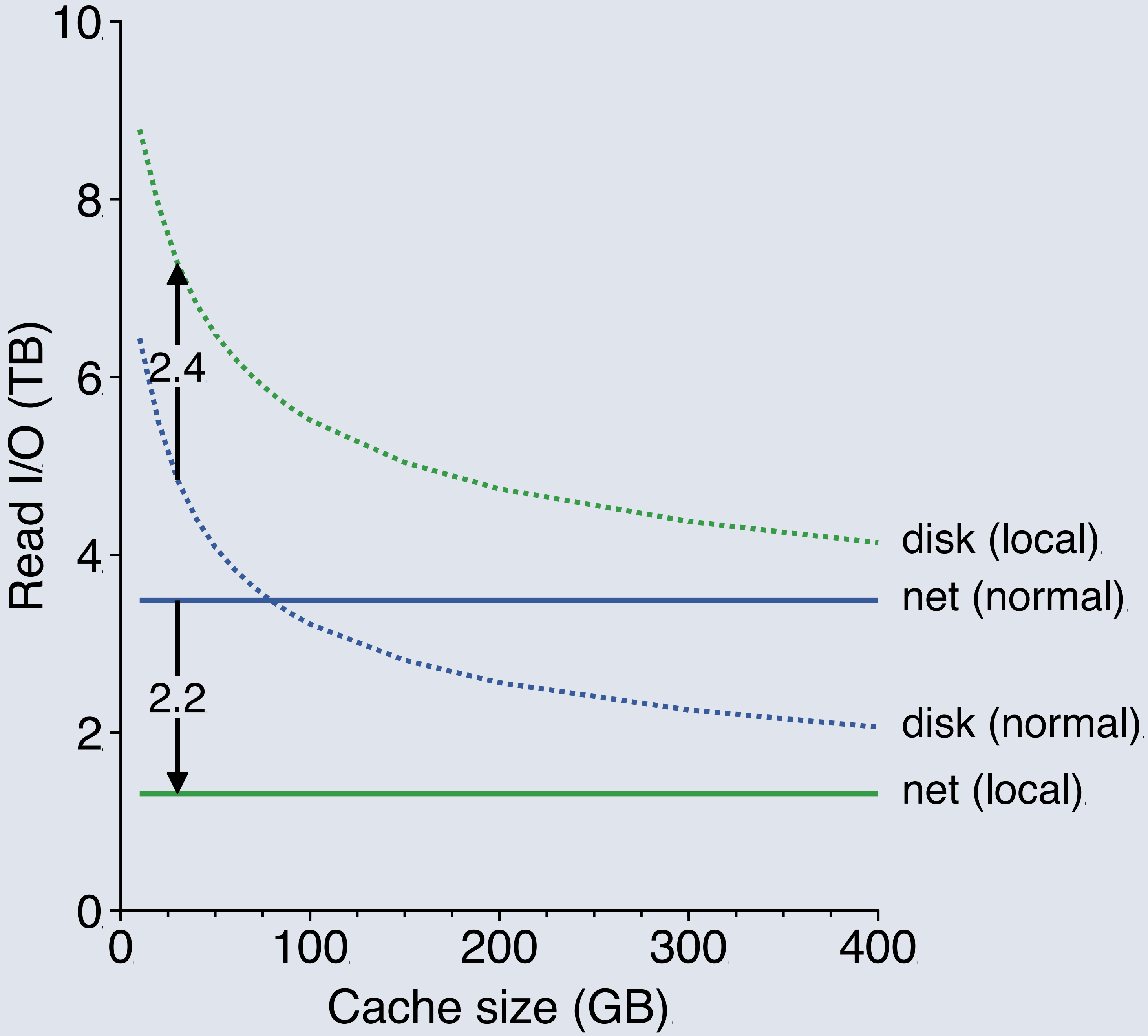
Local Compaction



Normally 3.5TB of network I/O

Local comp: 62% reduction

Local Compaction



Normally 3.5TB of network I/O

Local comp: 62% reduction

Network I/O becomes disk I/O

- 9% overhead (30GB cache)
- Compaction reads are
  (a) usually misses,
  (b) pollute cache
- Disk I/O is much cheaper

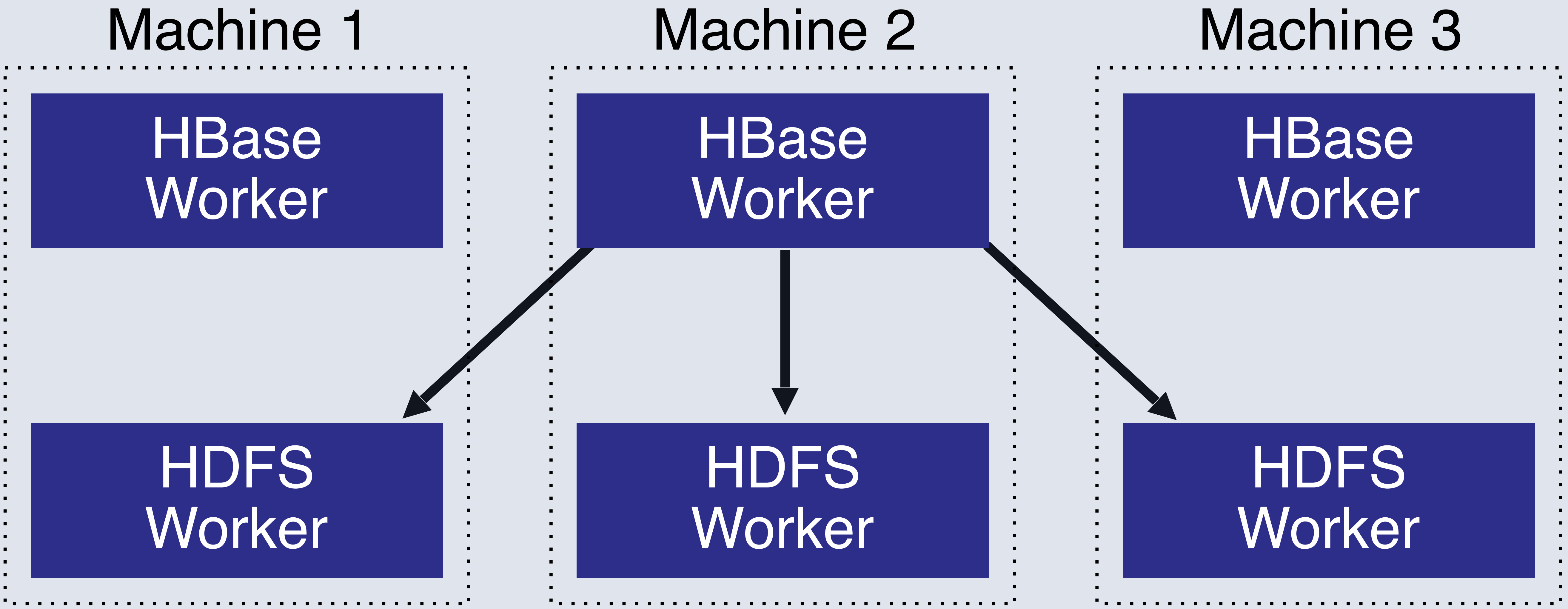
Related Work: Salus

Wang *et al.* built Salus, an implementation of the HBase interface that replicates DB compute as well as storage

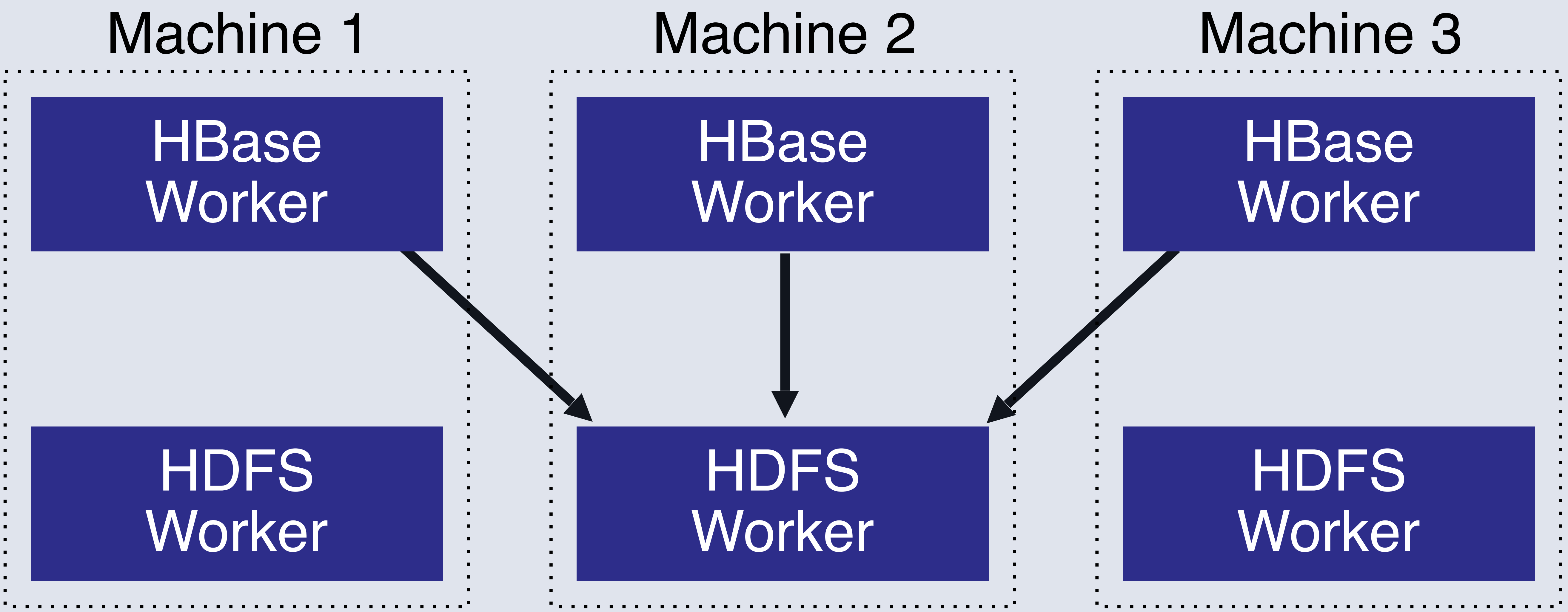
- Side effect: compaction work is replicated, so Salus does local compaction

Finding: *"Salus often outperforms HBase, especially when disk bandwidth is plentiful compared to network bandwidth."*

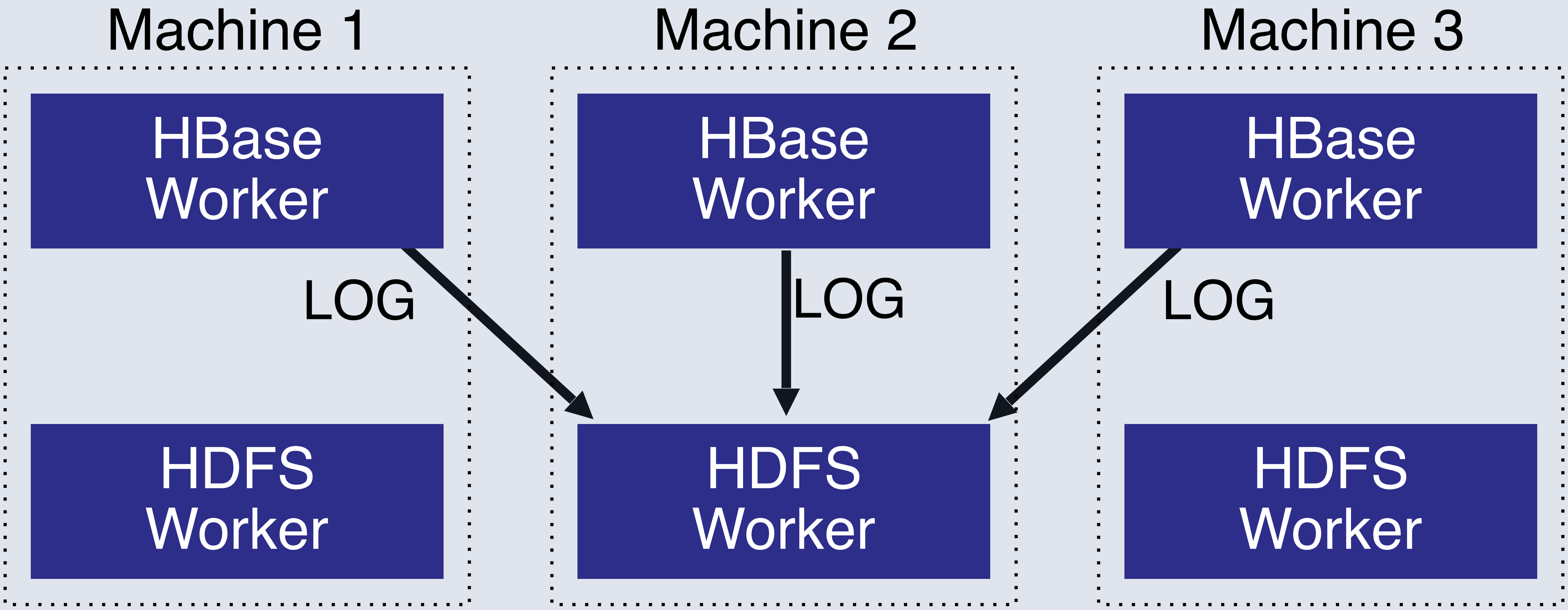
Replication Overview



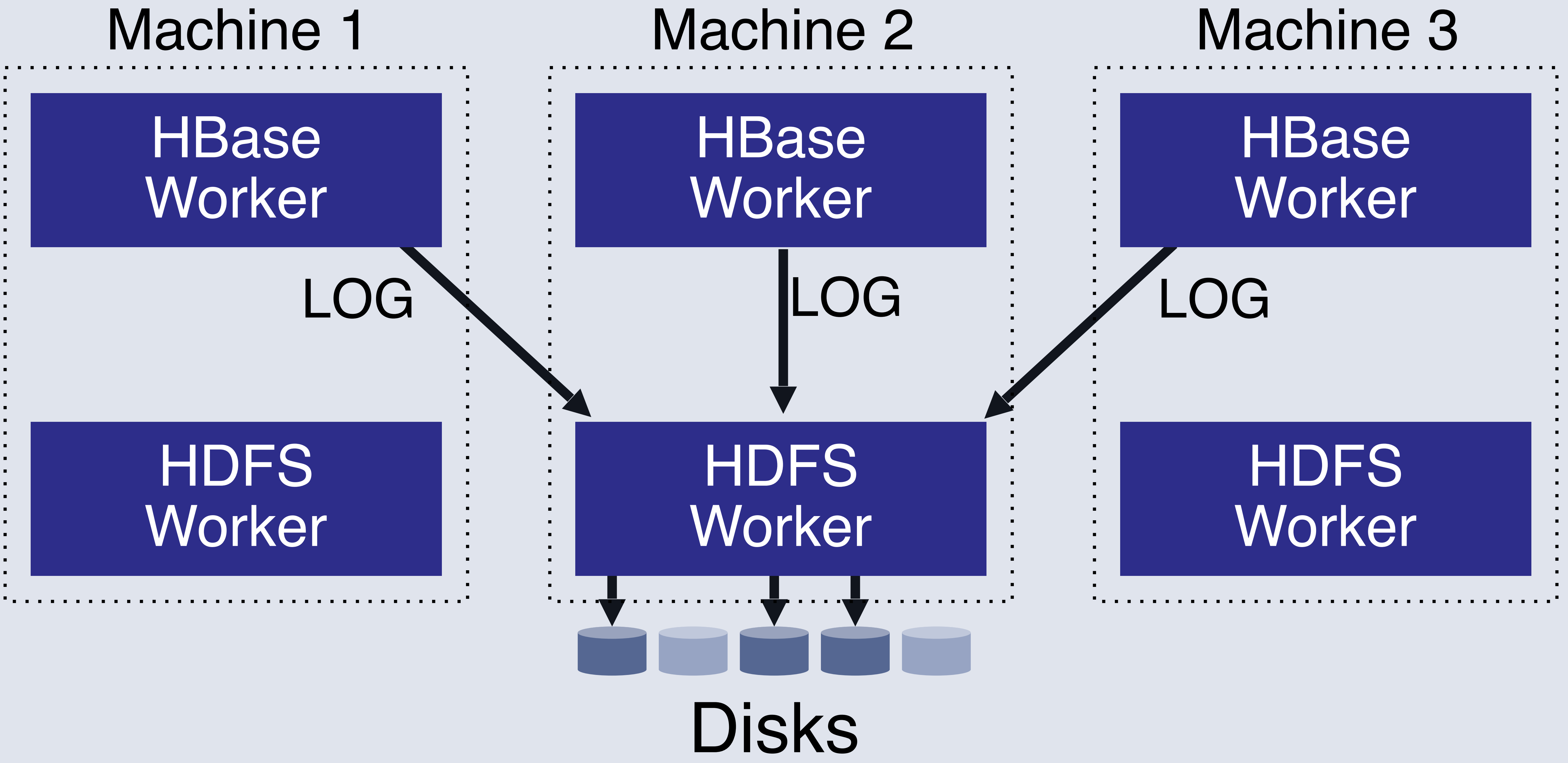
Replication Overview



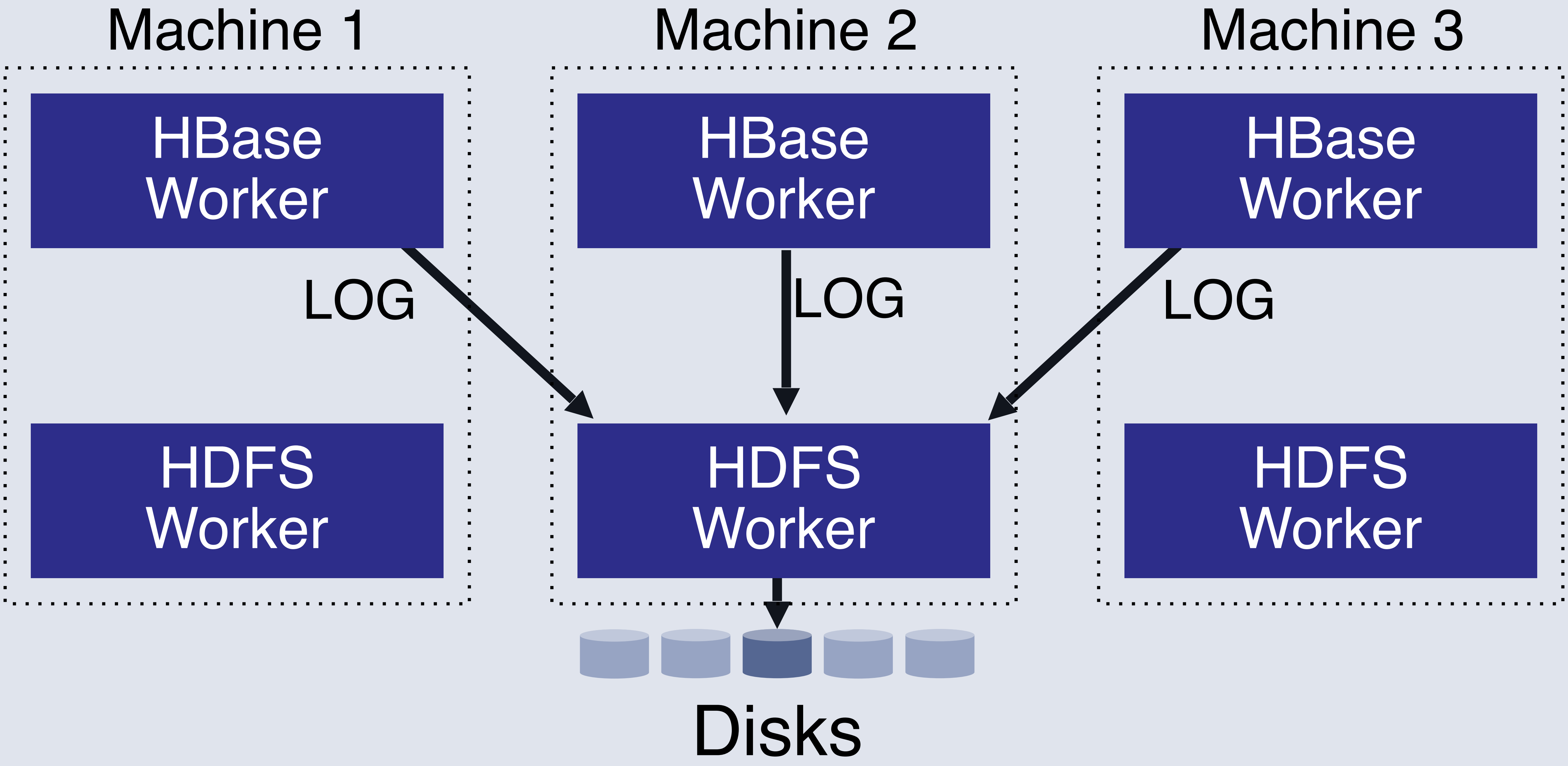
Typical HDFS Worker Receives Logs from 3



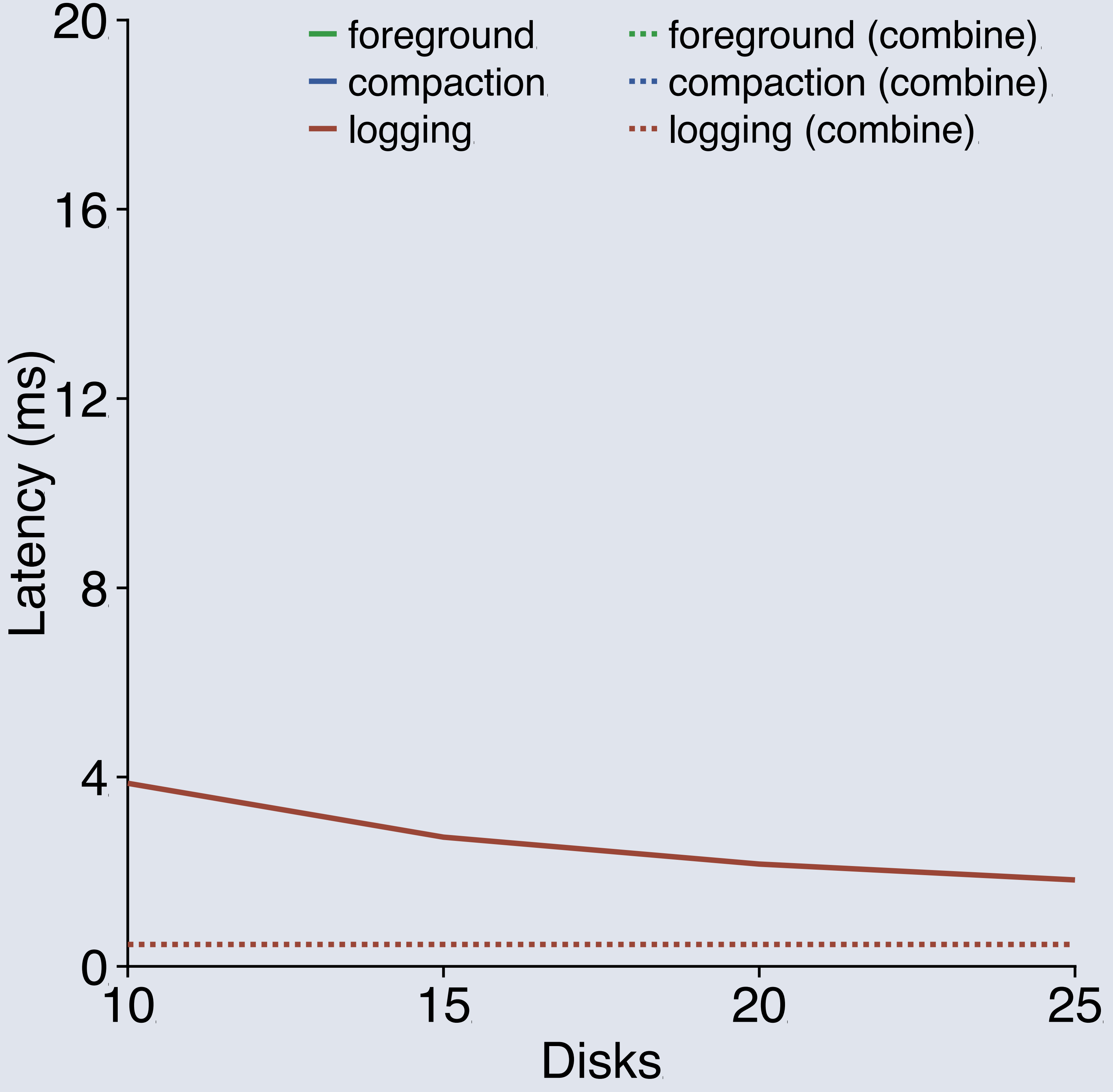
Problem: Extra Seeks for Logging



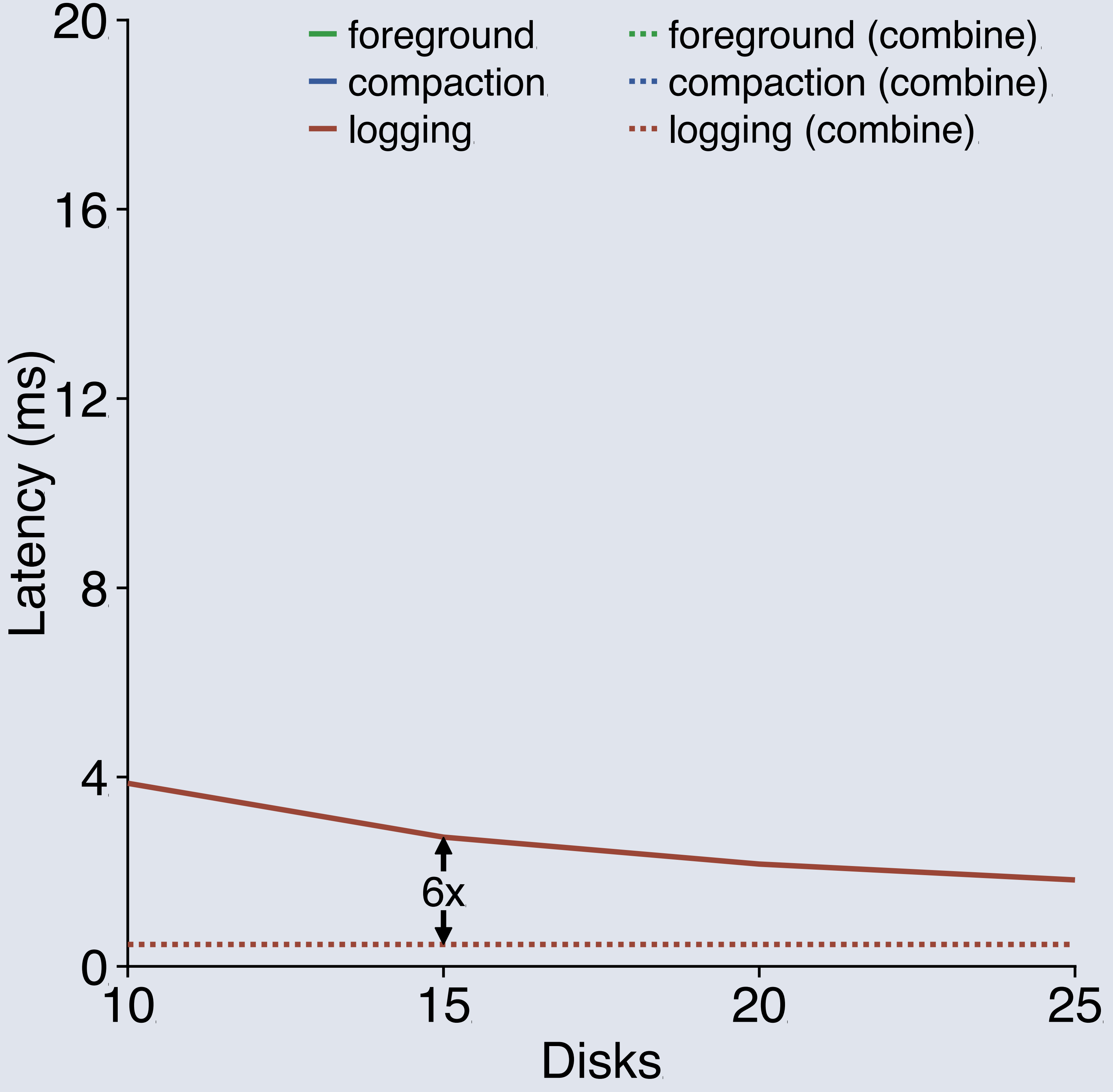
Solution: Combine Logs (New HDFS API)



Combined Logging

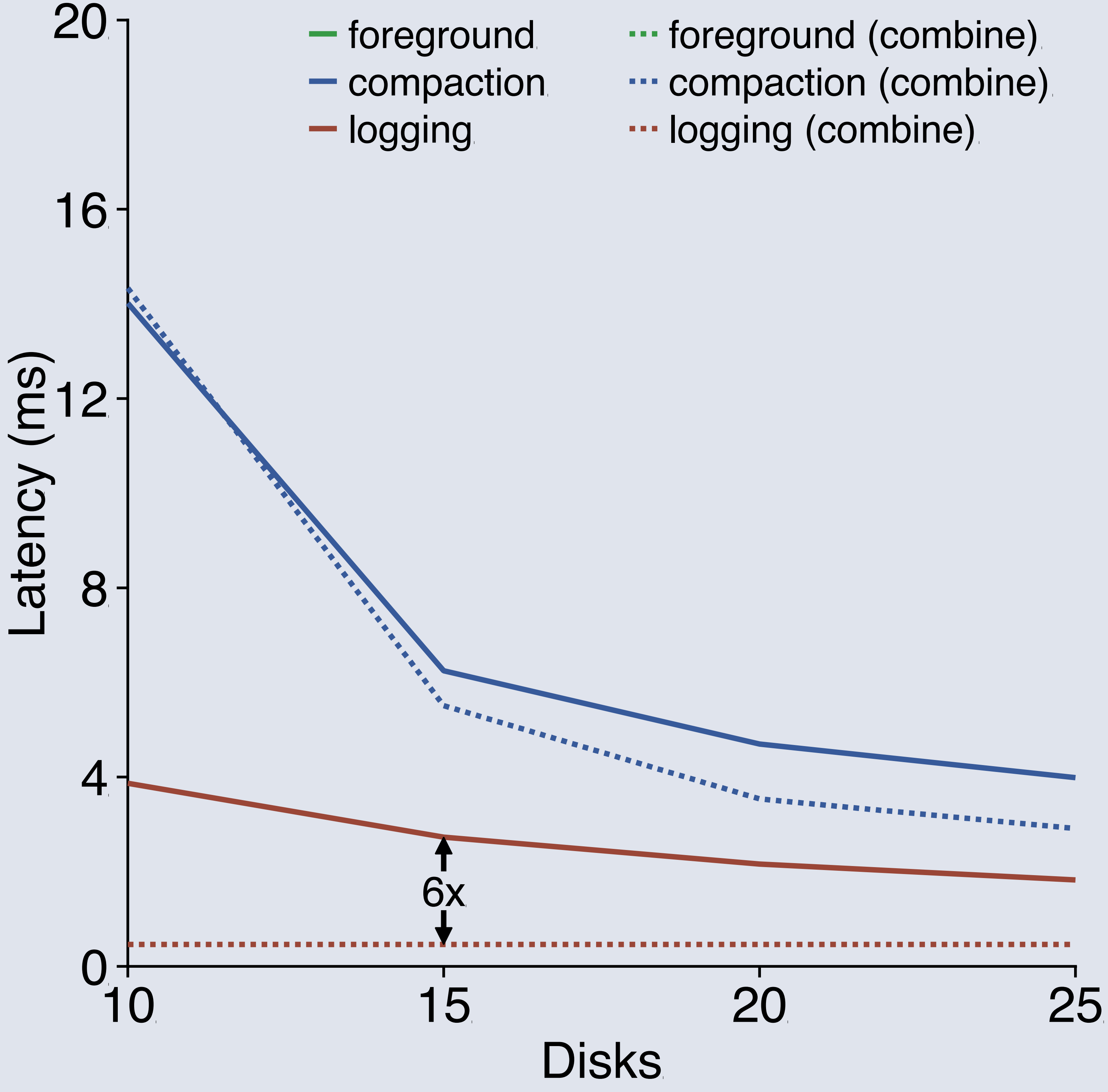


Combined Logging



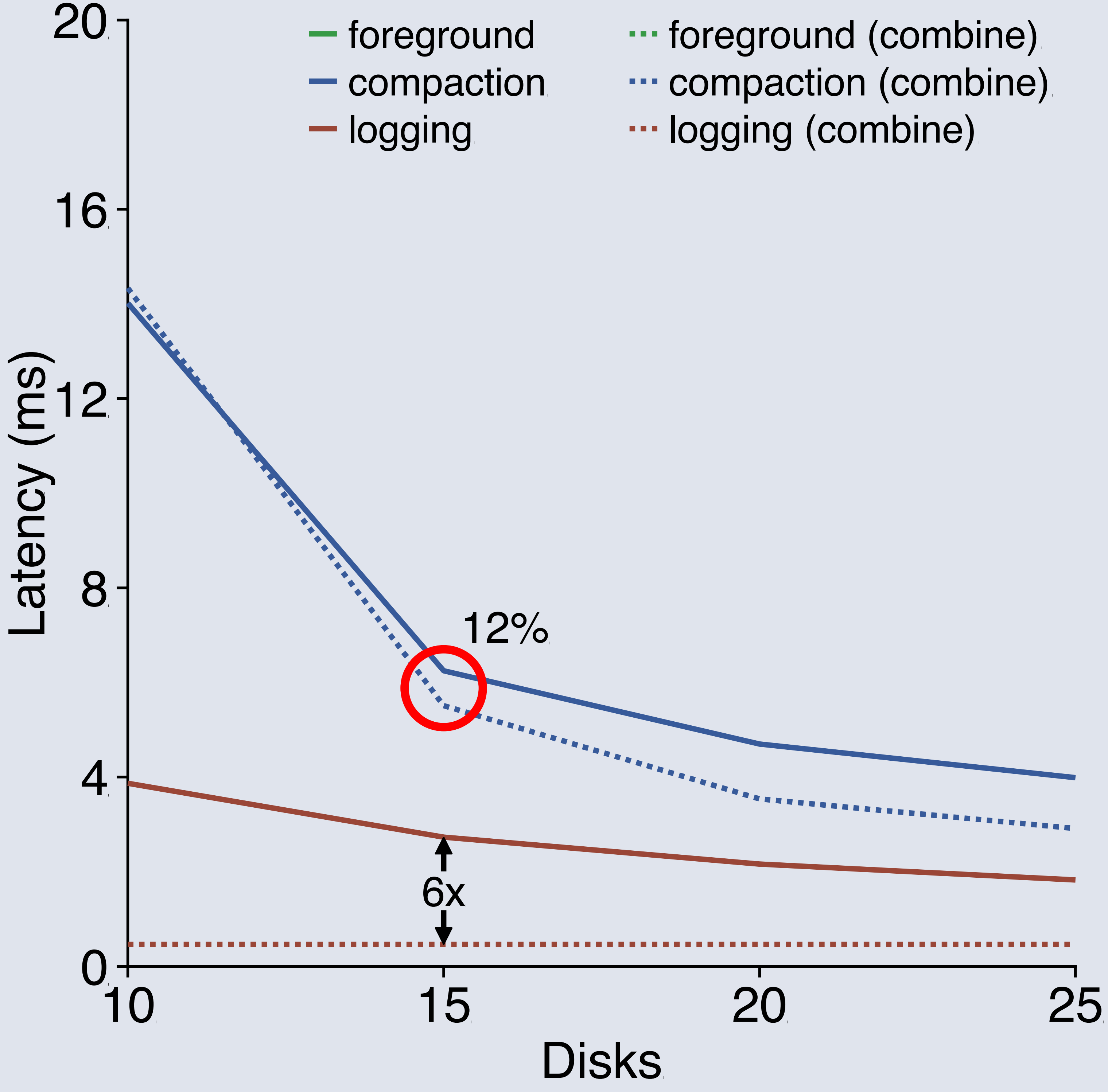
Log writes 6x faster (15 disks)

Combined Logging



Log writes 6x faster (15 disks)

Combined Logging

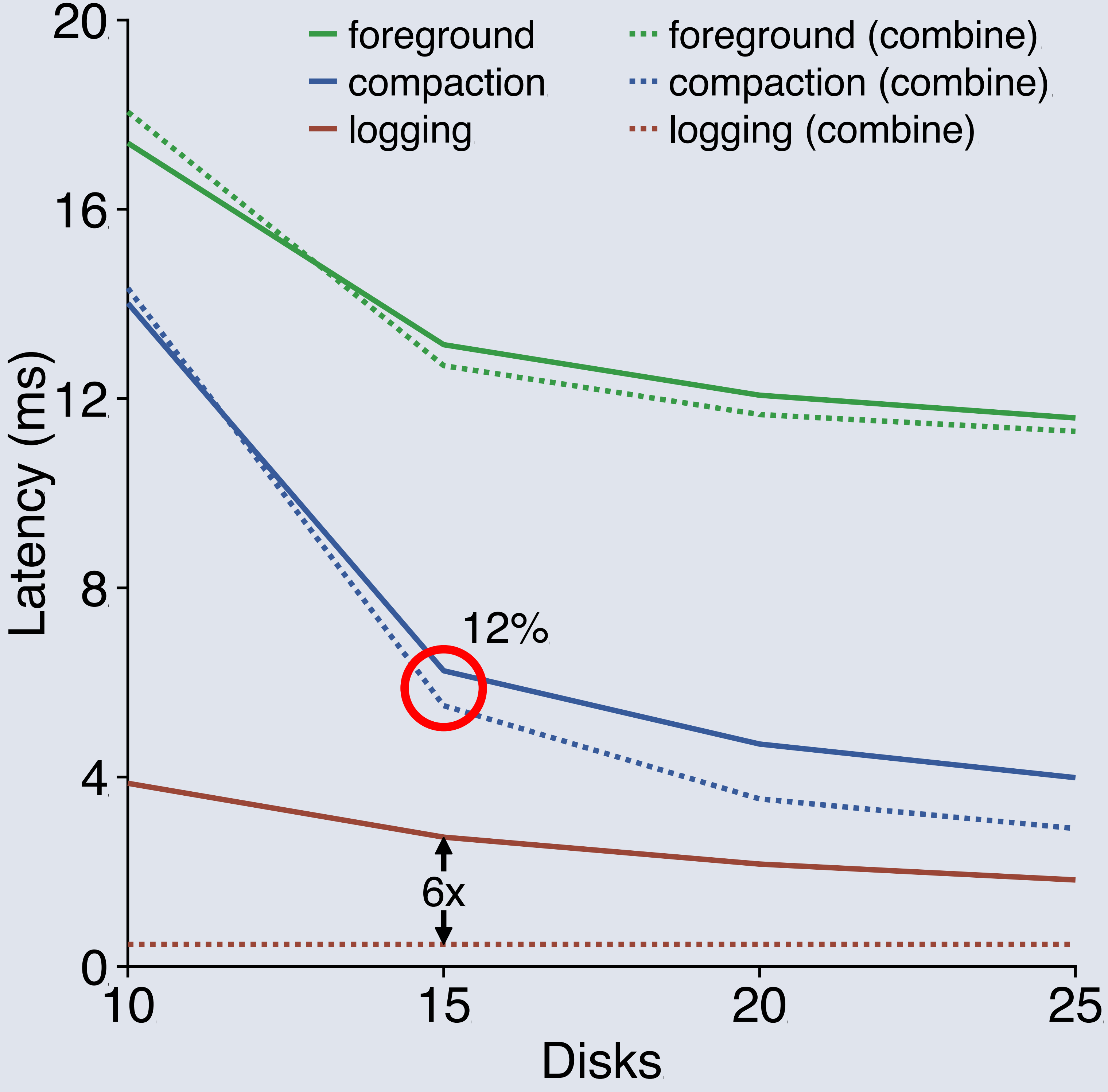


Log writes 6x faster (15 disks)

Compaction 12% faster

- Less competition with logs

Combined Logging

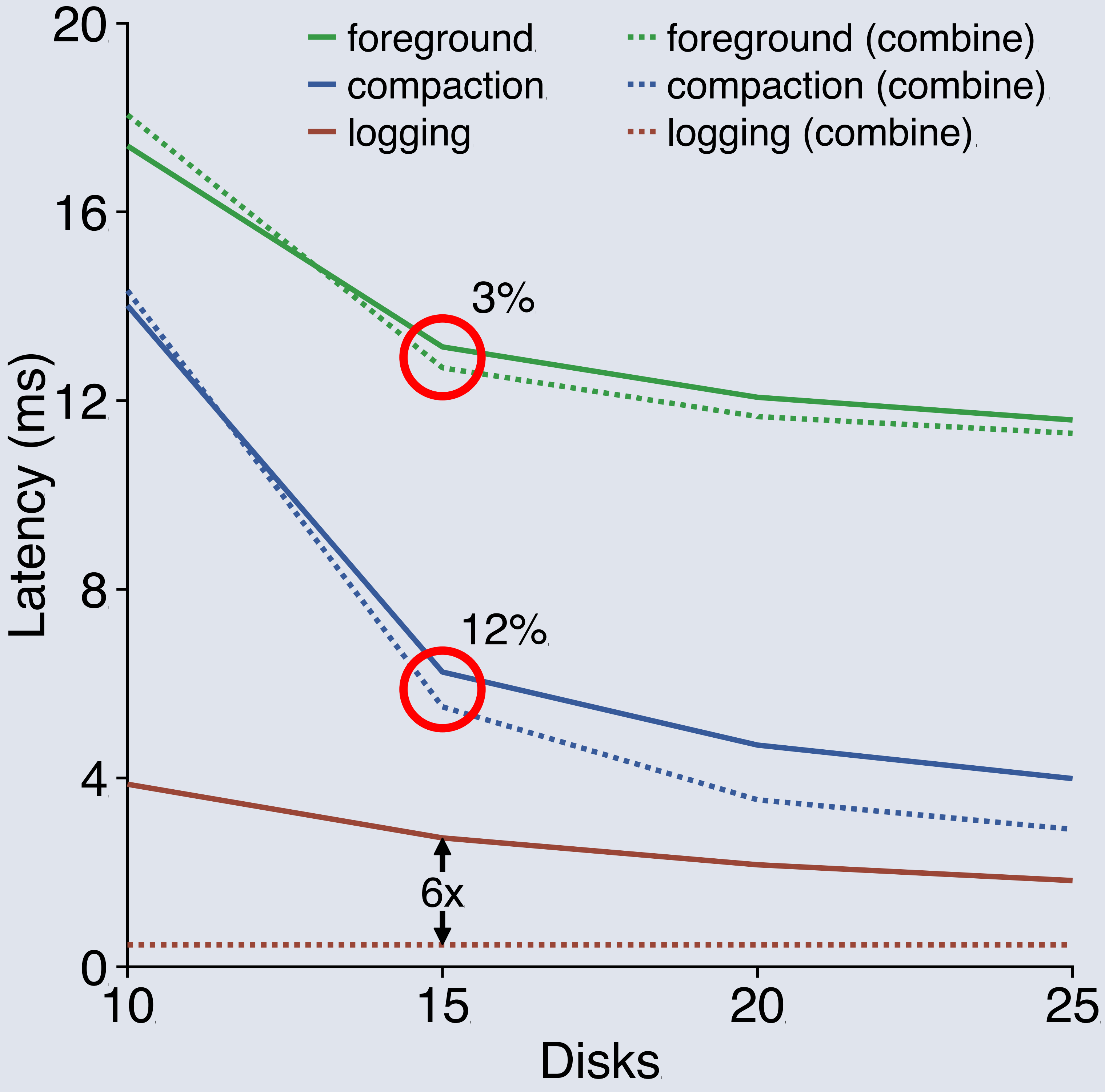


Log writes 6x faster (15 disks)

Compaction 12% faster

- Less competition with logs

Combined Logging



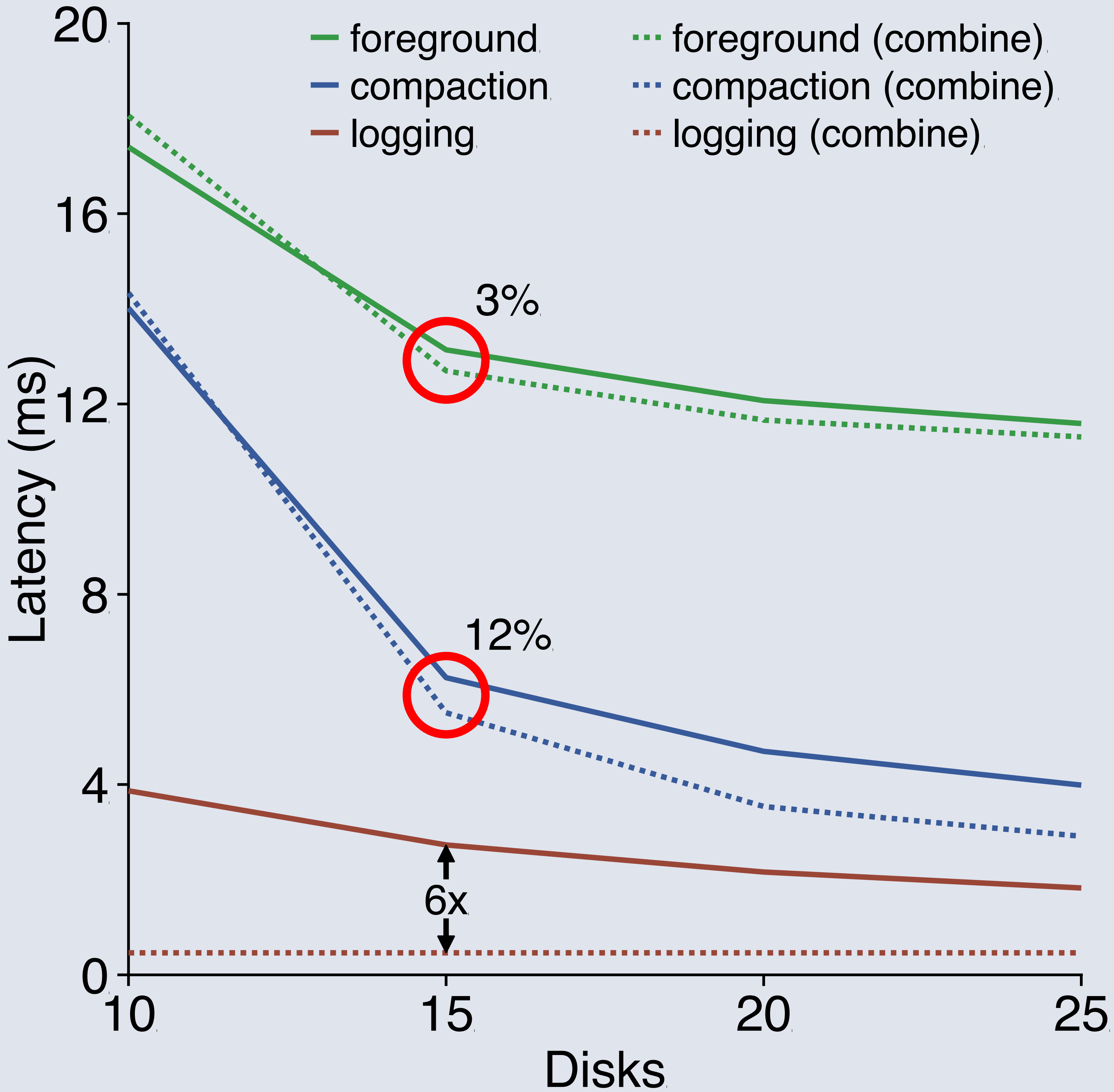
Log writes 6x faster (15 disks)

Compaction 12% faster

- Less competition with logs

Foreground reads 3% faster

Combined Logging



Log writes 6x faster (15 disks)

Compaction 12% faster

- Less competition with logs

Foreground reads 3% faster

Puts do not block currently

- Very useful if `put()`'s were to block until logs on disk

Facebook Messages Outline

Background

Workload Analysis

- I/O causes
- File size
- Sequentiality

Layer Integration

- Local compaction
- Combined logging

Discussion

Conclusion 1: New Workload on an Old Stack

Original GFS paper:

- "*high sustained bandwidth is more important than low latency*"
- "*multi-GB files are the common case*"

We find files are small and reads are random

- 50% of files <750KB
- 50% of read runs <130KB

Comparison to previous findings:

- Chen *et al.* found HDFS files to be **23 GB** at 90th percentile
- We find HDFS files to be **6.3 MB** at the 90th percentile

Conclusion 2: Layering is not Free

Layering “*proved to be vital for the verification and logical soundness*” of the THE operating system ~ Dijkstra

Layering is not free

- Over half of network I/O for replication is unnecessary

Layers can amplify writes, multiplicatively

- Logging overhead (10x) with replication (3x) => 30x write amp

Outline

Motivation: Modularity in Modern Storage

Overview: Types of Modularity

Library Study: Apple Desktop Applications

Layer Study: Facebook Messages

Microservice Study: Docker Containers

Slacker: a Lazy Docker Storage Driver

Conclusions

Container Popularity

docker
Rocket
CoreOS
kubernetes by Google
Google
facebook
twitter
amazon
heroku
spoon.net

What is a Container?

Goal: provide lightweight virtualization (compared to VMs)

Operating systems have long virtualized CPU and memory

But many resources have not been historically virtualized:

- file system mounts
- network
- host names
- IPC queues
- process IDs
- user IDs

What is a Container?

Goal: provide lightweight virtualization (compared to VMs)

Operating systems have long virtualized CPU and memory

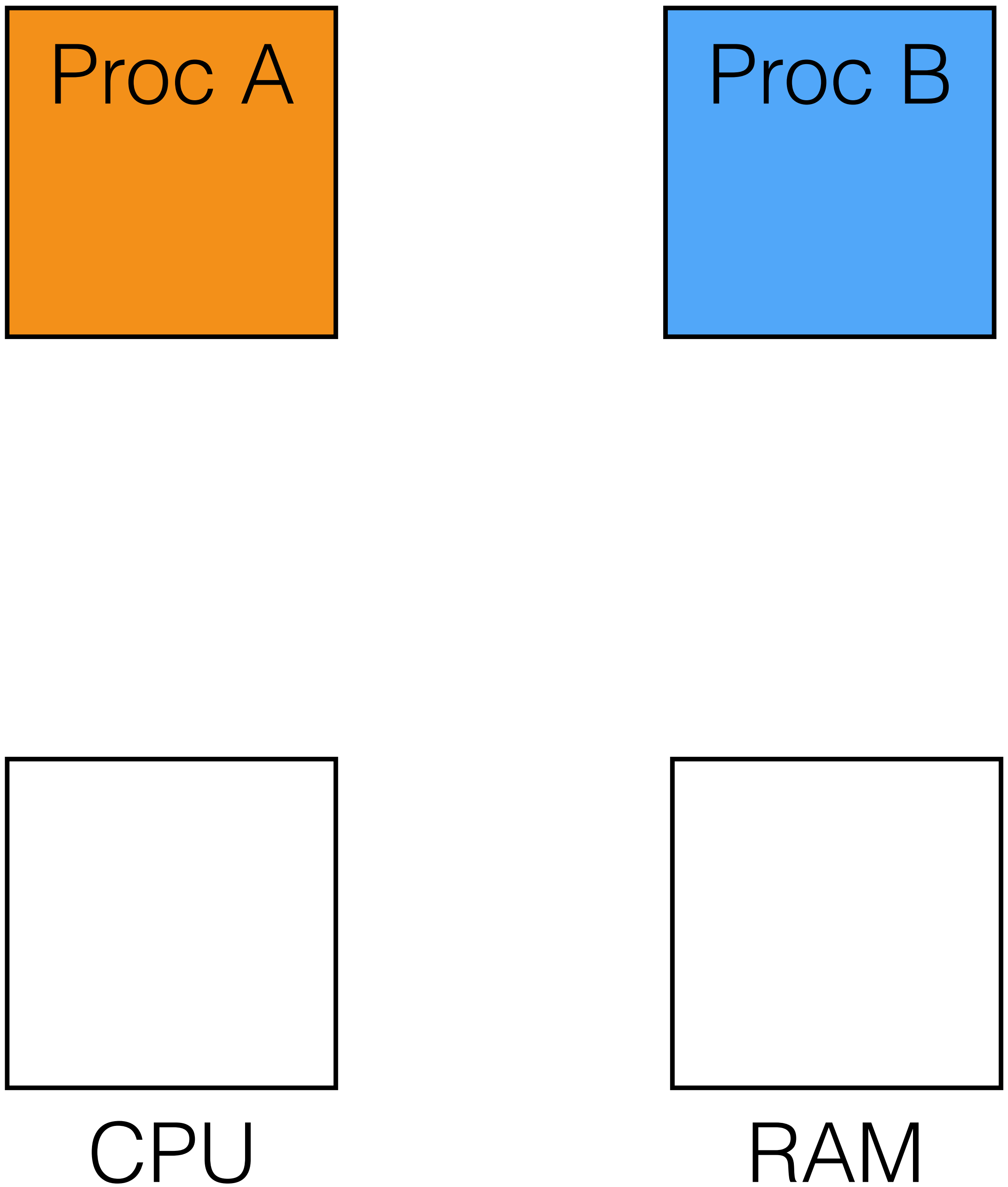
But many resources have not been historically virtualized:

- file system mounts
- network
- host names
- IPC queues
- process IDs
- user IDs

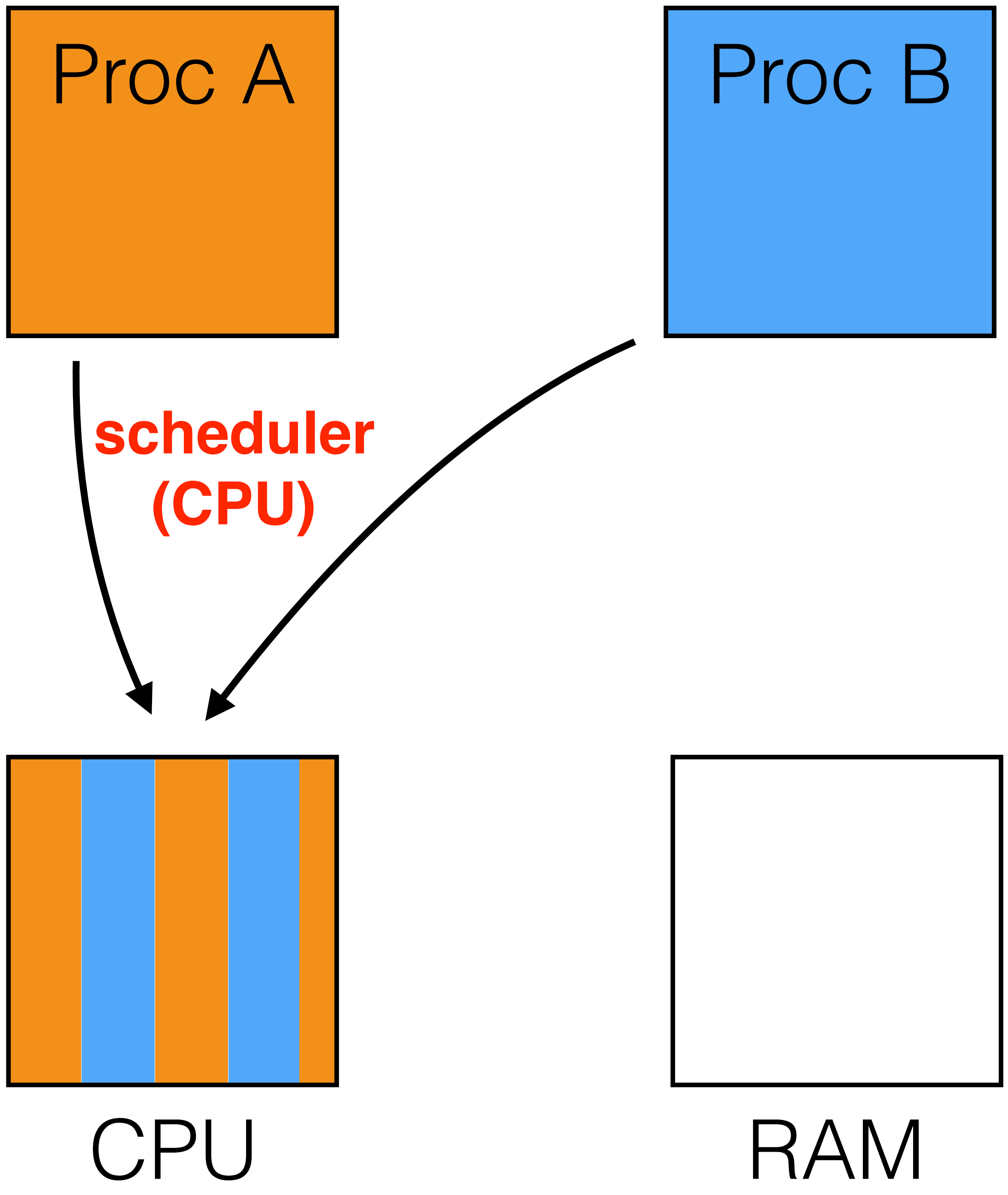
New namespaces are collectively called “containers”

- lightweight, like virtual memory
- old idea rebranded (Plan 9 OS)

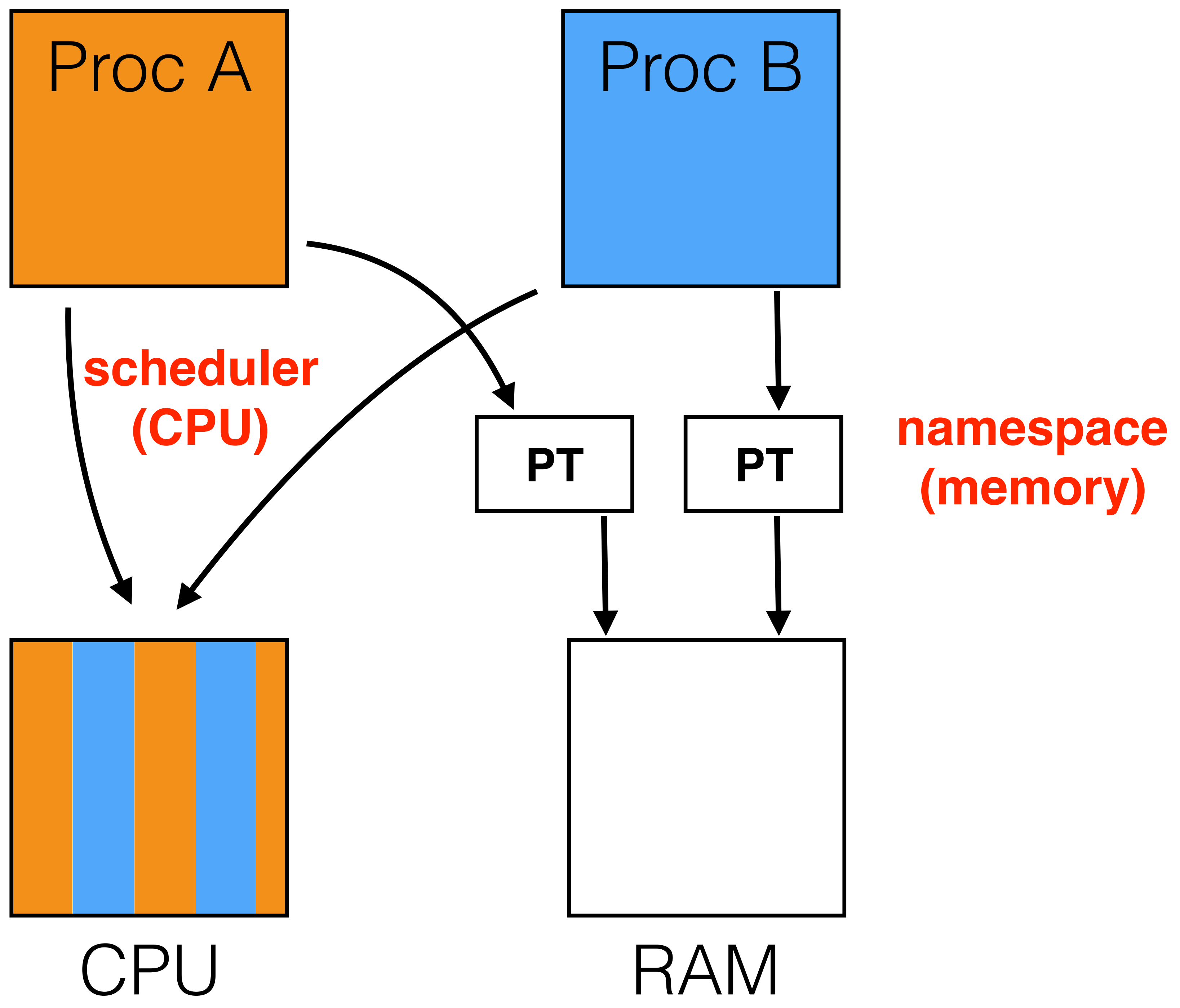
OS-Level Virtualization



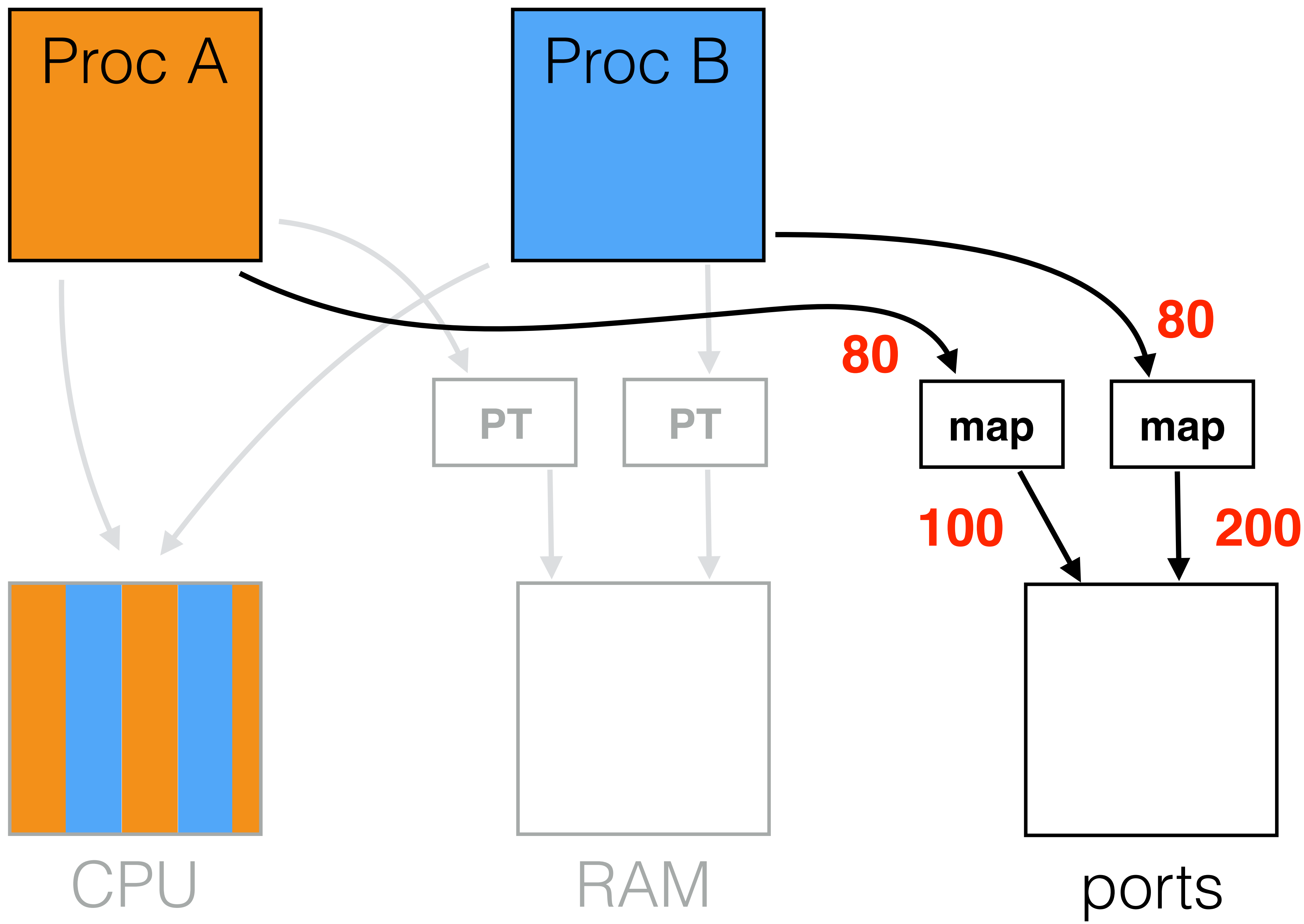
OS-Level Virtualization



OS-Level Virtualization



OS-Level Virtualization



Implications for Microservices

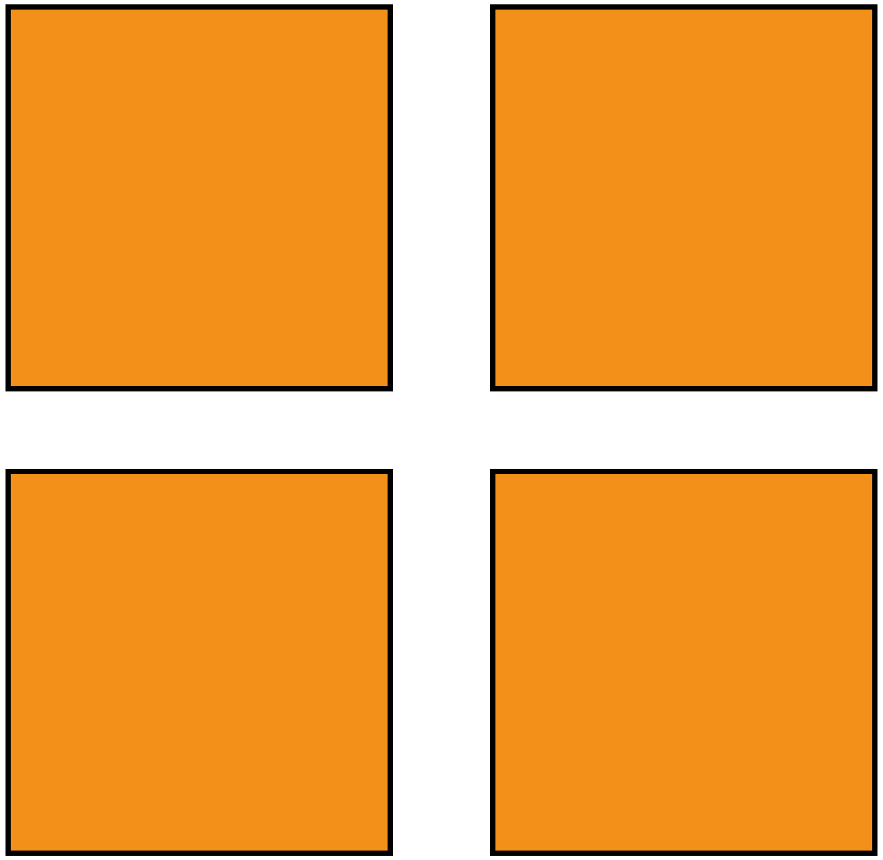
Decomposing applications is an old technique.

How fine grained should the components be?

Implications for Microservices

Decomposing applications is an old technique.

How fine grained should the components be?

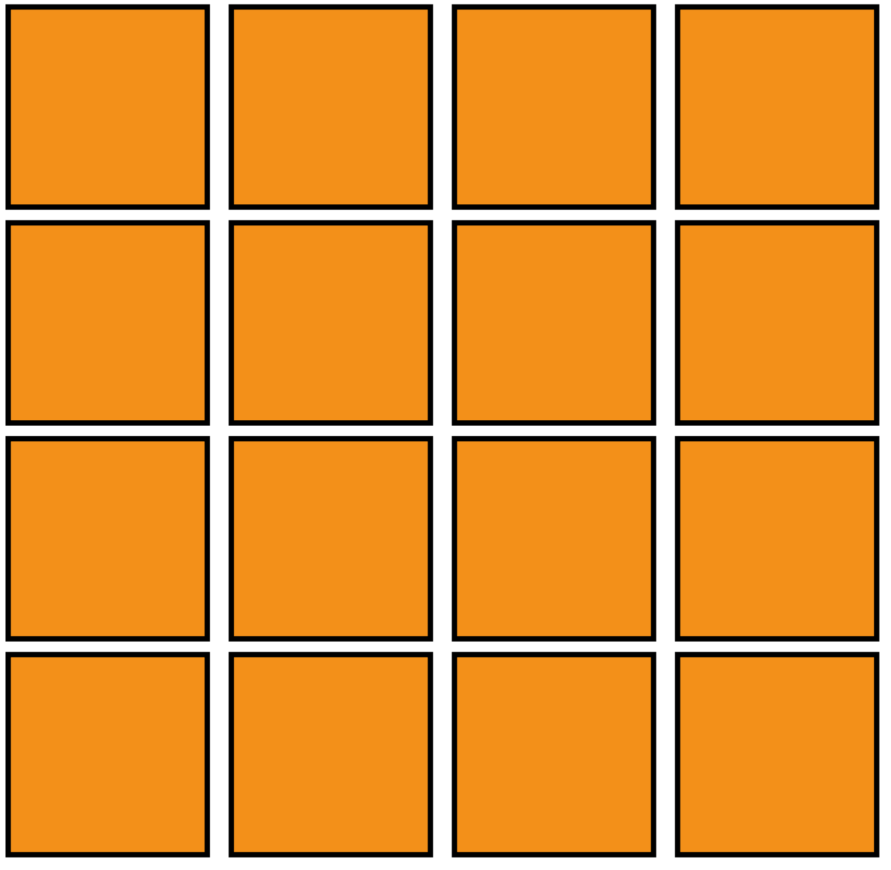


coarse if sandboxes are **expensive**
(e.g., **virtual machines** are used)

Implications for Microservices

Decomposing applications is an old technique.

How fine grained should the components be?

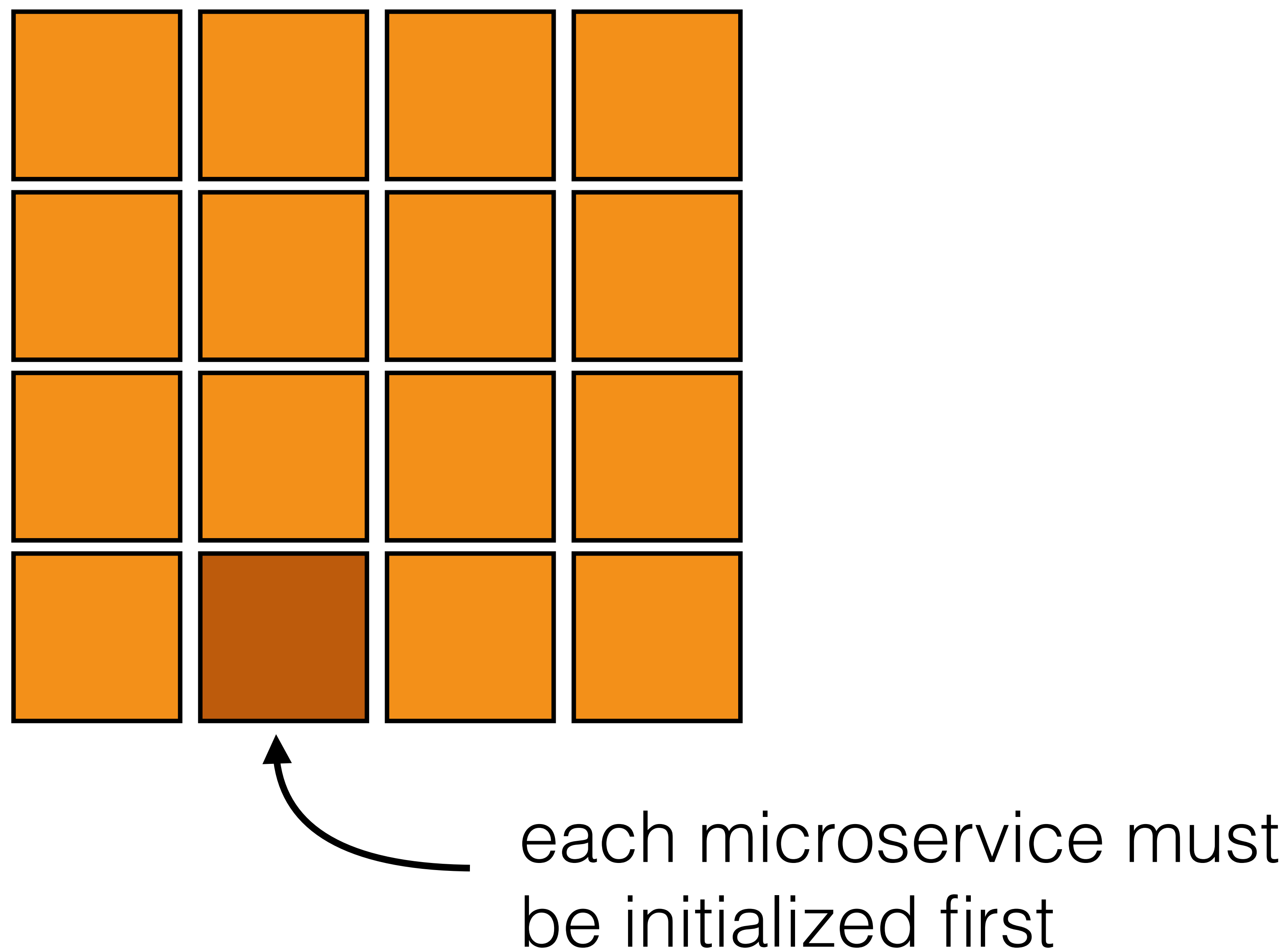


fine if sandboxes are **cheap**
(e.g., **containers** are used)

Implications for Microservices

Decomposing applications is an old technique.

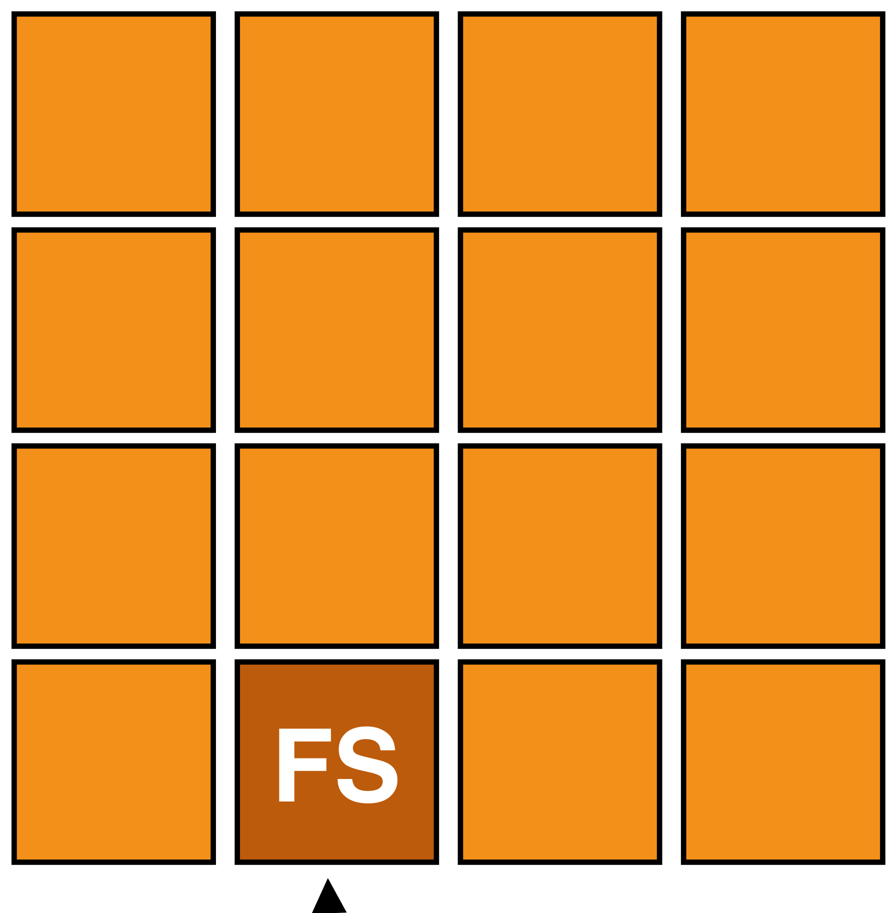
How fine grained should the components be?



Implications for Microservices

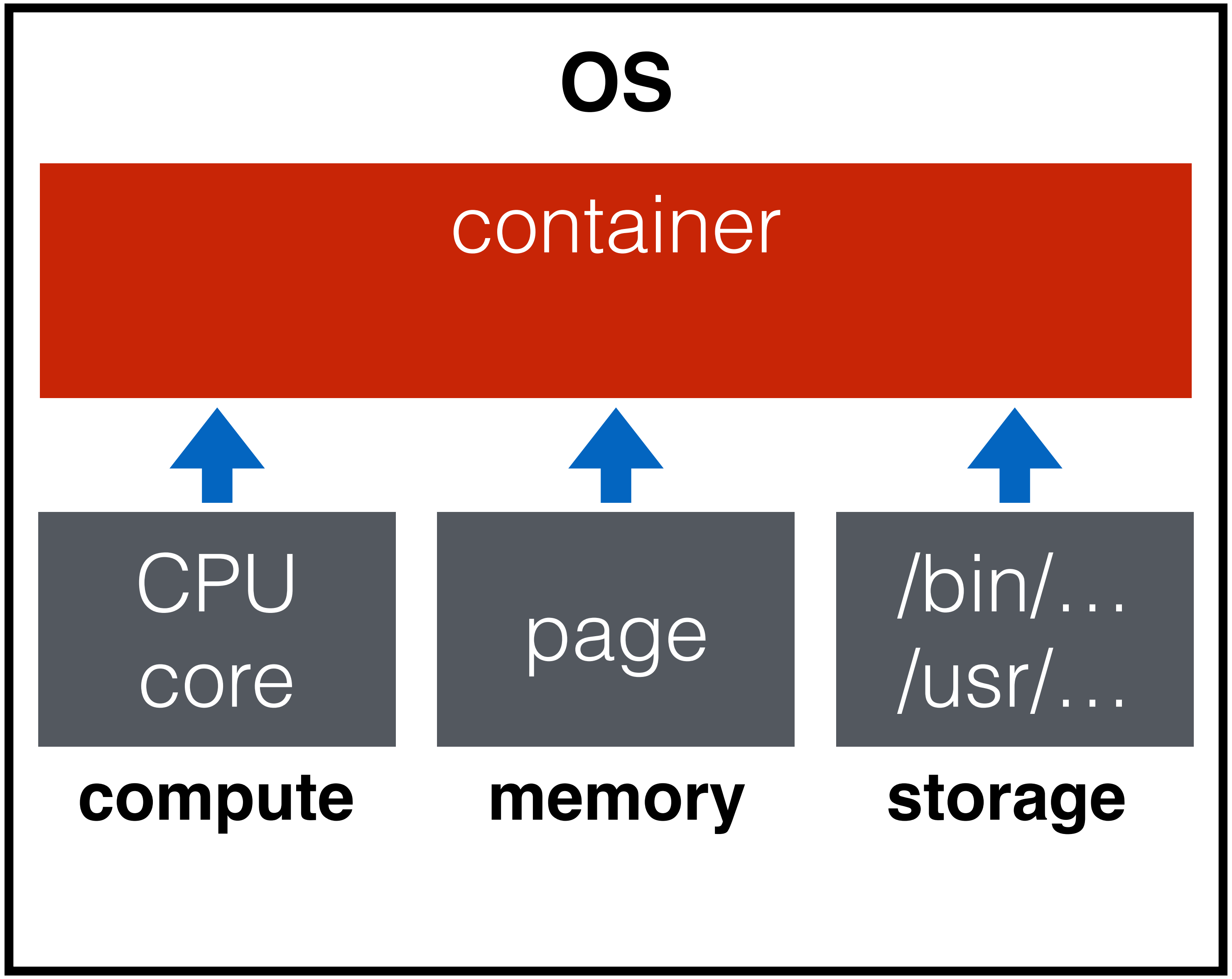
Decomposing applications is an old technique.

How fine grained should the components be?

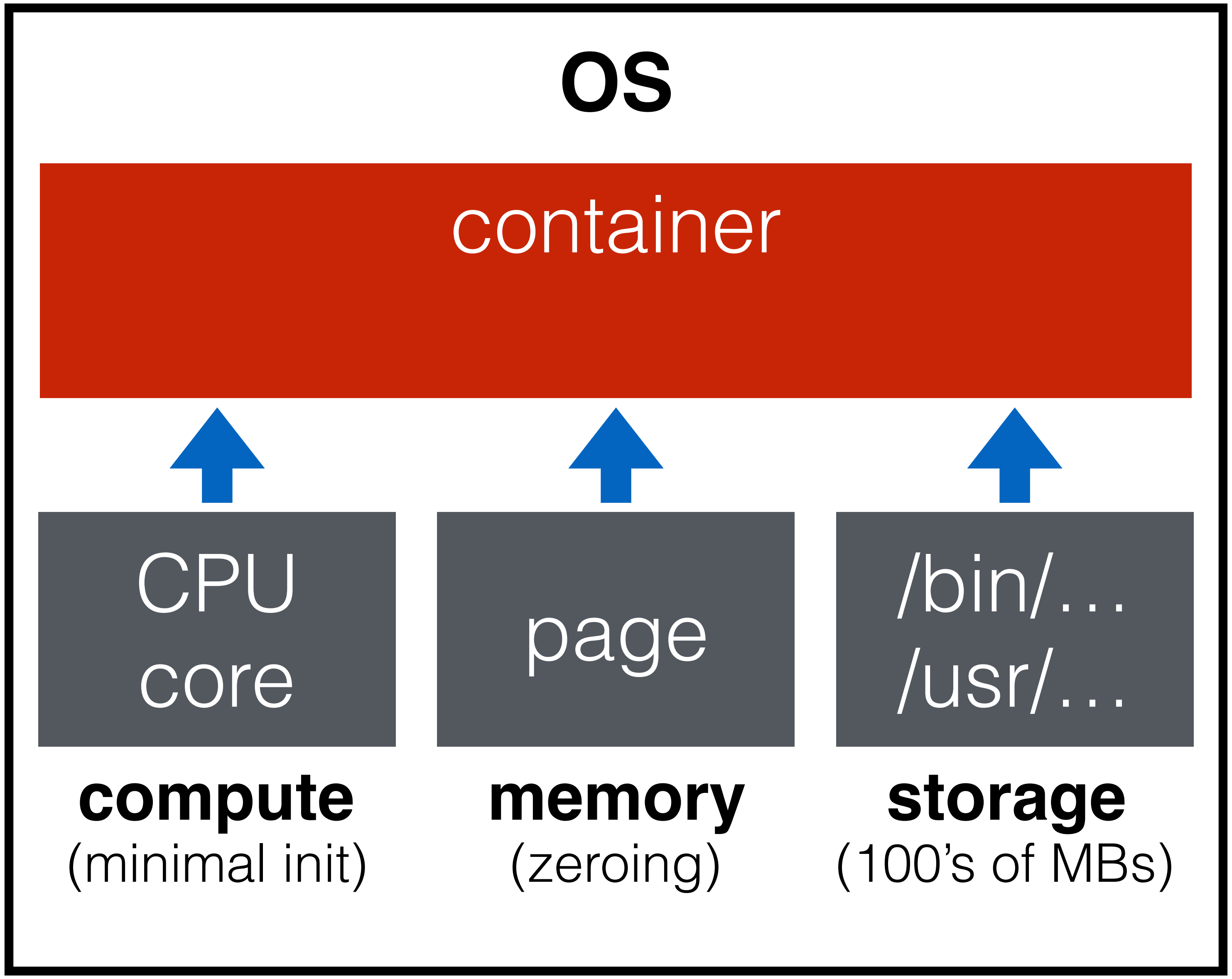


file system provisioning is an interesting problem

Resource Initialization



Resource Initialization



Theory and Practice

Theory: containers are lightweight

- just like starting a process!

Theory and Practice

Theory: containers are lightweight

- just like starting a process!

Practice: container startup is slow

- Large-scale cluster management at Google with Borg [1]
- **25 second** median startup
- 80% of time spent on package installation
- contention for disk a bottleneck
- this problem *"has received and continues to receive significant attention"*

[1] Large-scale cluster management at Google with Borg.
http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43438.pdf

Theory and Practice

Theory: containers are lightweight

- just like starting a process!

Practice: container startup is slow

- Large-scale cluster management at Google with Borg [1]
- **25 second** median startup
- 80% of time spent on package installation
- contention for disk a bottleneck
- this problem *"has received and continues to receive significant attention"*

Startup time matters

- flash crowds
- load balance
- interactive development

[1] Large-scale cluster management at Google with Borg.
http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43438.pdf

Docker Outline

Container and Microservice Background

Docker Background

HelloBench Workload

Analysis

- Data distribution across layers
- Access patterns

Docker Background

Deployment tool built on containers

An application is defined by a file-system image

- application binary
- shared libraries
- etc.

Version-control model

- **extend** images by committing additional files
- **deploy** applications by pushing/pulling images

Containers as Repos

LAMP stack example

- commit 1: **L**inux packages (e.g., Ubuntu)
- commit 2: **A**pache
- commit 3: **M**ySQL
- commit 4: **P**HP

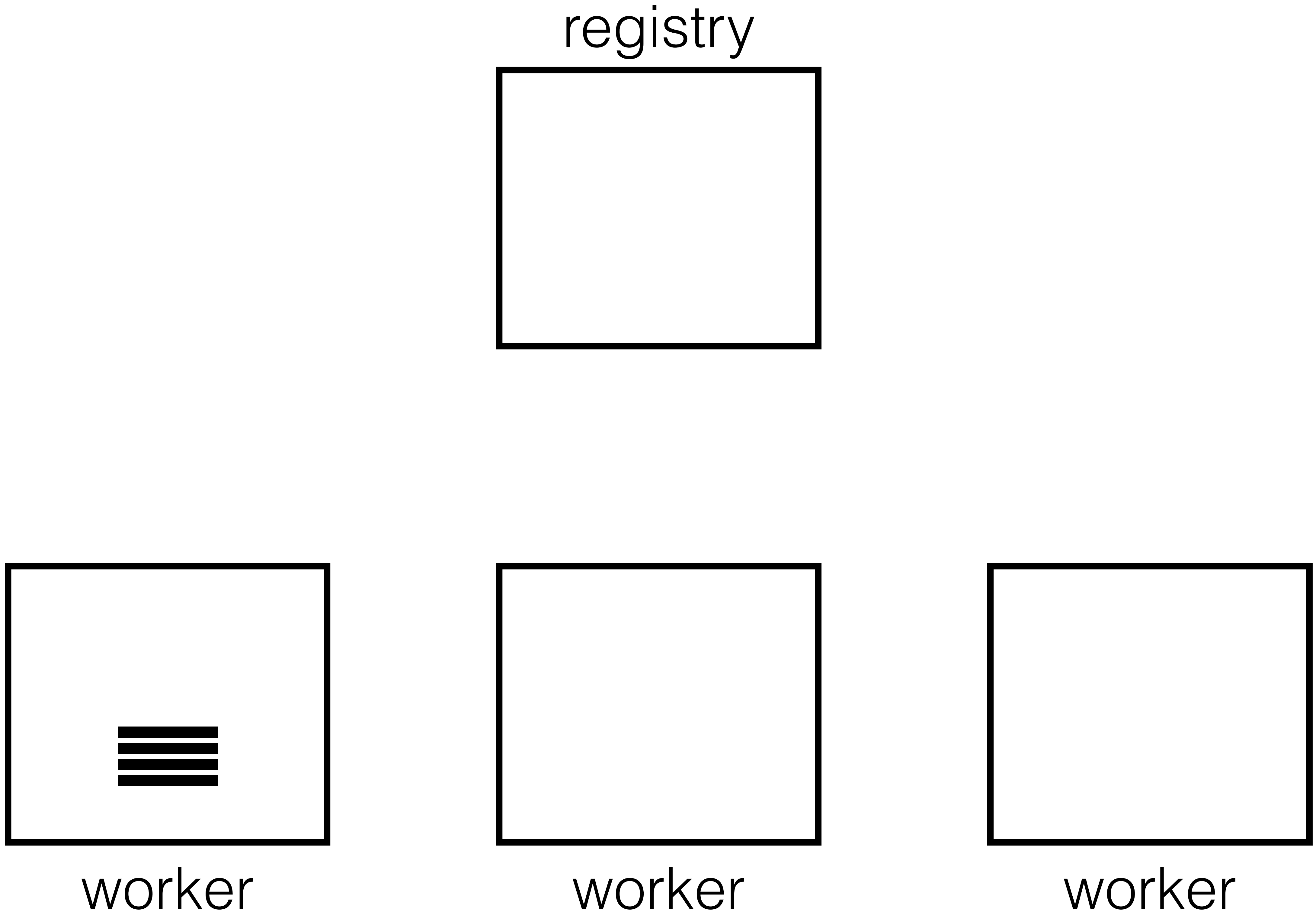
Docker “layer”

- commit
- container scratch space

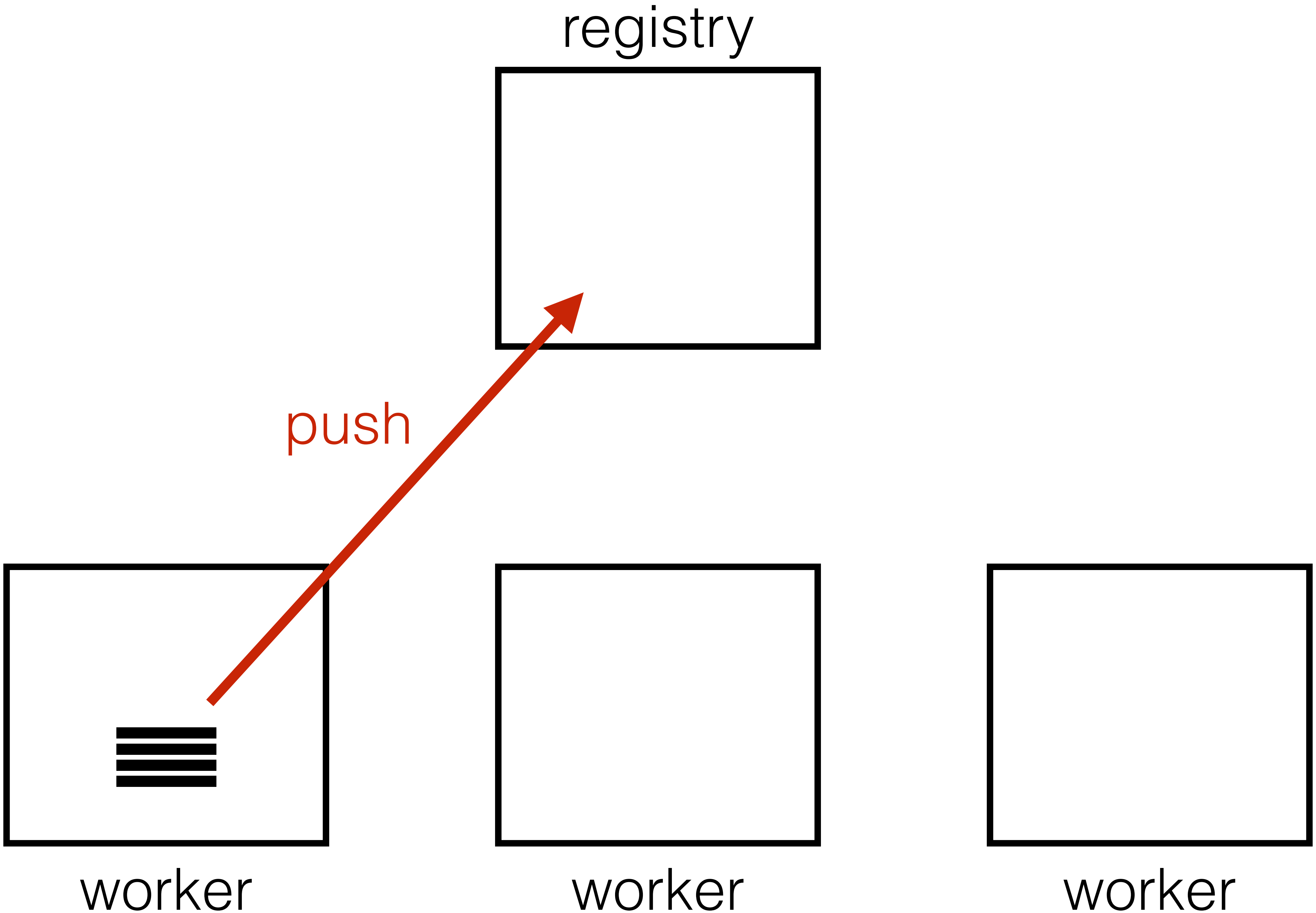
Central registries

- Docker HUB
- private registries

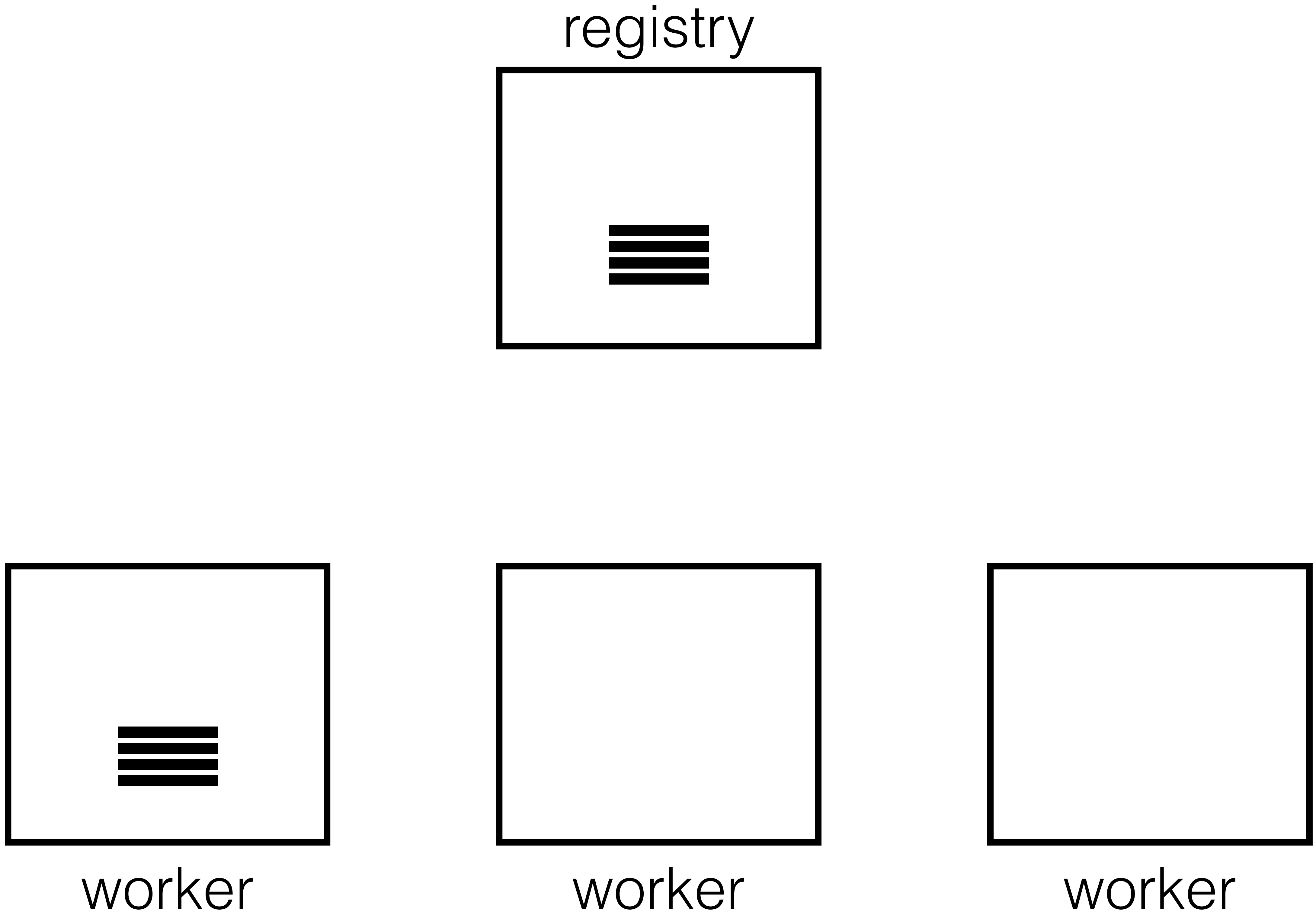
Push, Pull, Run



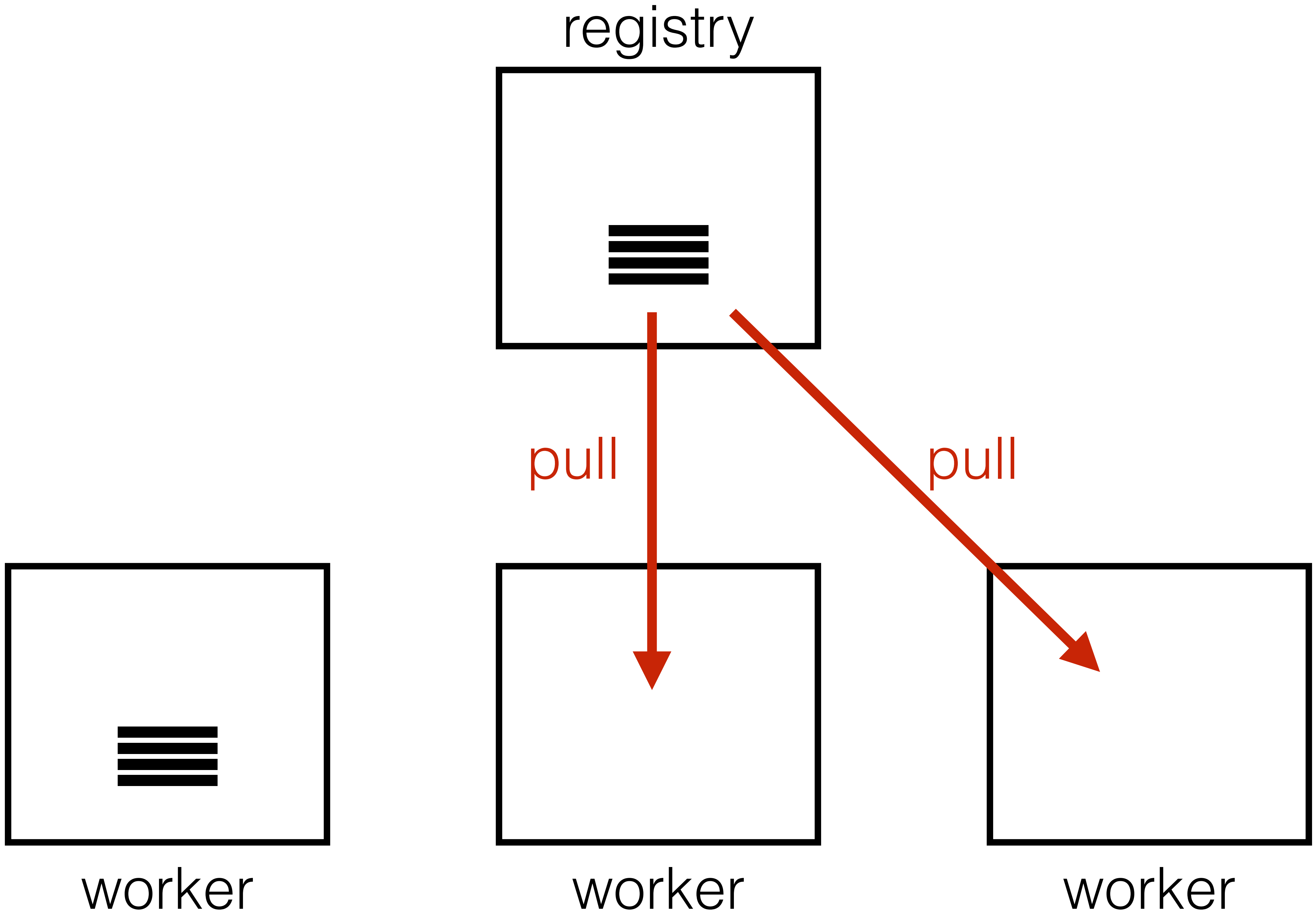
Push, Pull, Run



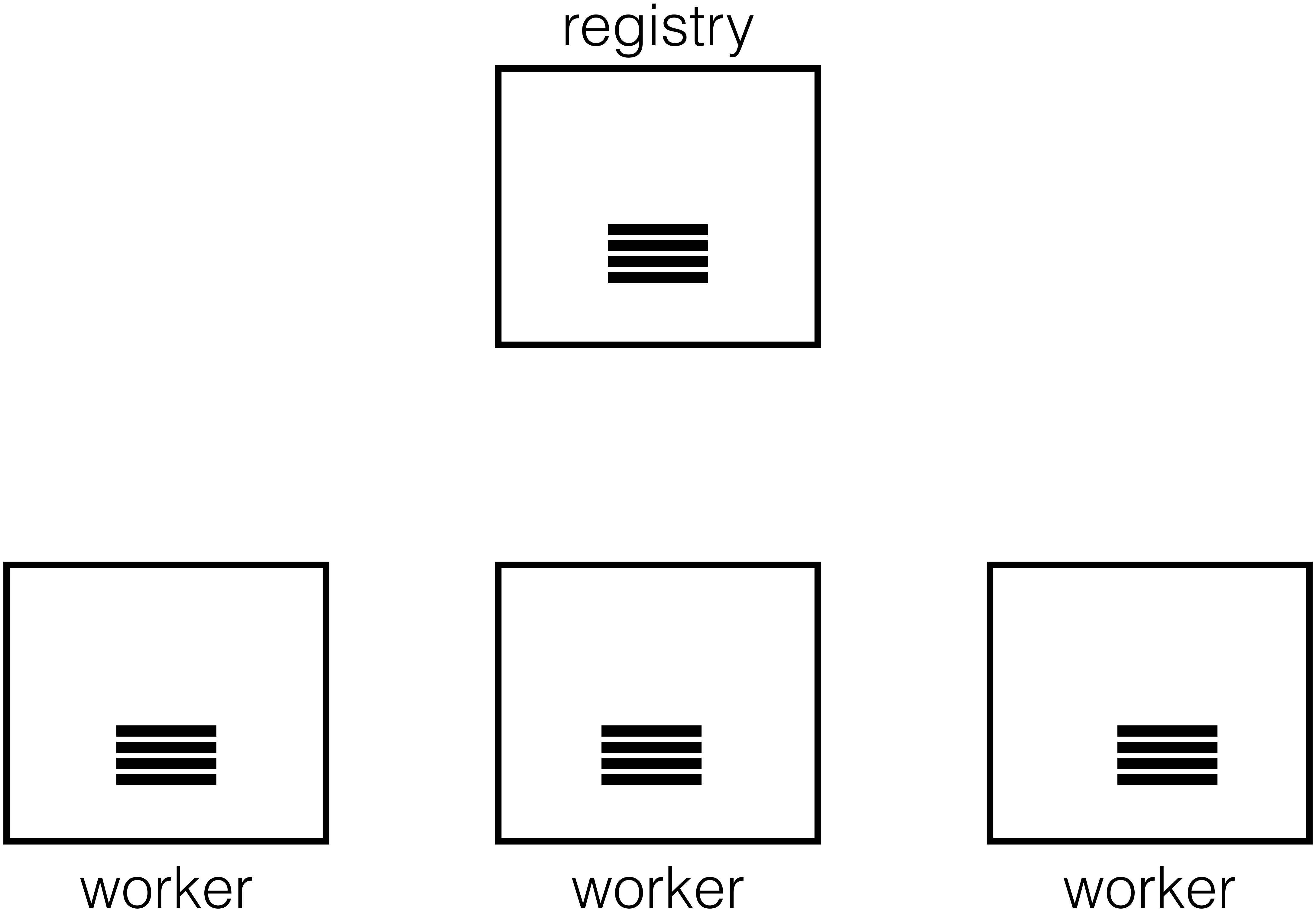
Push, Pull, Run



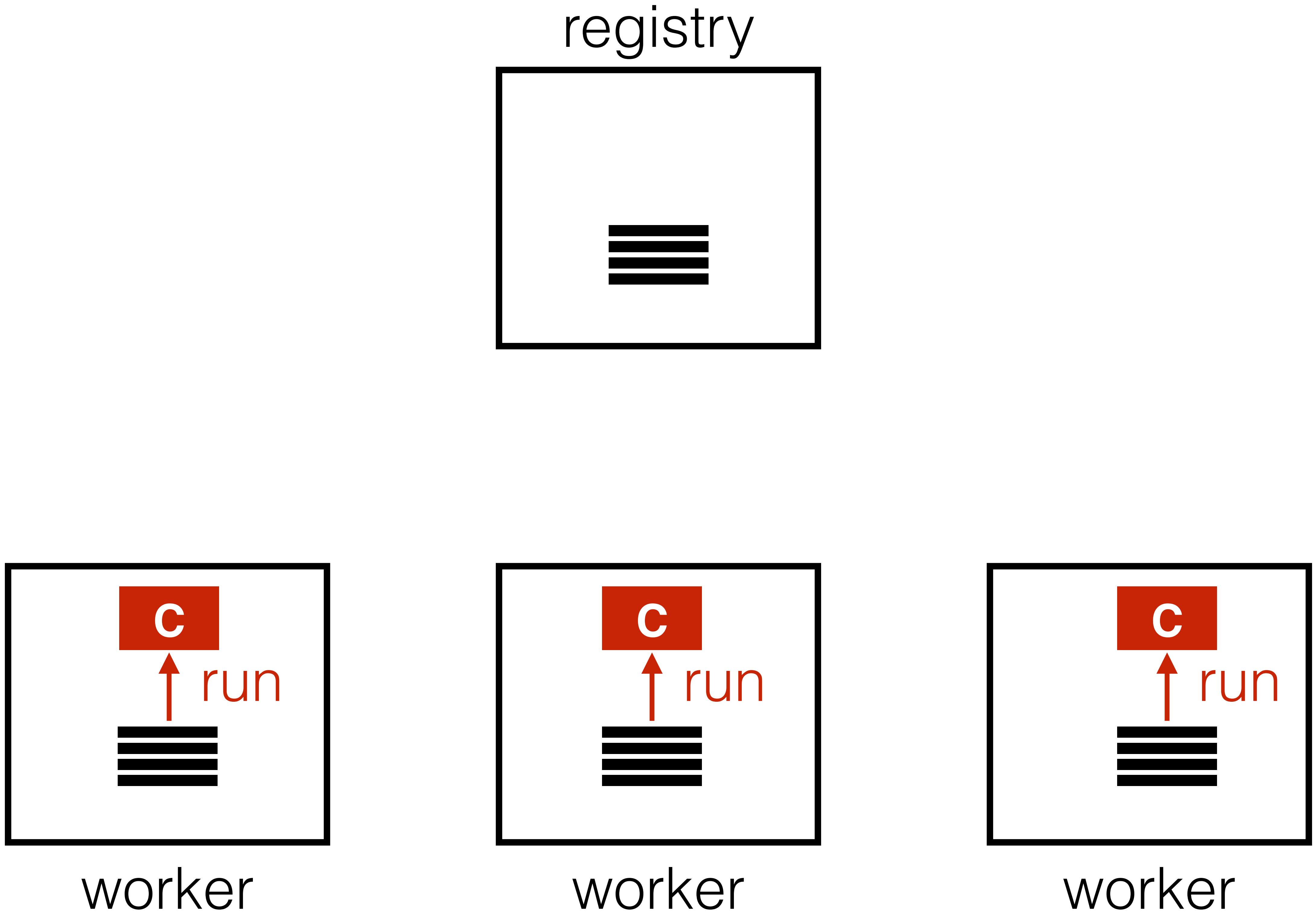
Push, Pull, Run



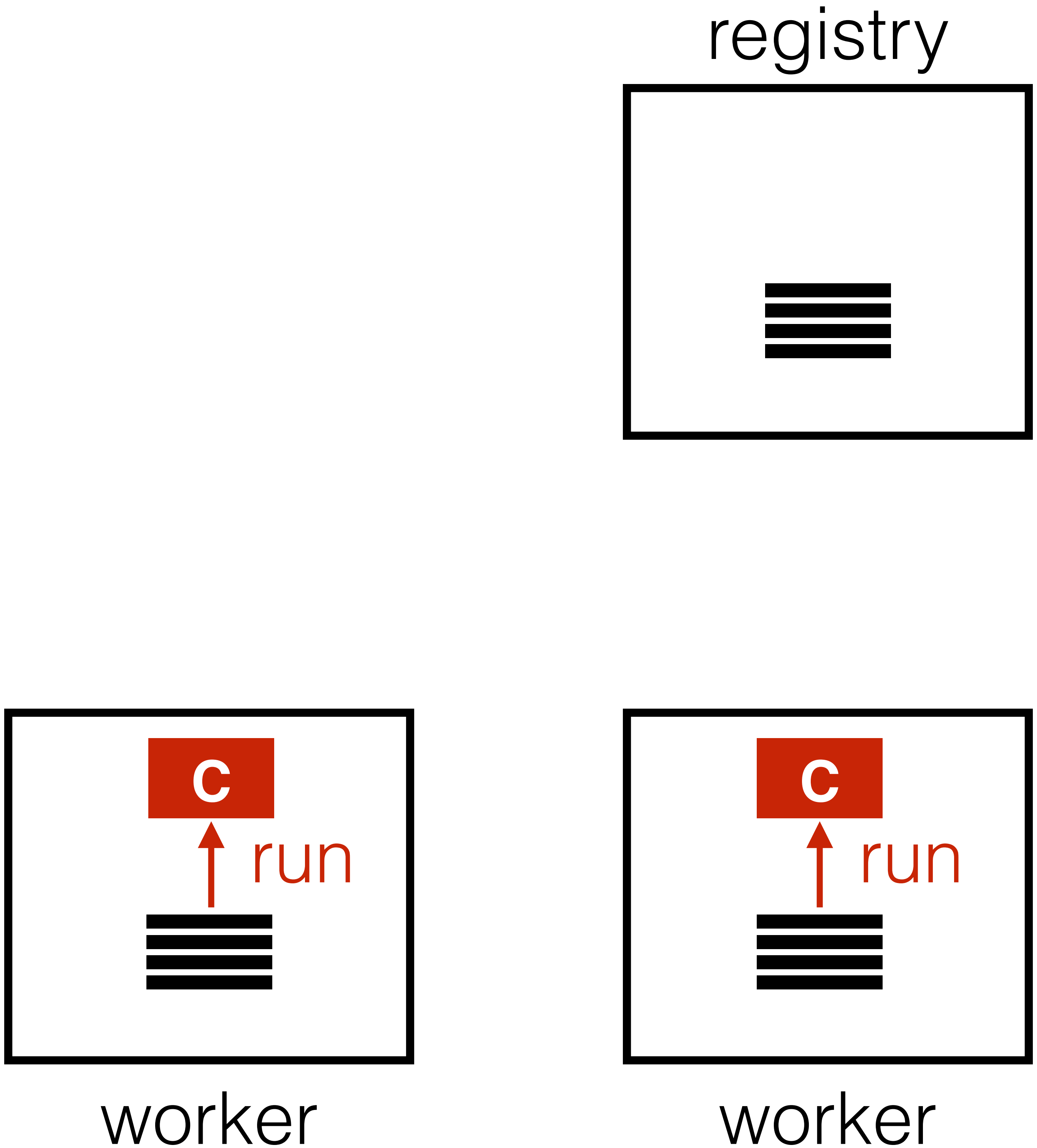
Push, Pull, Run



Push, Pull, Run



Push, Pull, Run



need a new benchmark to measure Docker push, pull, and run operations.

Docker Outline

Container and Microservice Background

Docker Background

HelloBench Workload

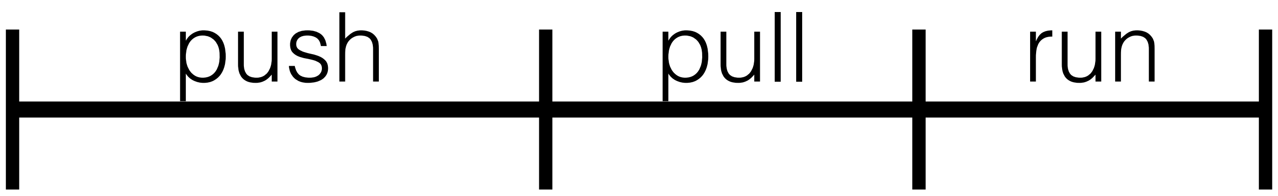
Analysis

- Data distribution across layers
- Access patterns

HelloBench

Goal: stress container startup

- including push/pull
- **57 container images** from Docker HUB
- run simple “hello world”-like task
- wait until it’s done/ready



HelloBench

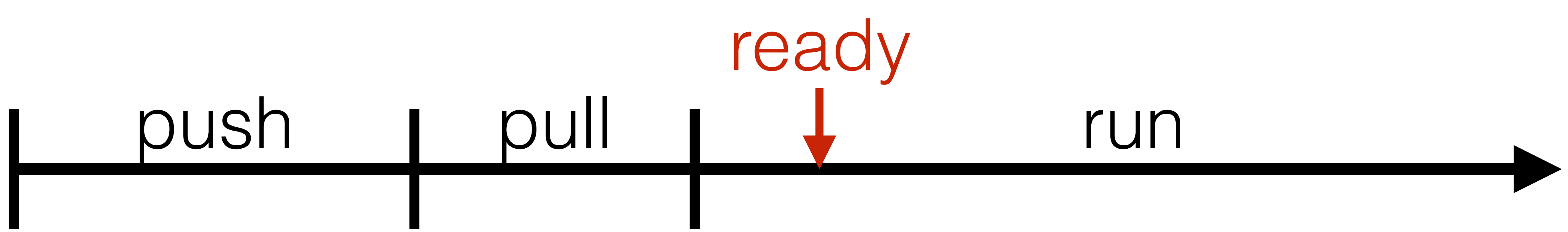
Goal: stress container startup

- including push/pull
- **57 container images** from Docker HUB
- run simple “hello world”-like task
- wait until it’s done/ready

HelloBench

Goal: stress container startup

- including push/pull
- **57 container images** from Docker HUB
- run simple “hello world”-like task
- wait until it’s done/ready



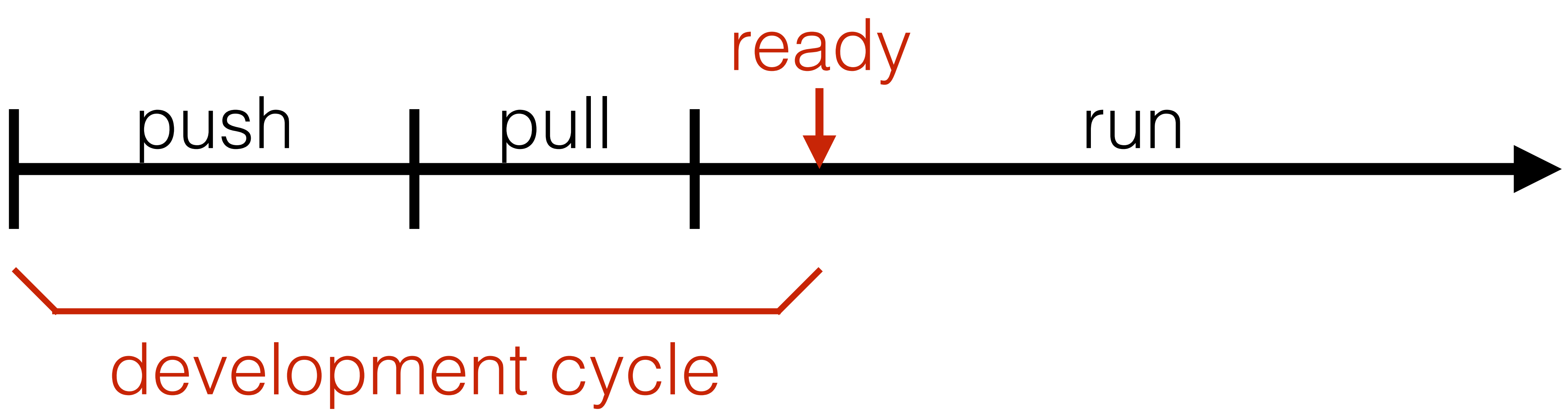
HelloBench

Goal: stress container startup

- including push/pull
- **57 container images** from Docker HUB
- run simple “hello world”-like task
- wait until it’s done/ready

Development cycle

- distributed programming/testing



HelloBench

Goal: stress container startup

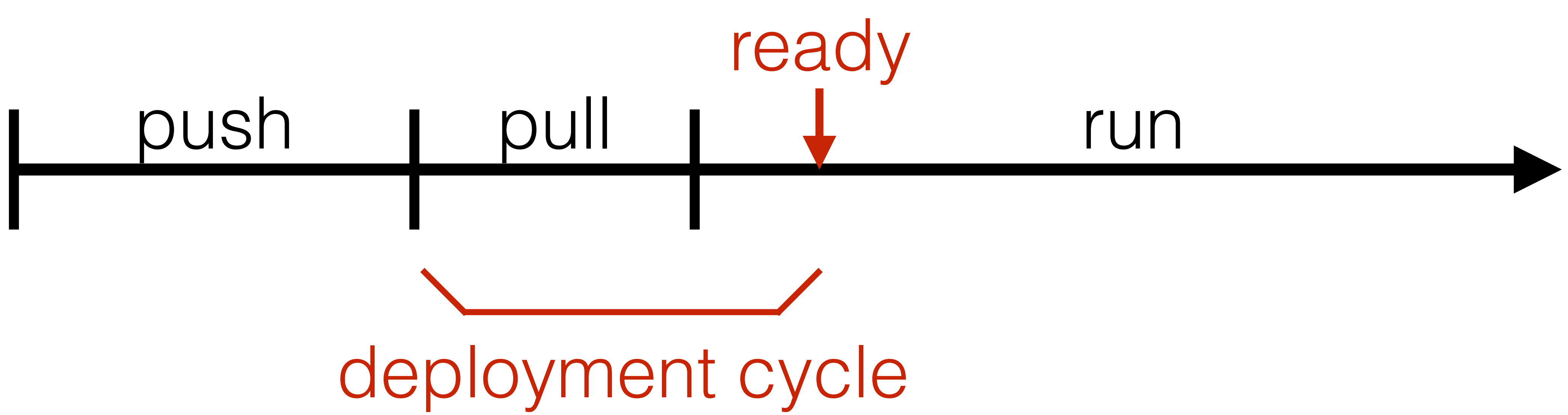
- including push/pull
- **57 container images** from Docker HUB
- run simple “hello world”-like task
- wait until it’s done/ready

Development cycle

- distributed programming/testing

Deployment cycle

- flash crowds, rebalance



Workload Categories

Language
clojure
gcc
golang
haskell
hylang
java
jruby
julia
mono
perl
php
pypy
python
r-base
rakudo-star
ruby
thrift

Linux Distro
alpine
busybox
centos
cirros
crux
debian
fedora
mageia
opensuse
oraclelinux
ubuntu
ubuntu-debootstrap
ubuntu-upstart

Database
cassandra
crate
elasticsearch
mariadb
mongo
mysql
percona
postgres
redis
rethinkdb

Web Framework
django
iojs
node
rails

Web Server
glassfish
httpd
jetty
nginx
php-zendserver
tomcat

Other
drupal
ghost
hello-world
jenkins
rabbitmq
registry
sonarqube

Docker Outline

Container and Microservice Background

Docker Background

HelloBench Workload

Analysis

- Data distribution across layers
- Access patterns

Analysis Questions

How is data distributed across Docker layers?

How much image data is needed for container startup?

Analysis Questions

How is data distributed across Docker layers?

How much image data is needed for container startup?

HelloBench images

- **circle**: commit
- **red**: image

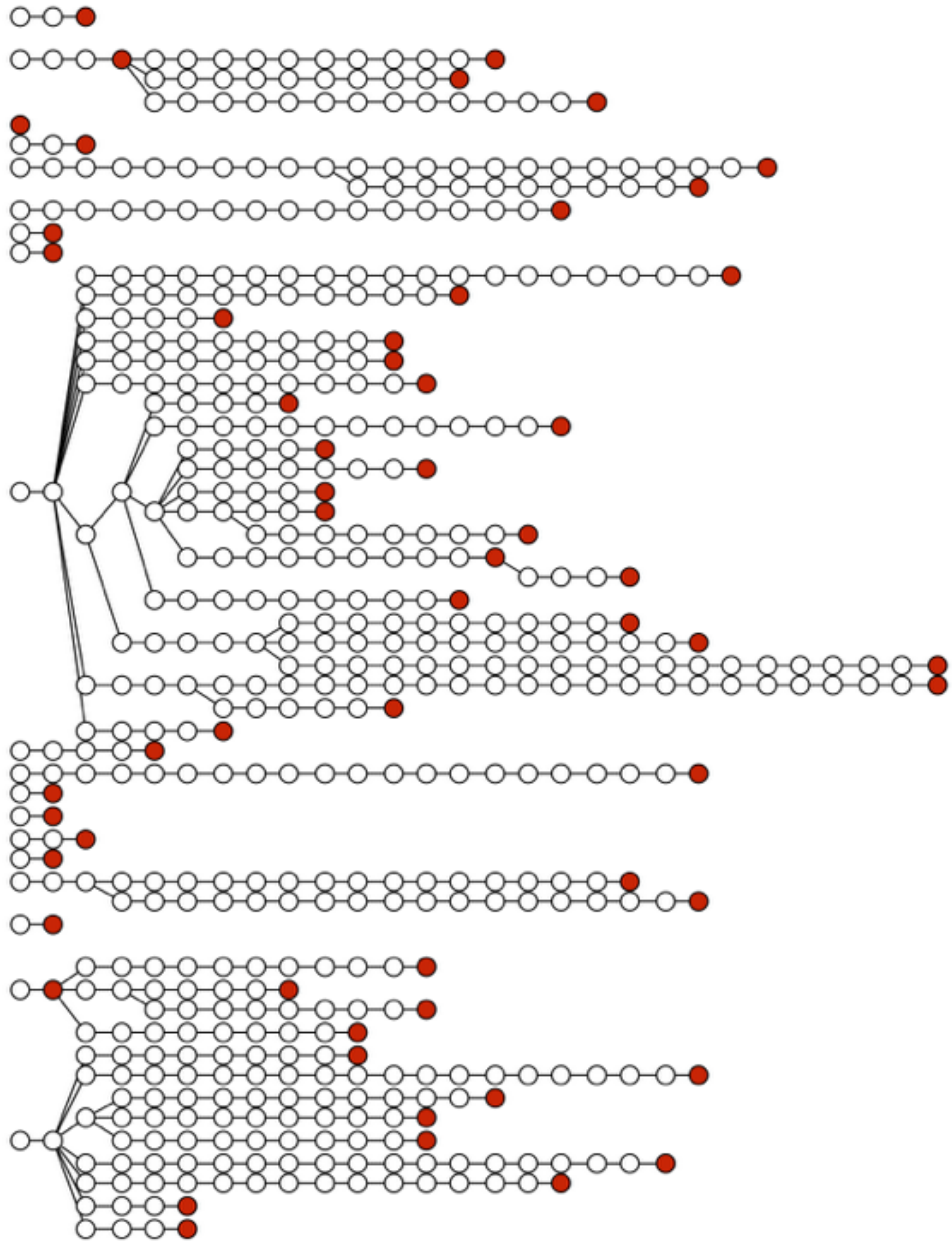


Image Data Depth

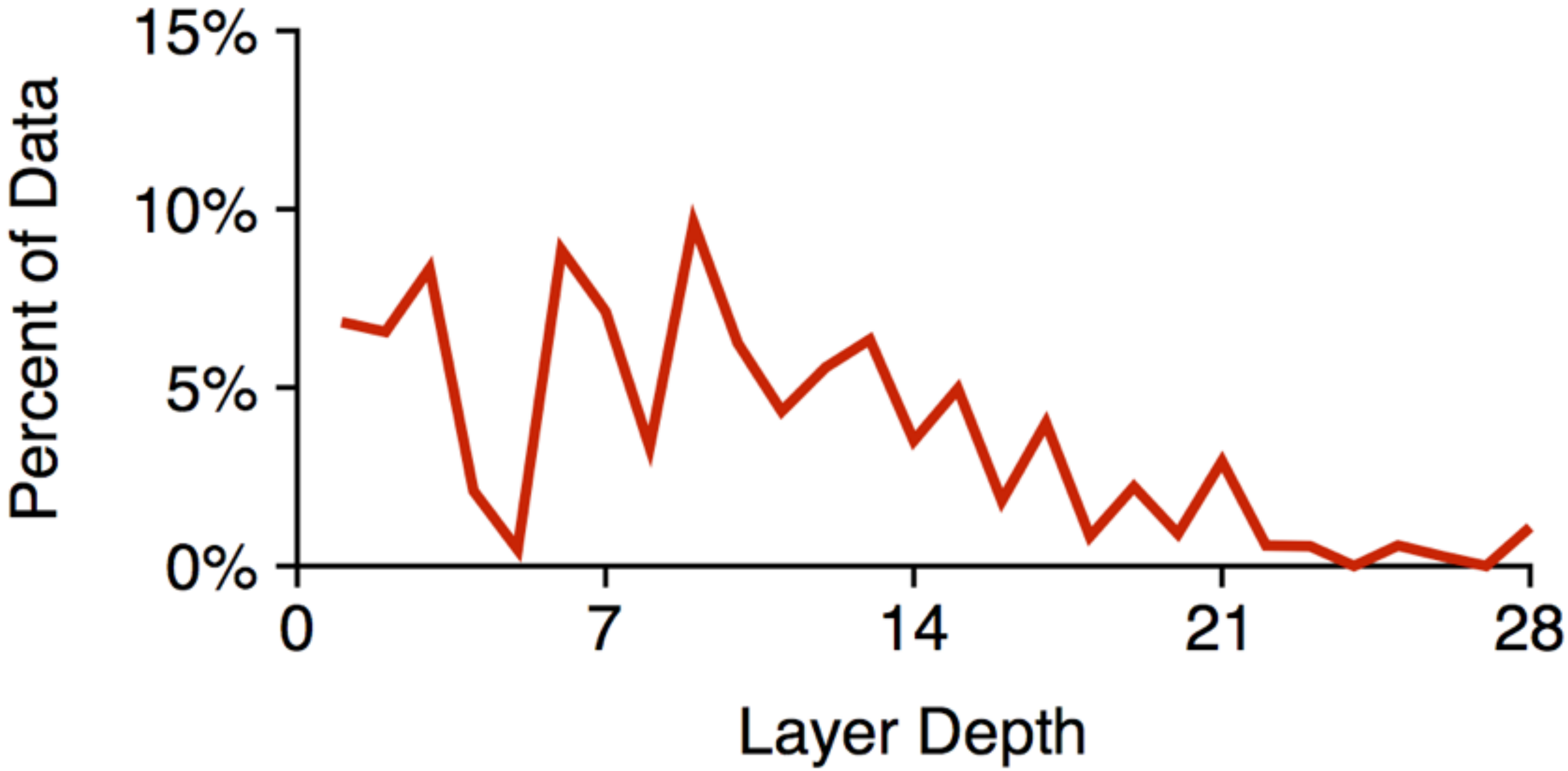
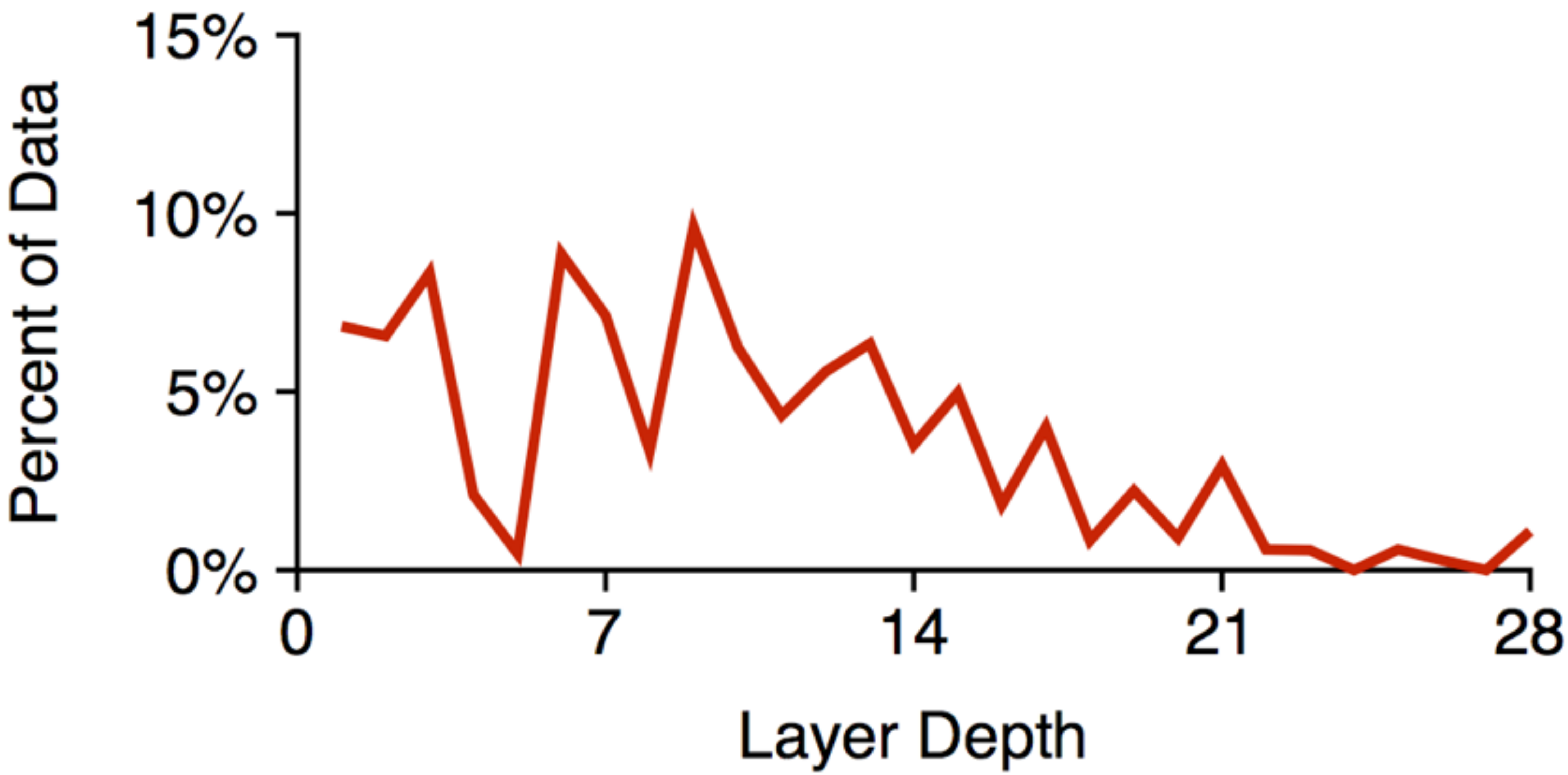


Image Data Depth



half of data is at depth 9+

Analysis Questions

How is data distributed across Docker layers?

- half of data is at depth 9+
- **design implication**: flatten layers at runtime

How much image data is needed for container startup?

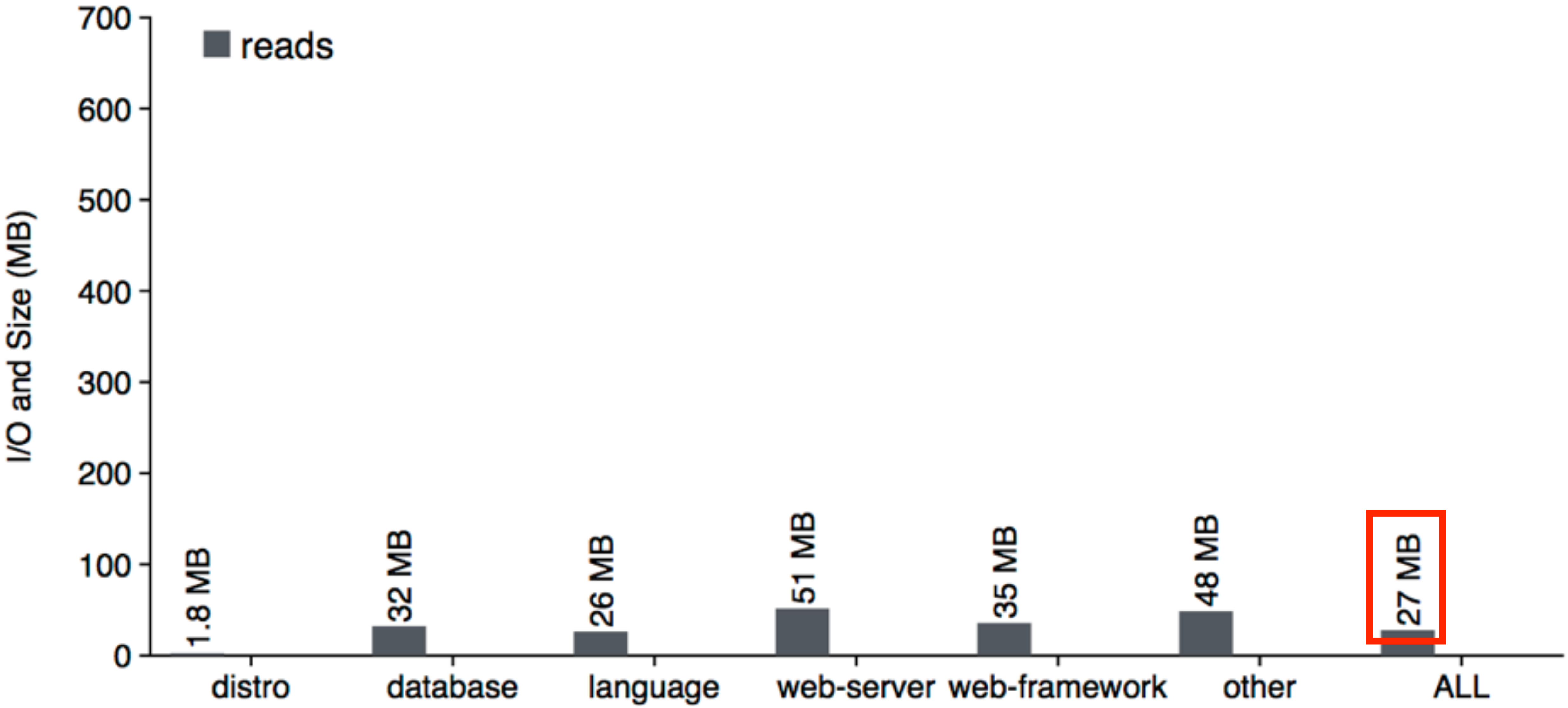
Analysis Questions

How is data distributed across Docker layers?

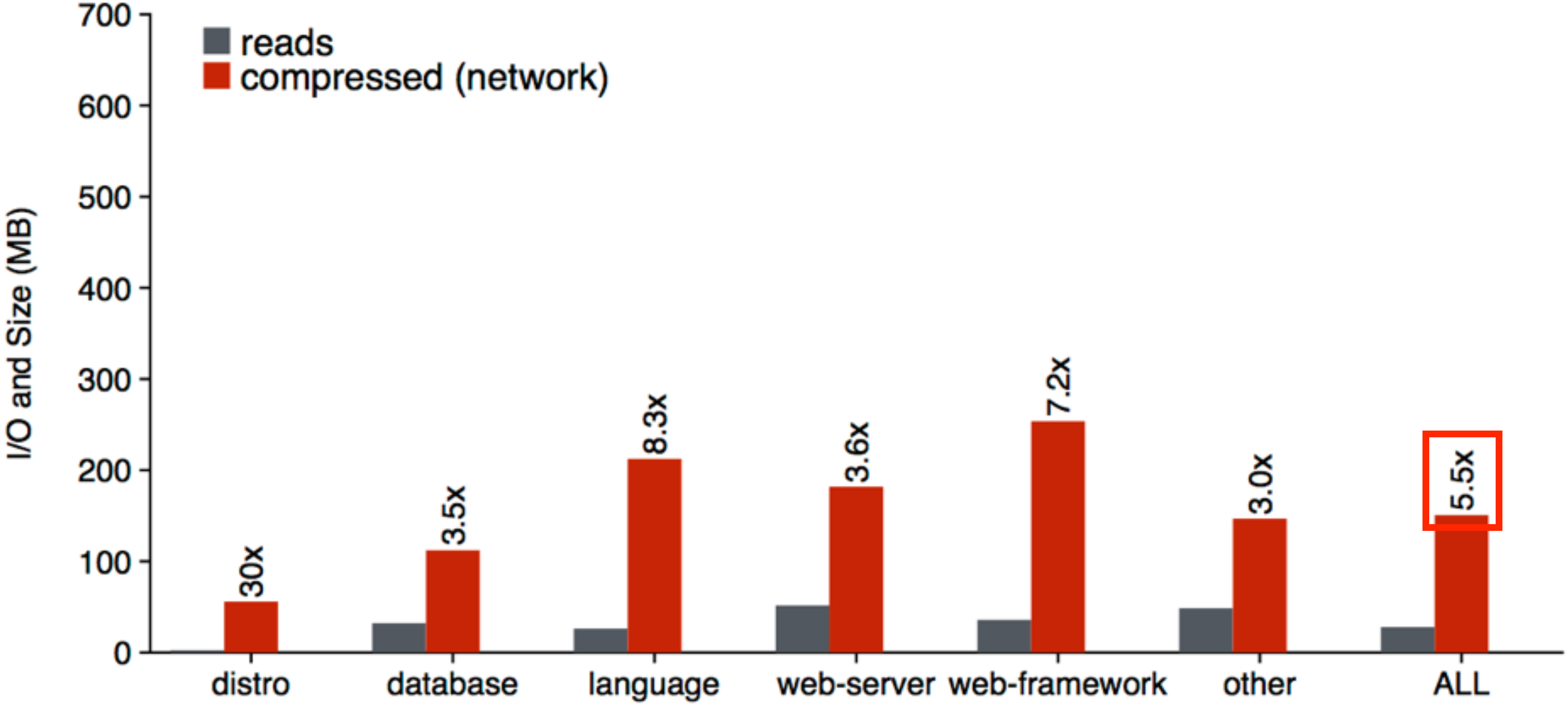
- half of data is at depth 9+
- **design implication**: flatten layers at runtime

How much image data is needed for container startup?

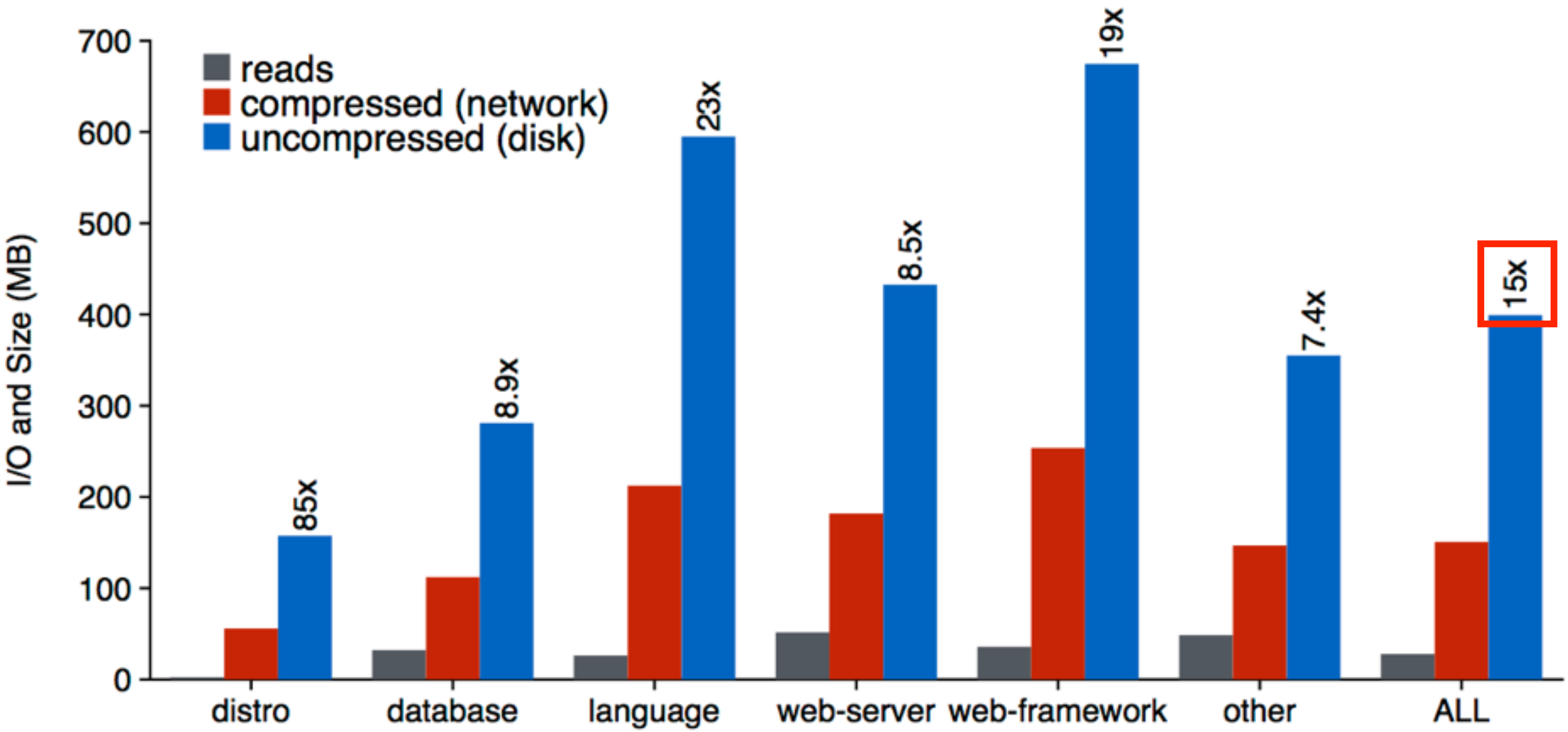
Container Amplification



Container Amplification



Container Amplification



only 6.4% of data needed during startup

Analysis Questions

How is data distributed across Docker layers?

- half of data is at depth 9+
- **design implication**: flatten layers at runtime

How much image data is needed for container startup?

- 6.4% of data is needed
- **design implication**: lazily fetch data

Outline

Motivation: Modularity in Modern Storage

Overview: Types of Modularity

Library Study: Apple Desktop Applications

Layer Study: Facebook Messages

Microservice Study: Docker Containers

Slacker: a Lazy Docker Storage Driver

Conclusions

Slacker Outline

AUFS Storage Driver Background

Slacker Design

Evaluation

AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

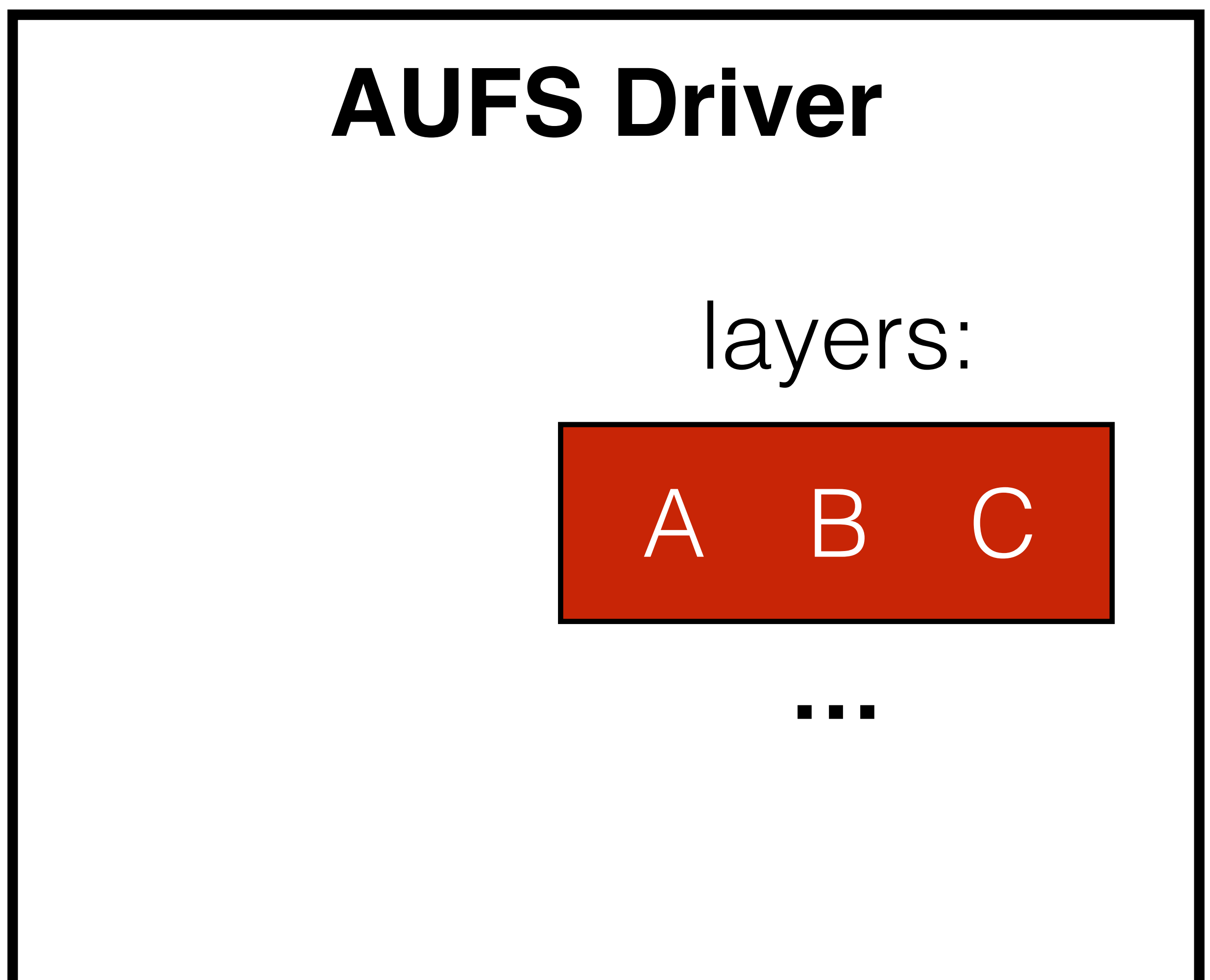
Operations

- push
- pull
- run

AUFS Storage Driver

Uses AUFS file system (Another Union FS)

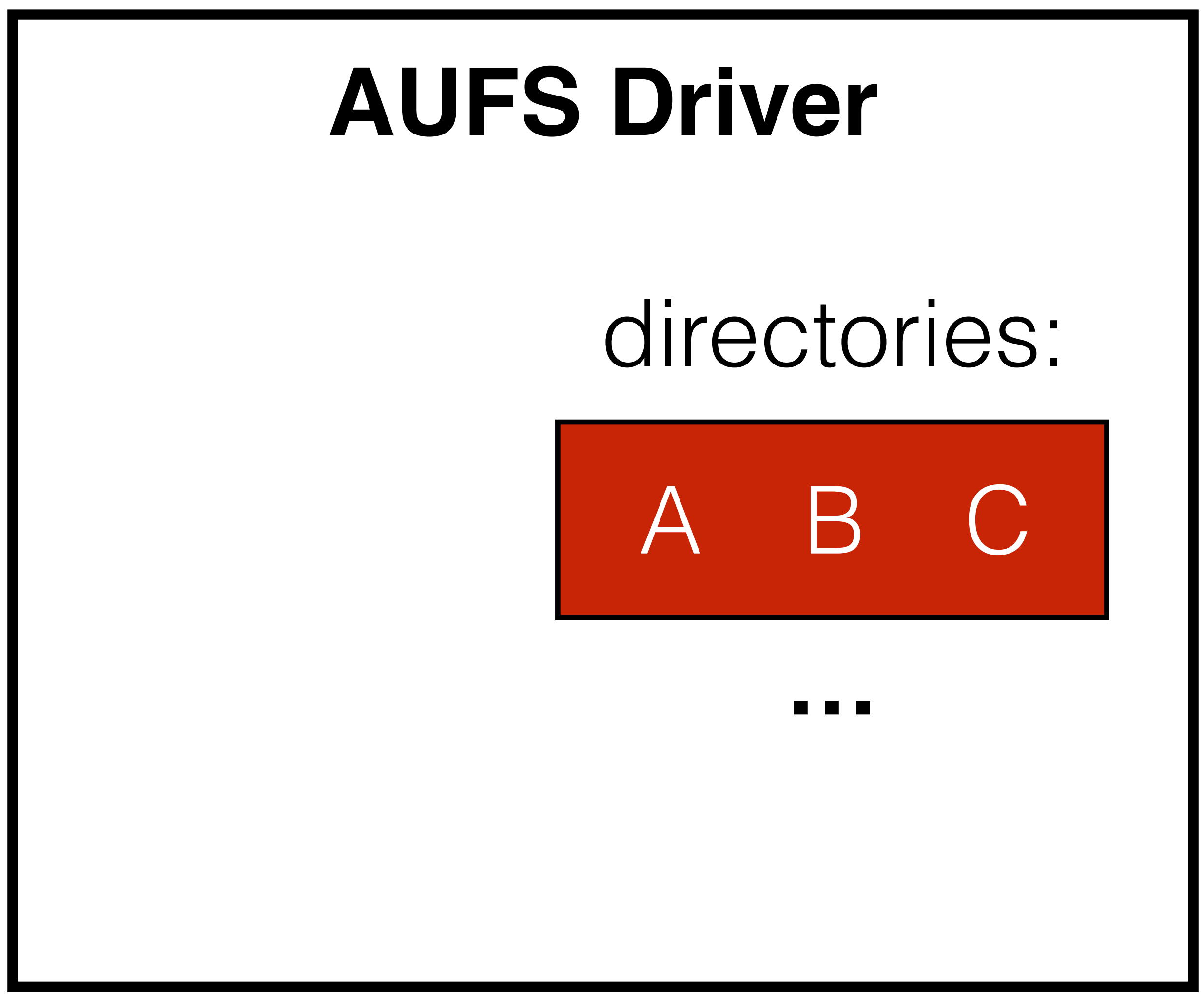
- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

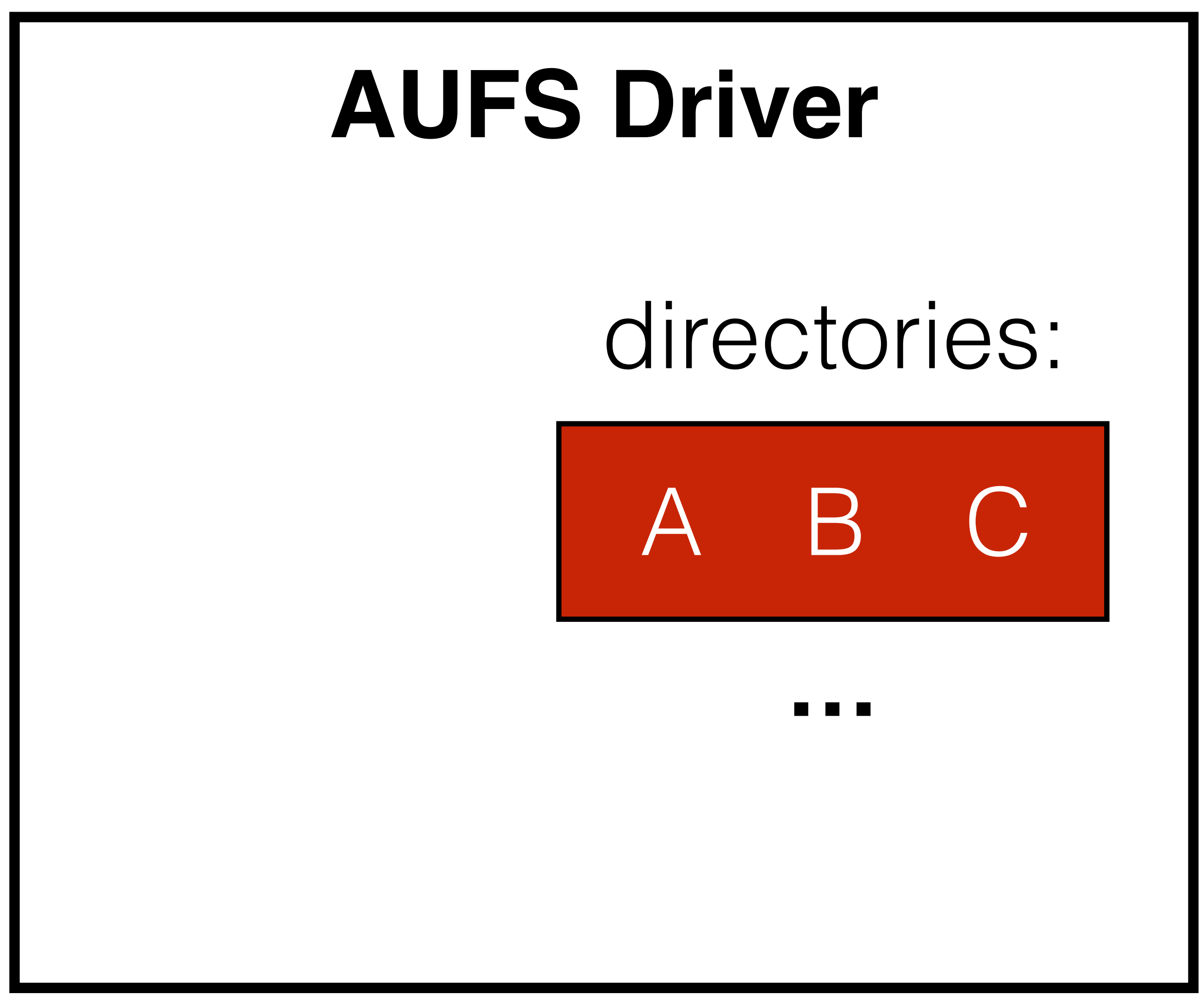


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

PUSH

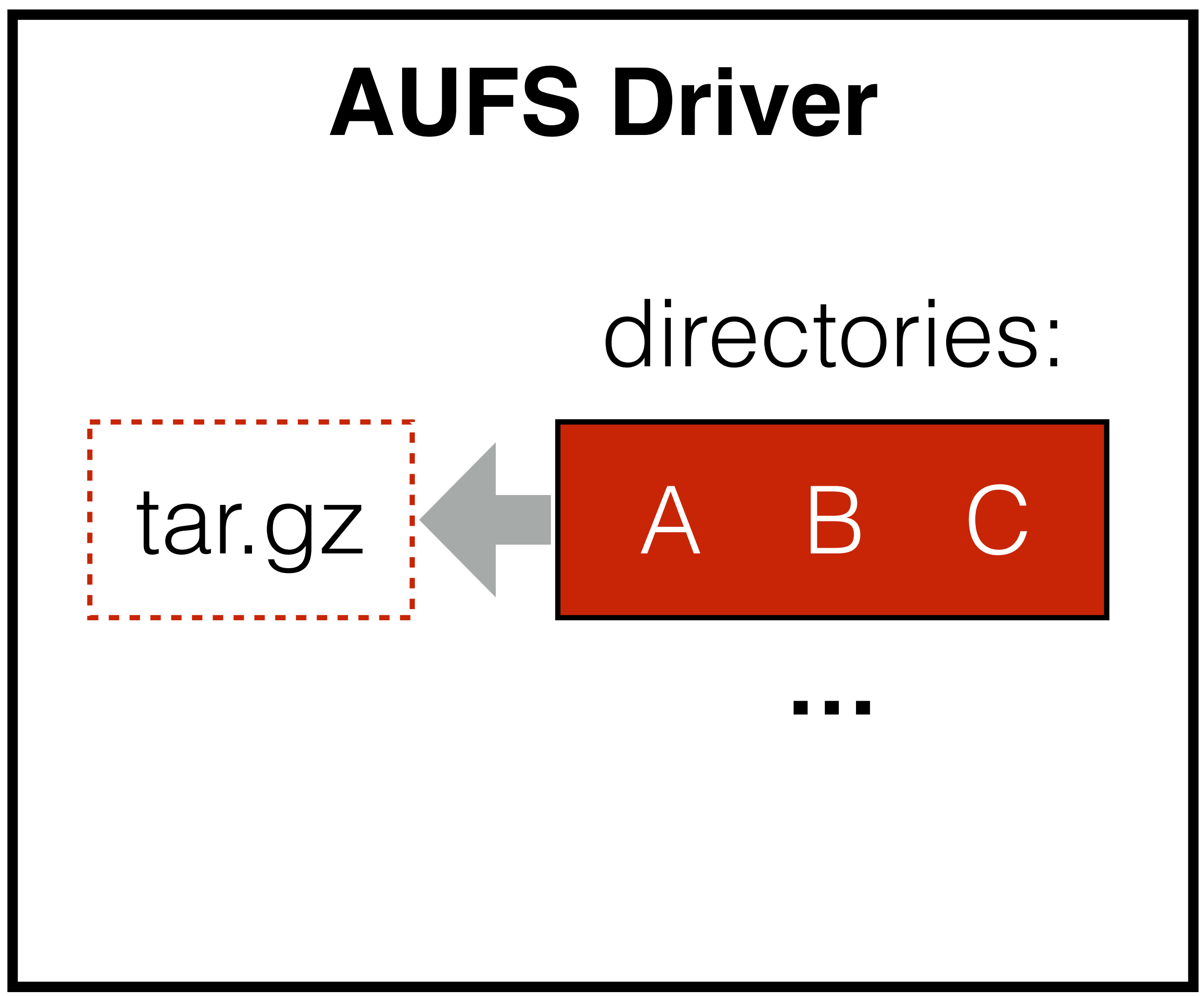


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

PUSH

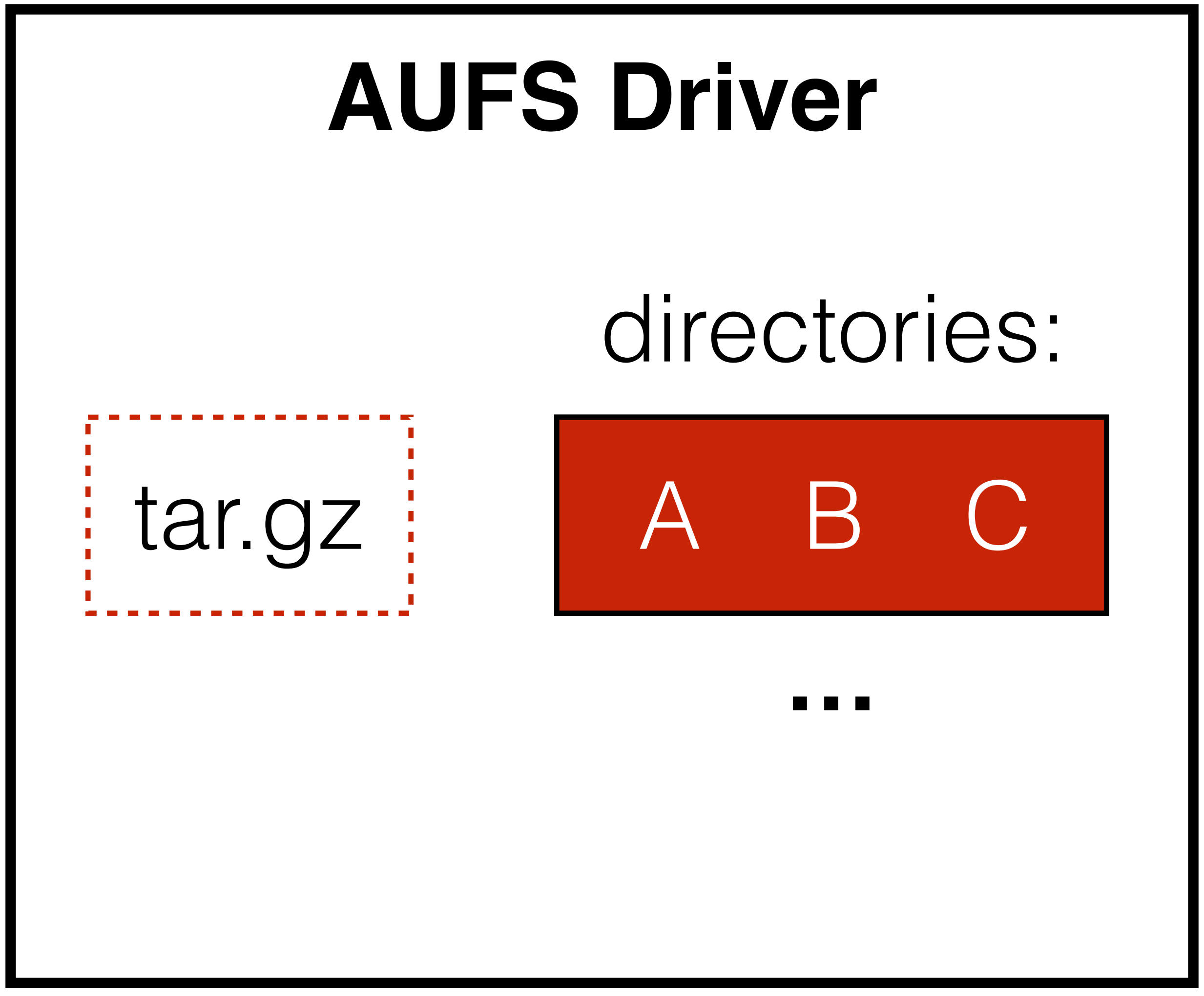


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

PUSH

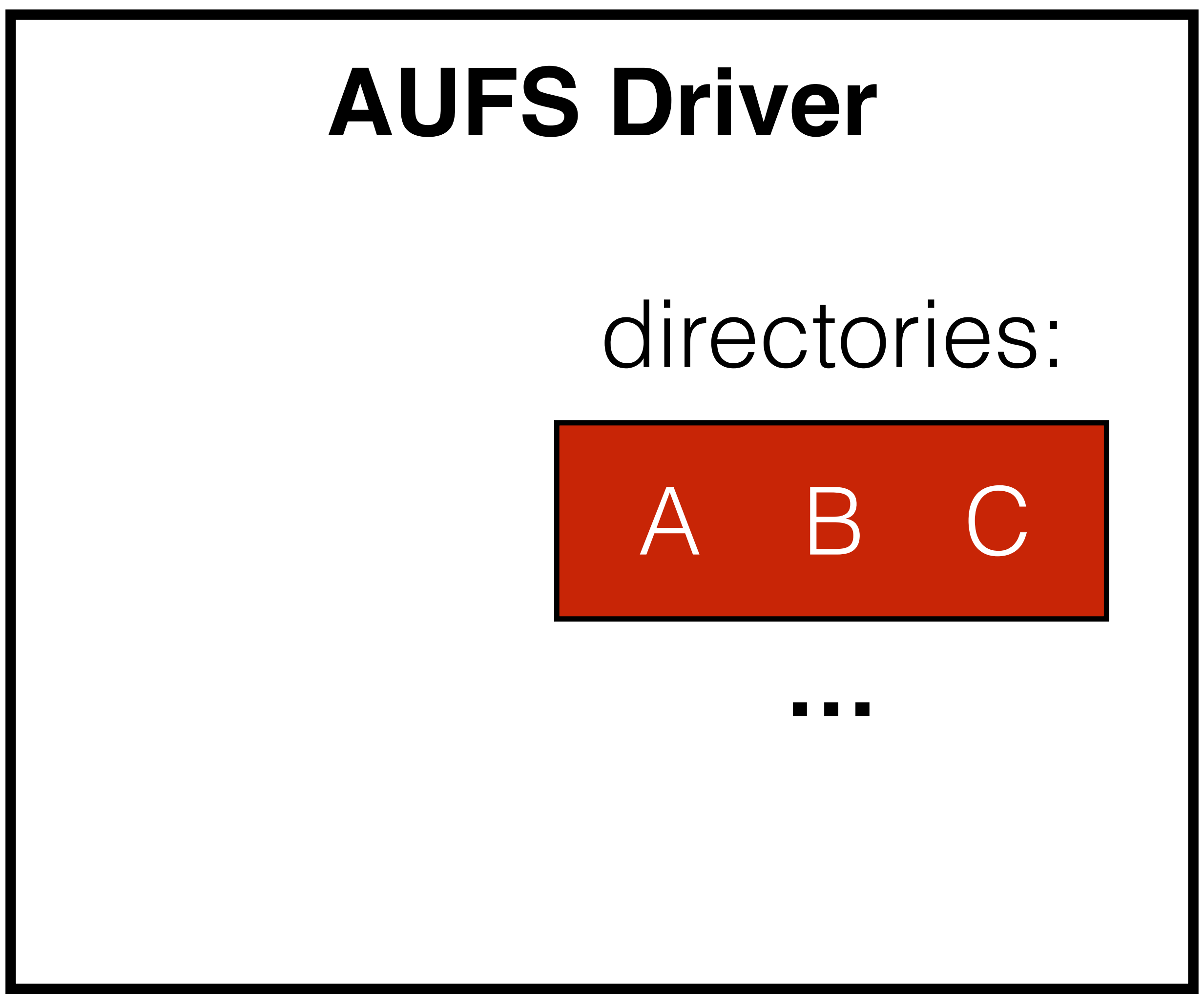


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

PULL

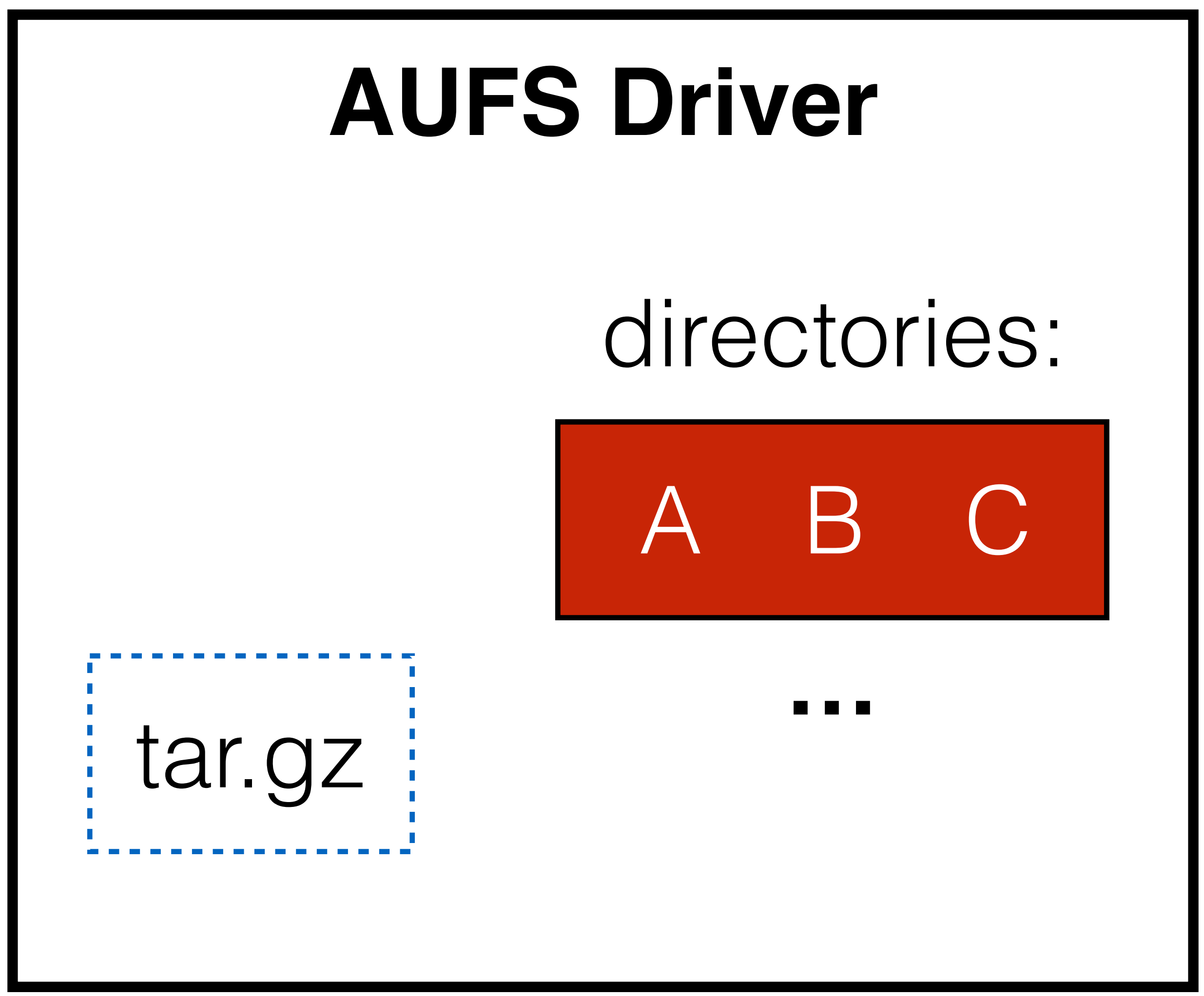


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

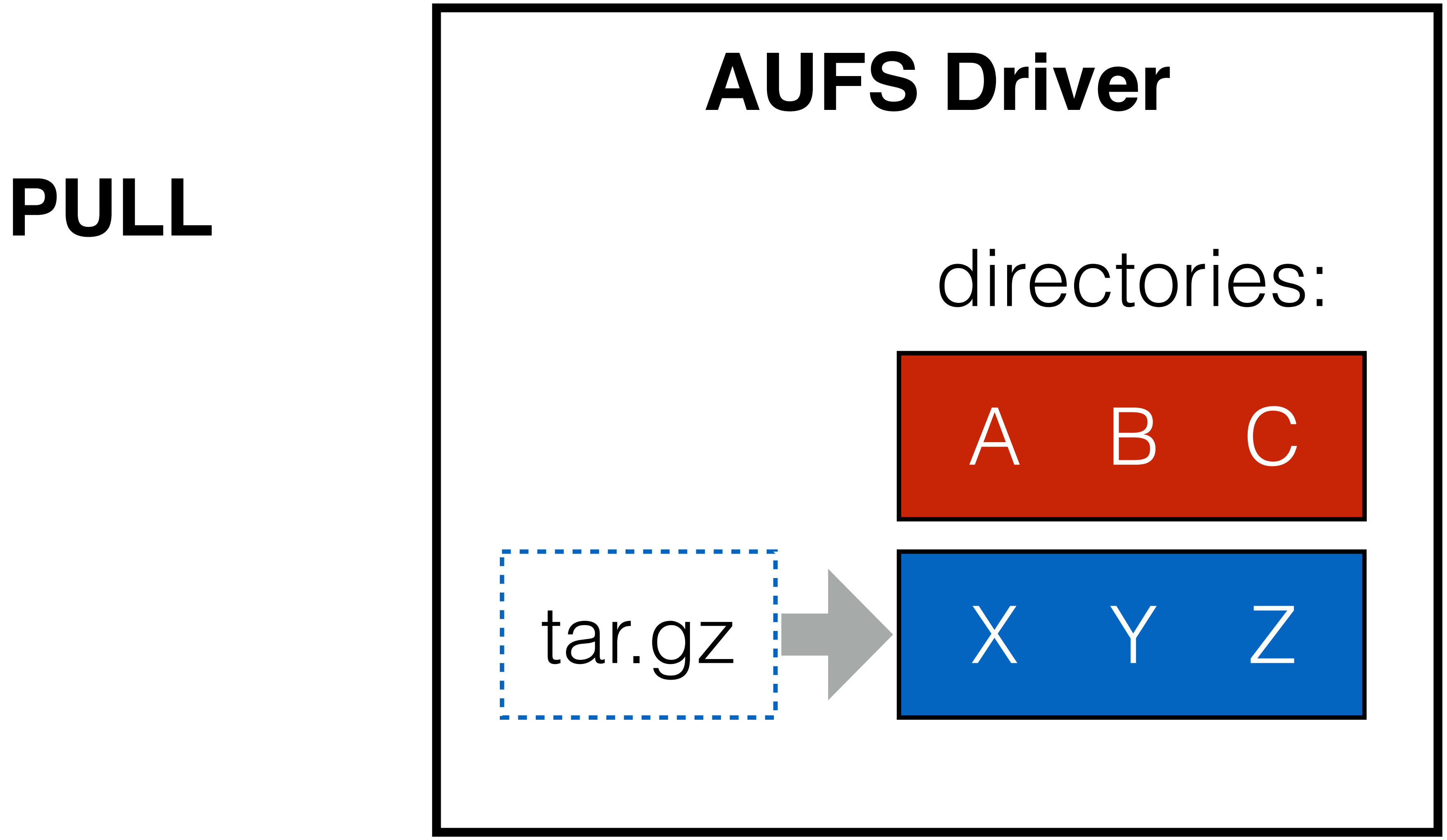
PULL



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

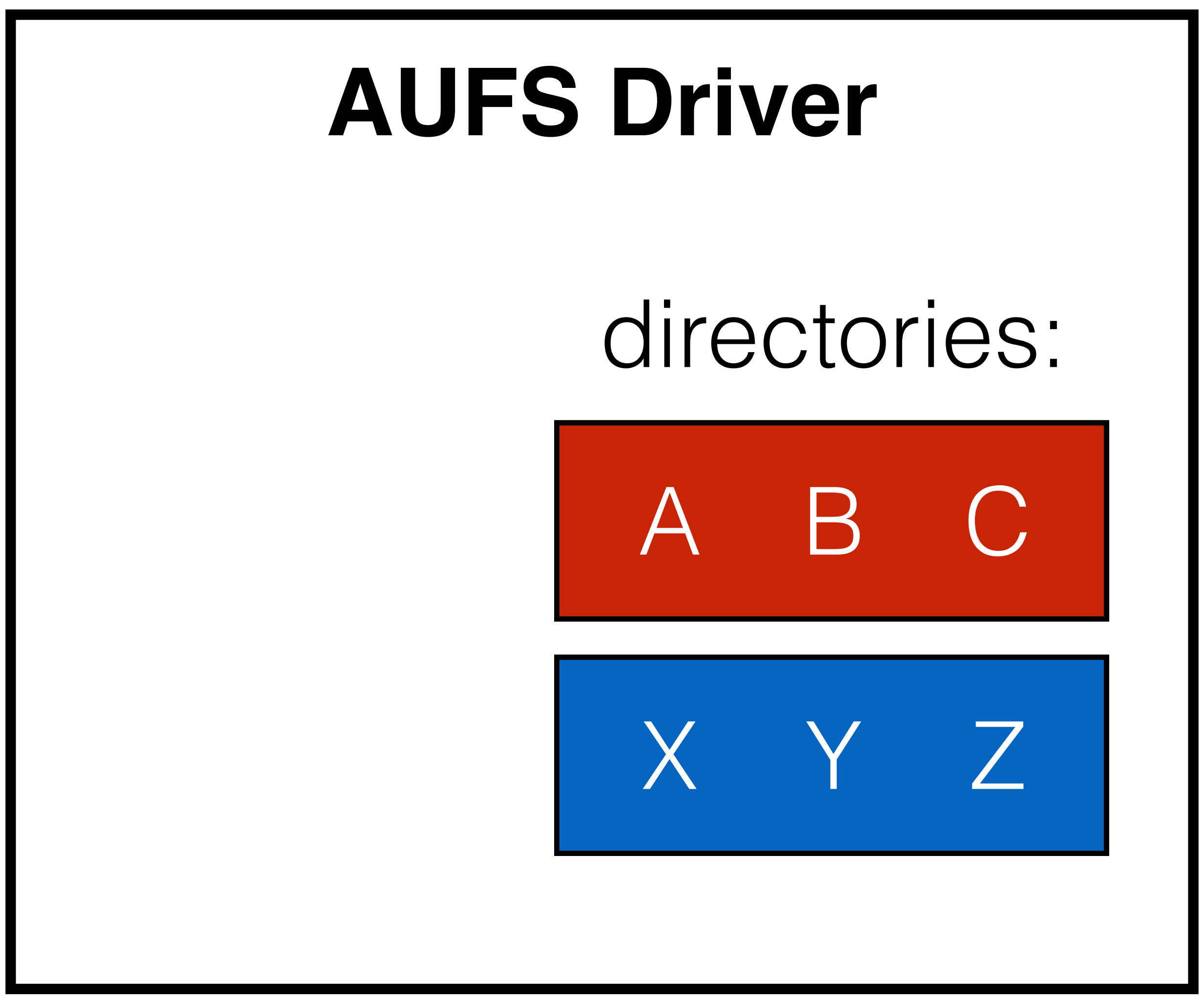


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

PULL

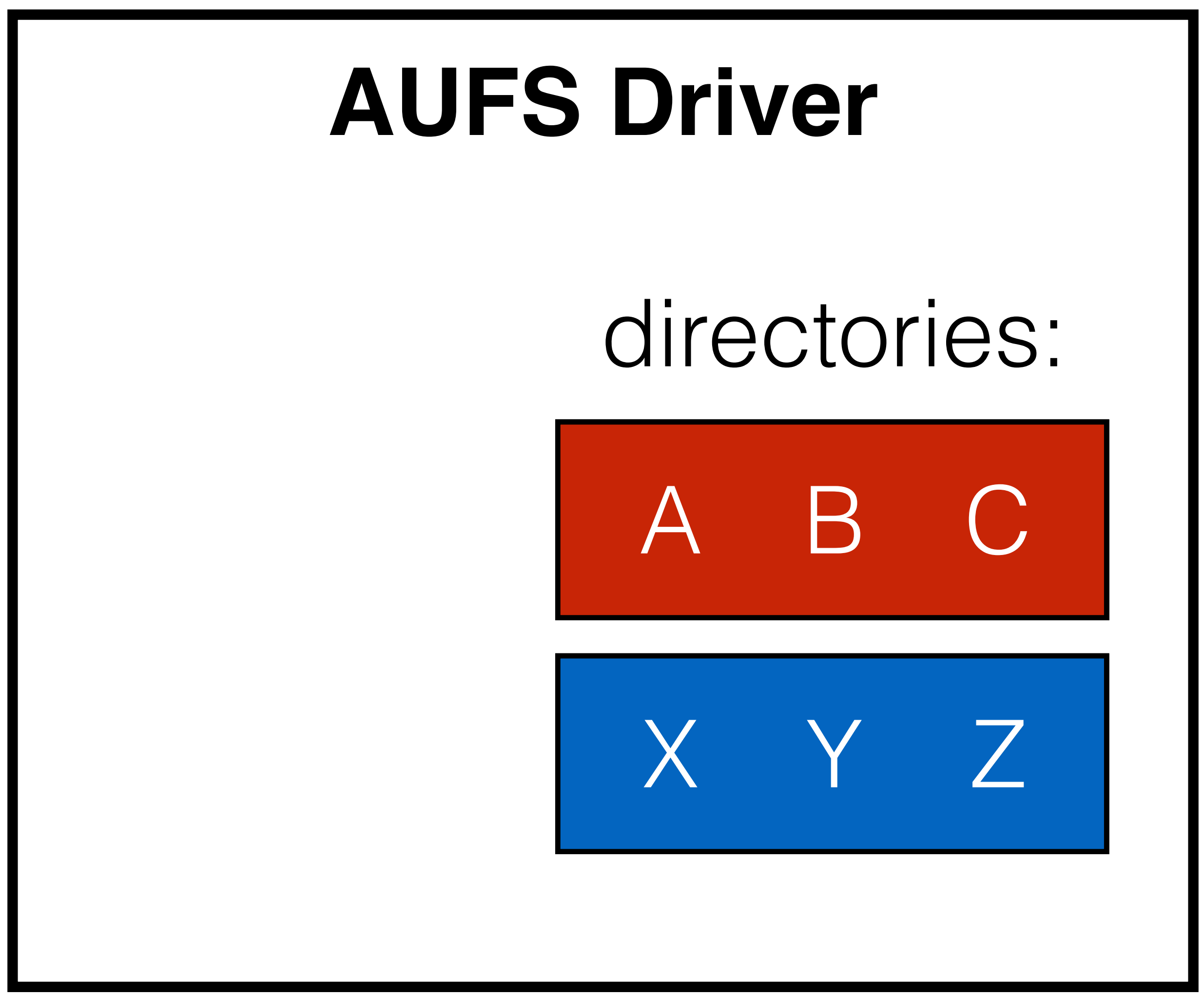


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

RUN

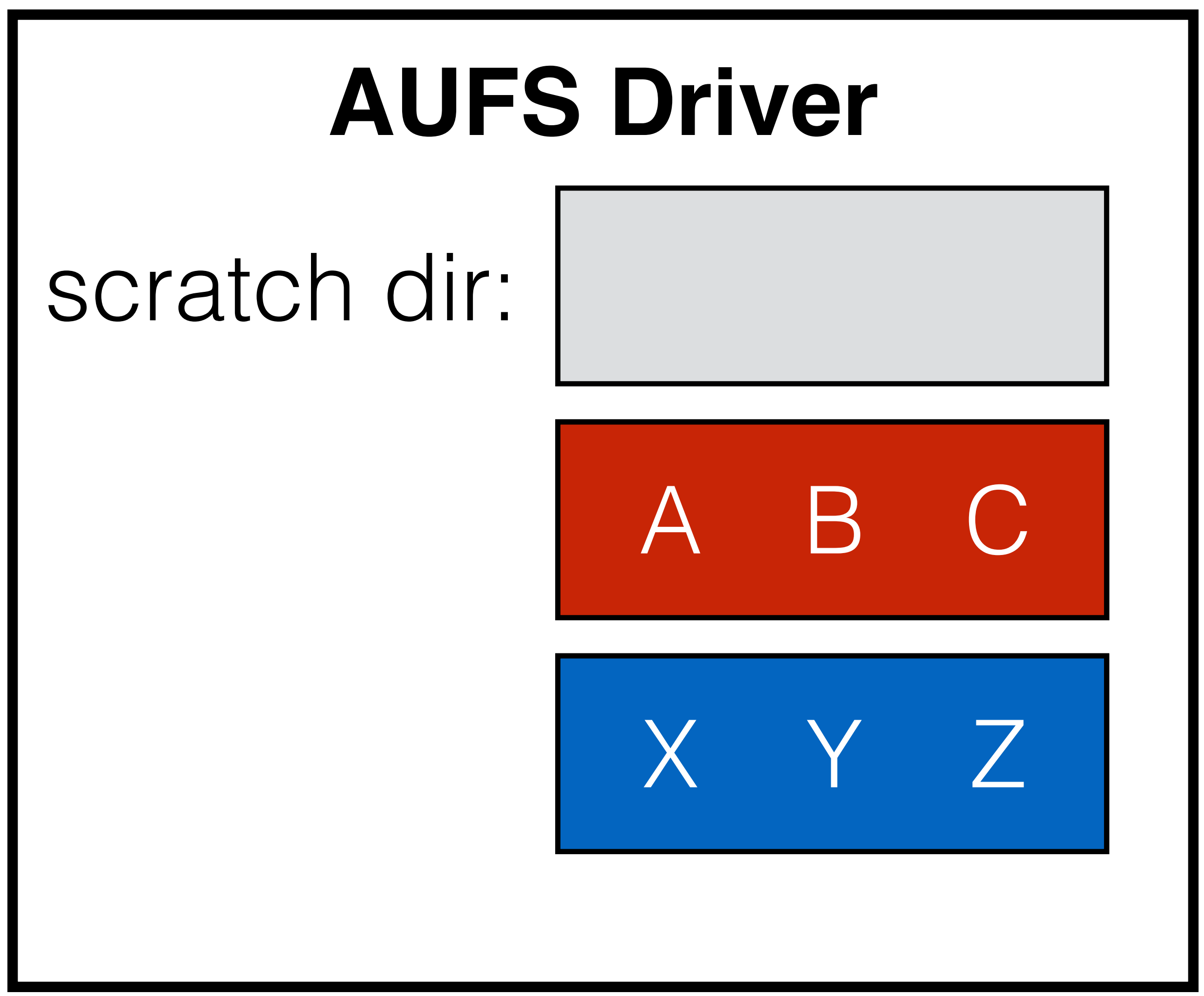


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

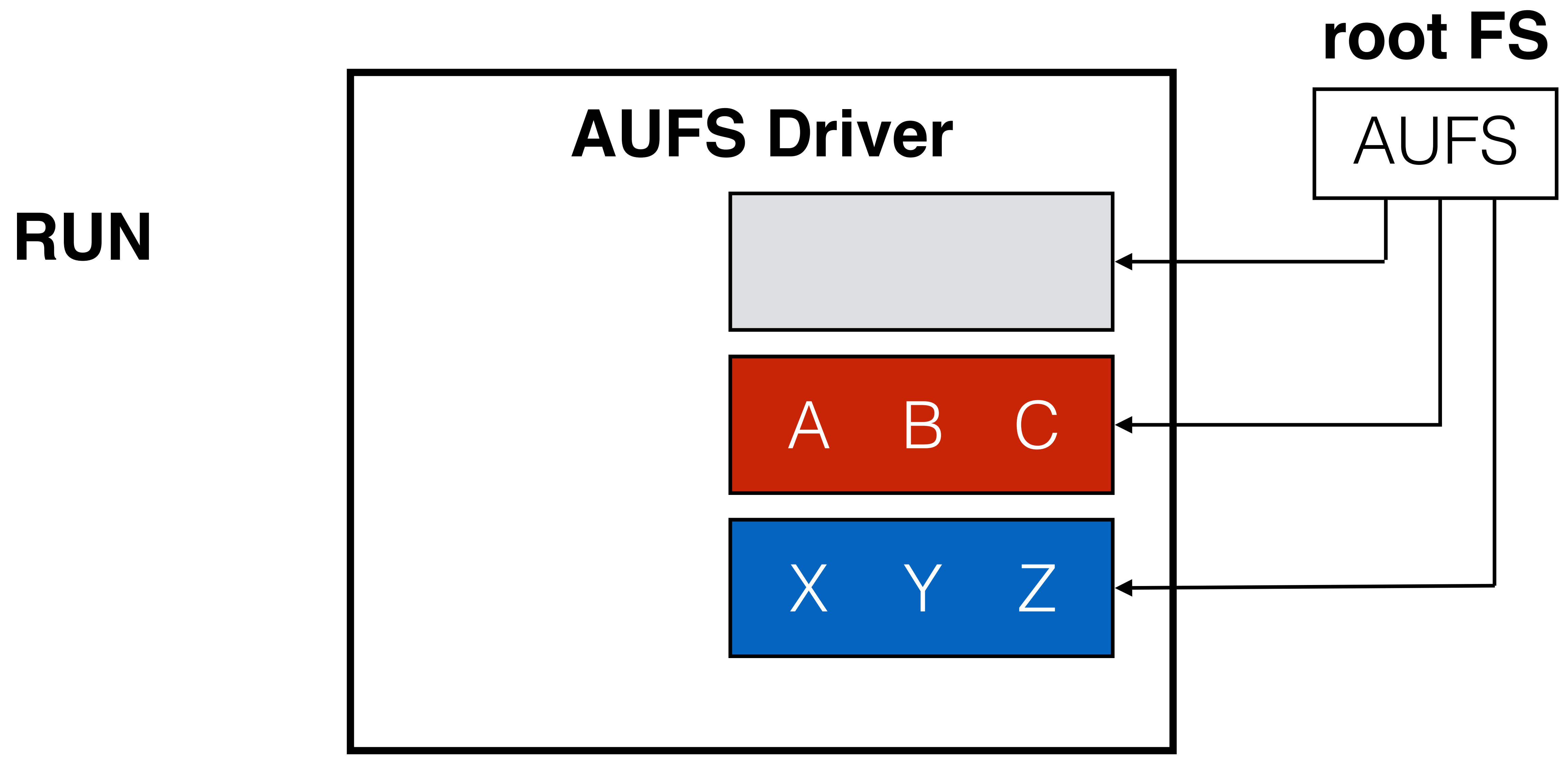
RUN



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

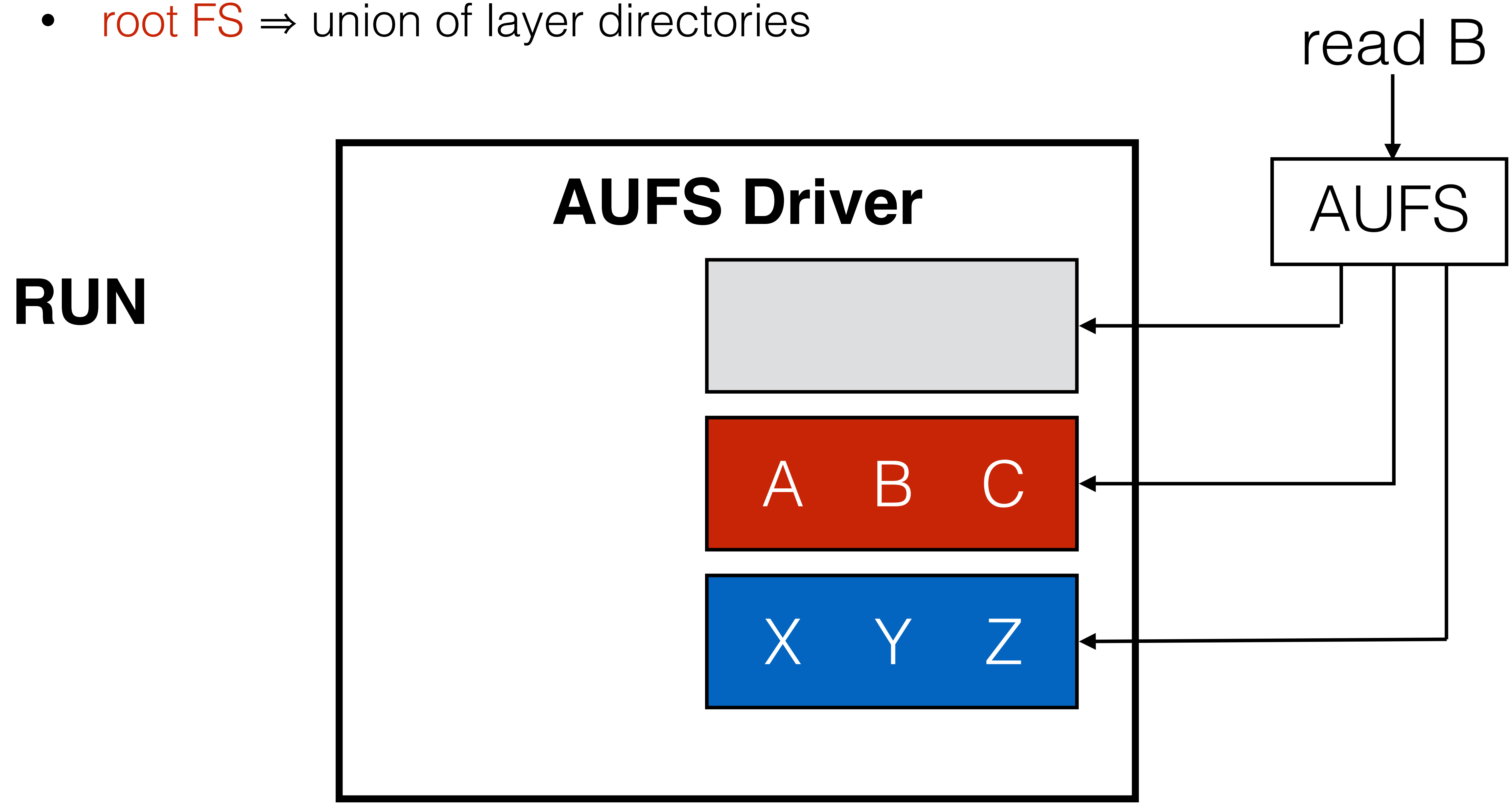
- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

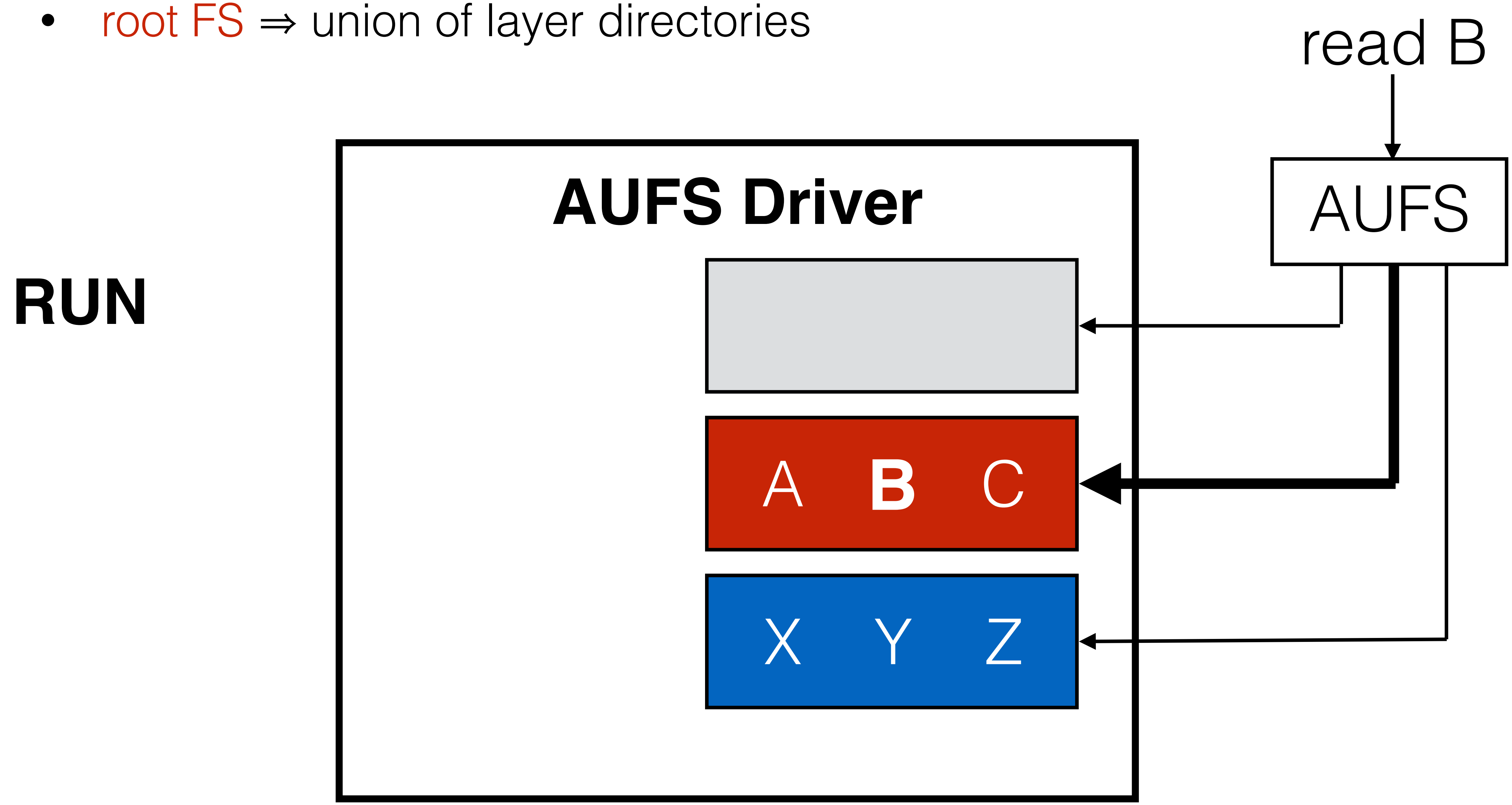
- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

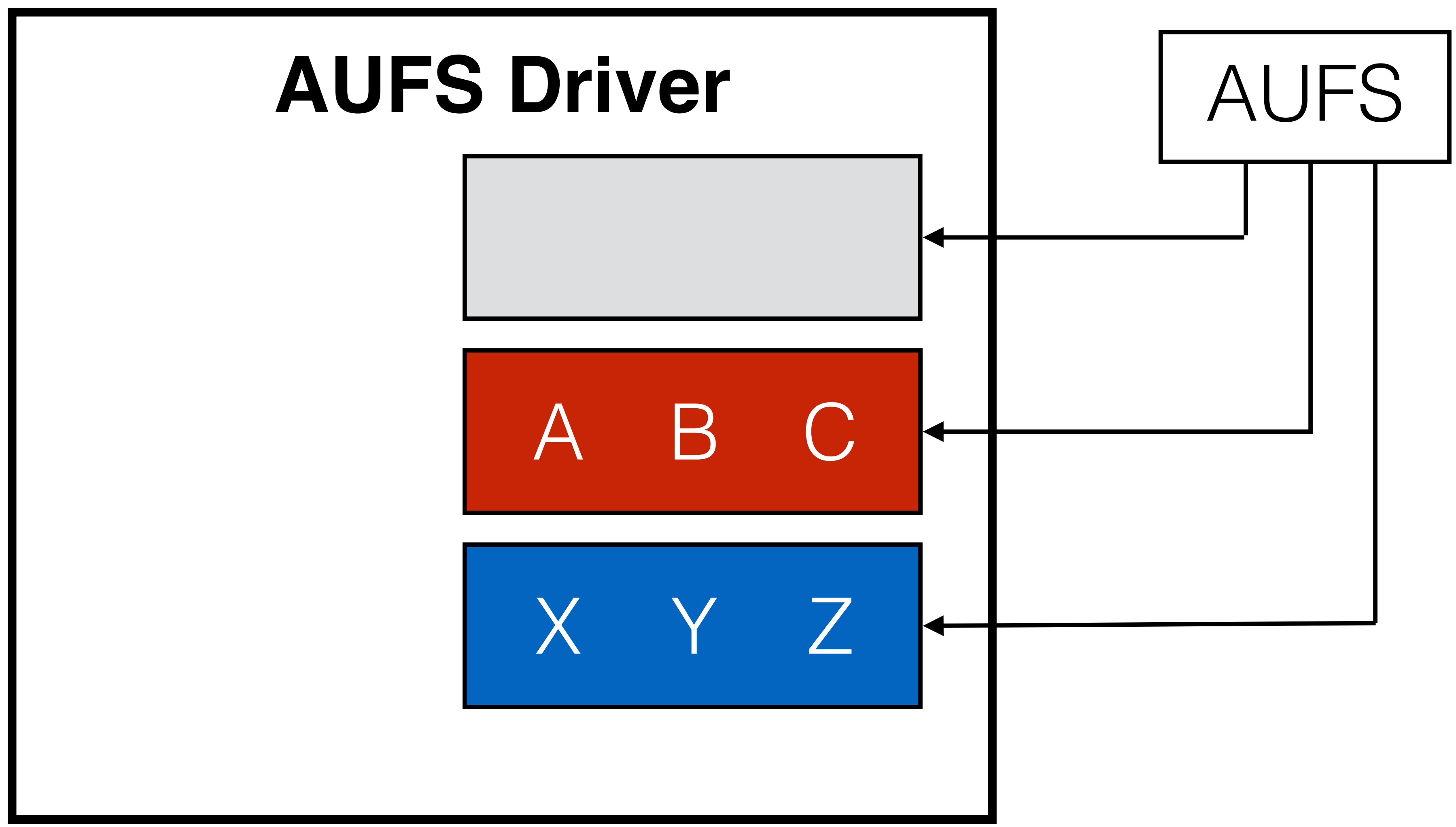


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

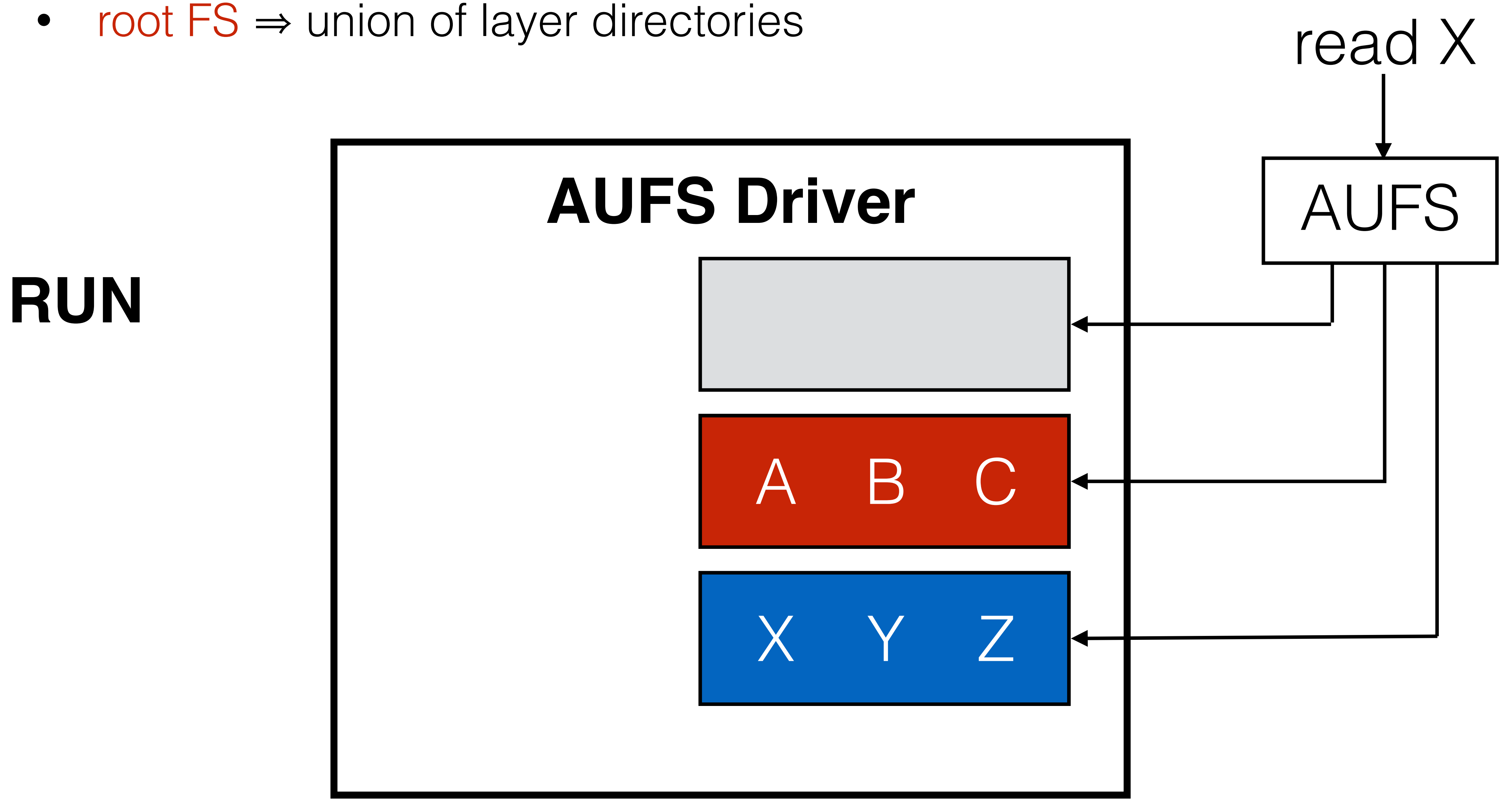
RUN



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

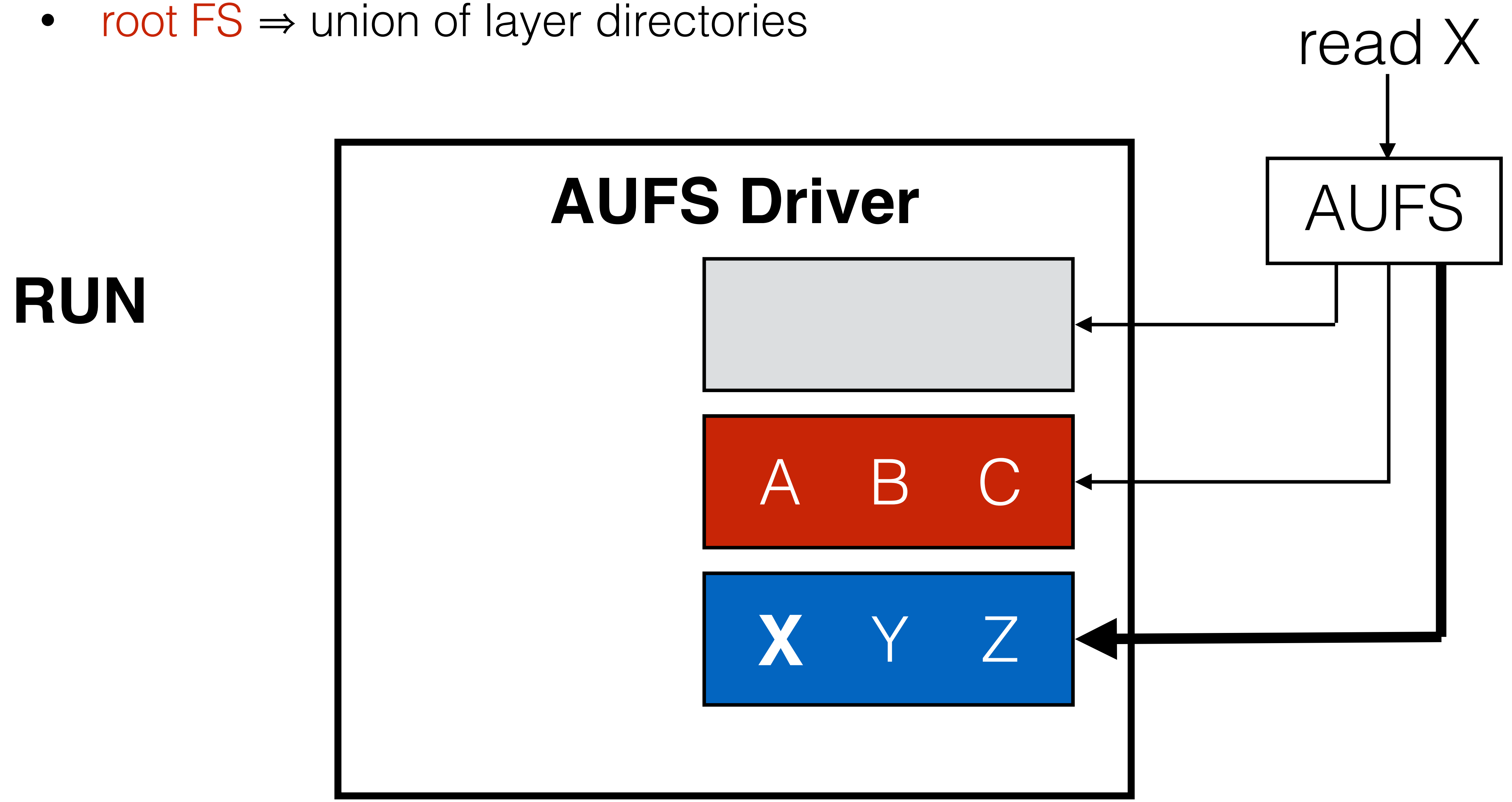
- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

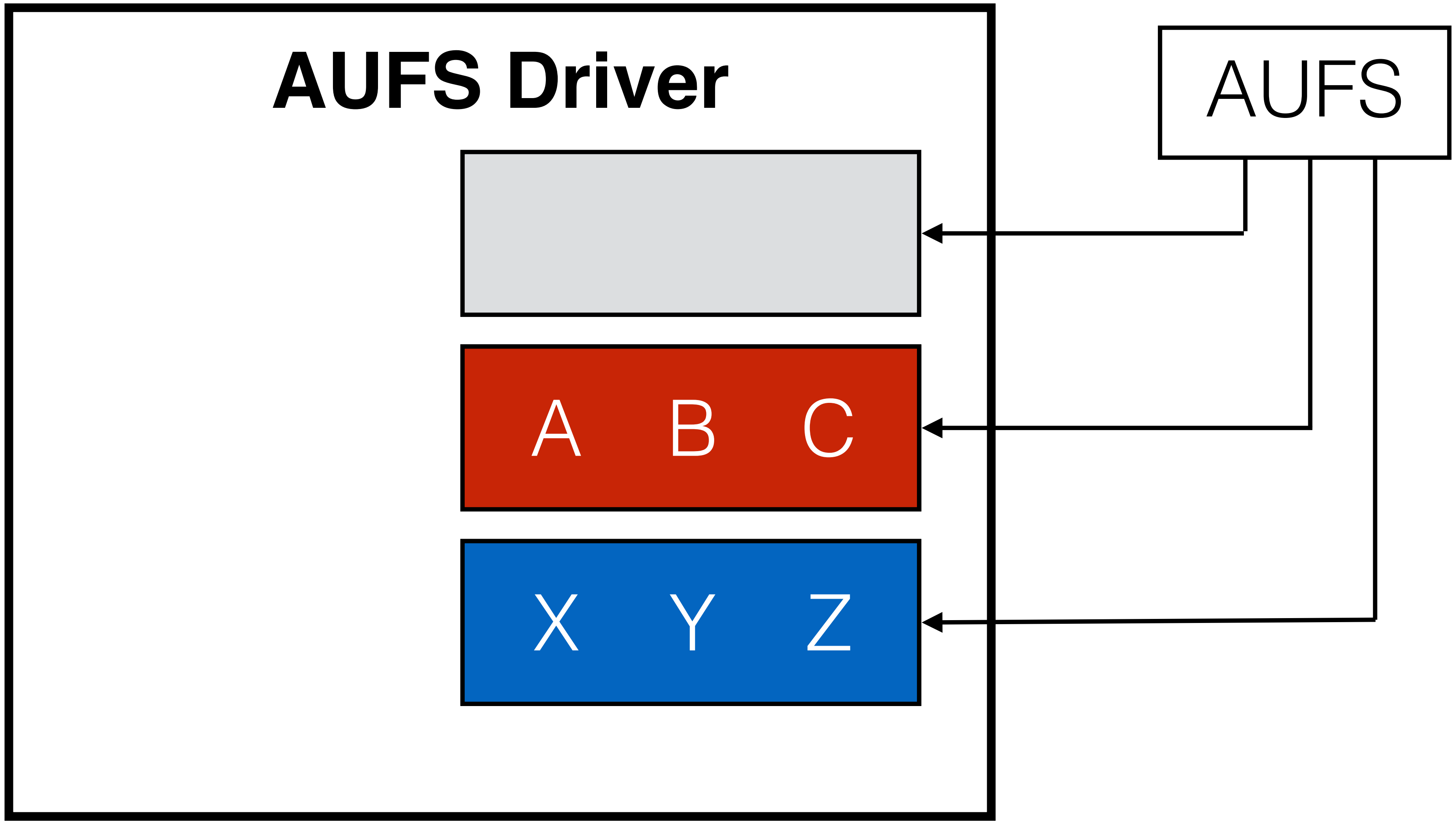


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

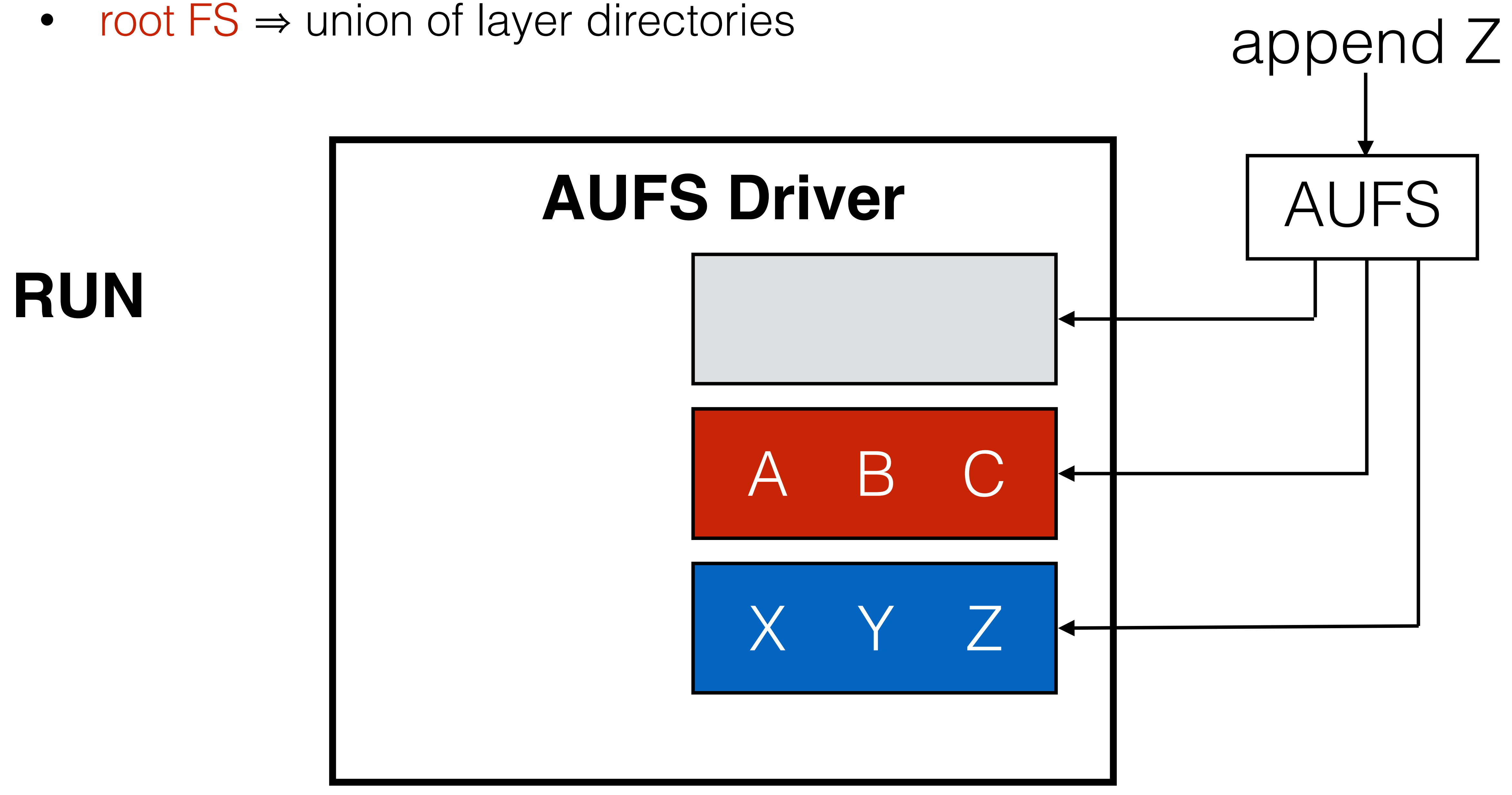
RUN



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

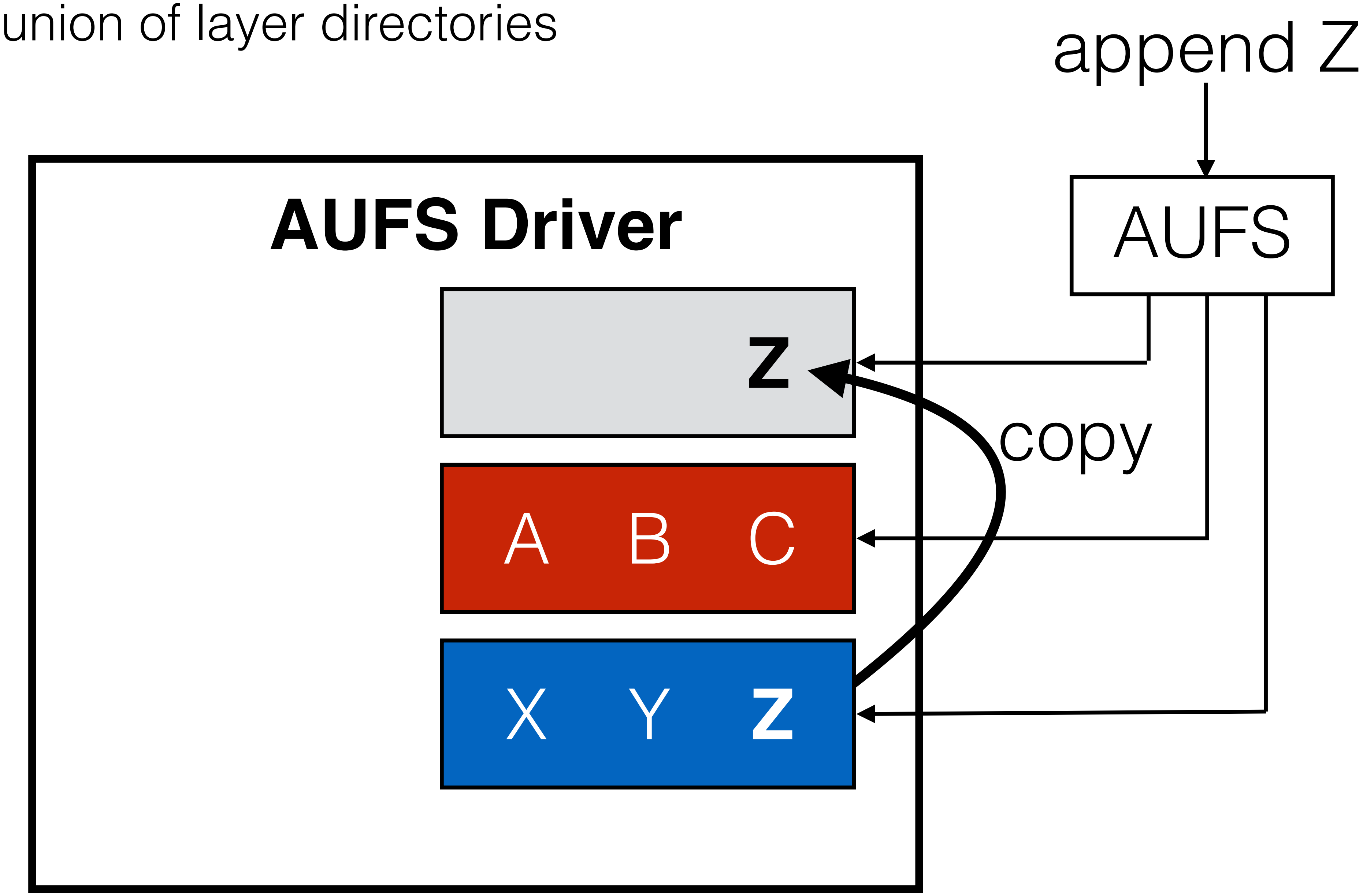


AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

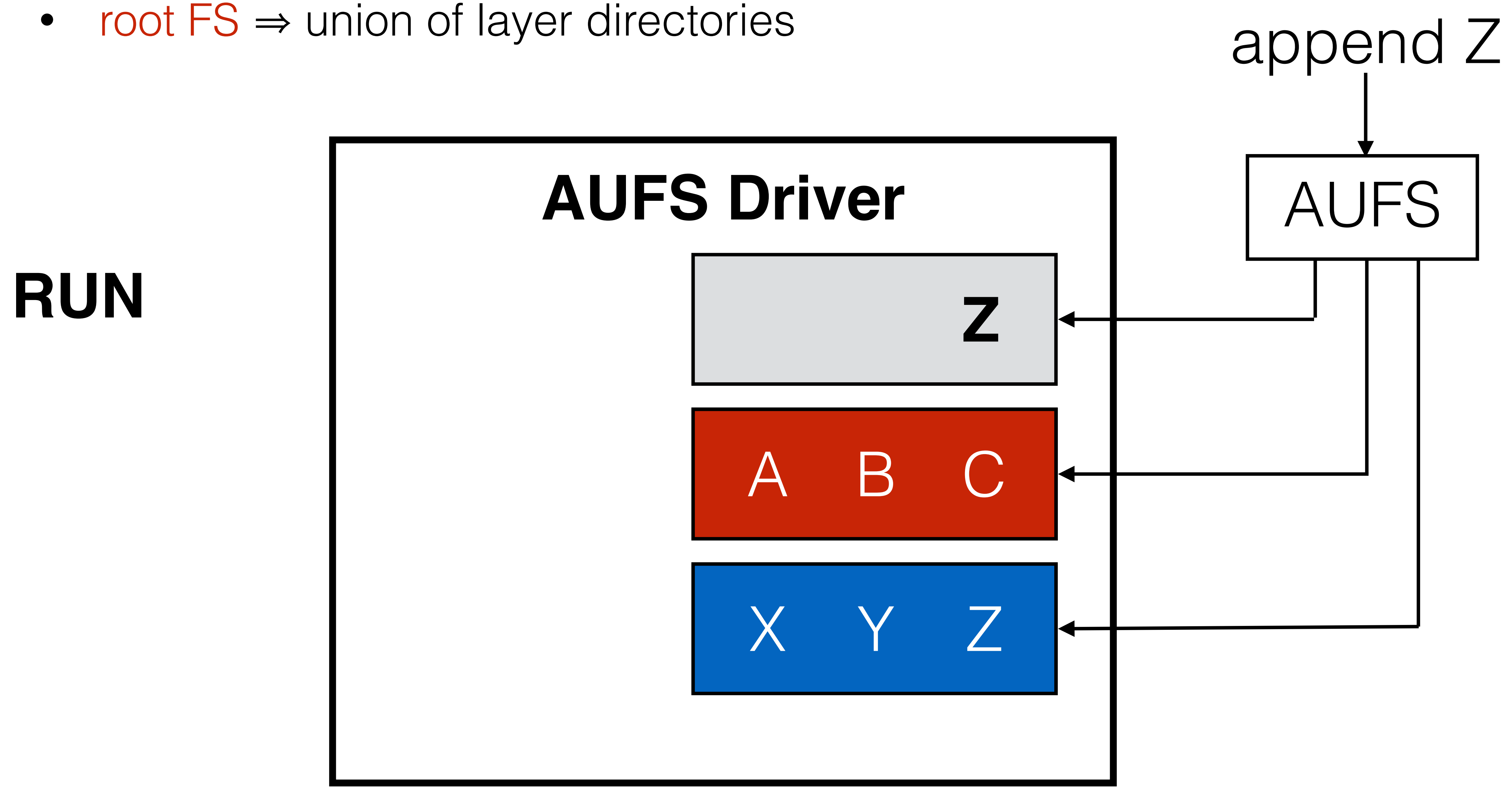
RUN



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

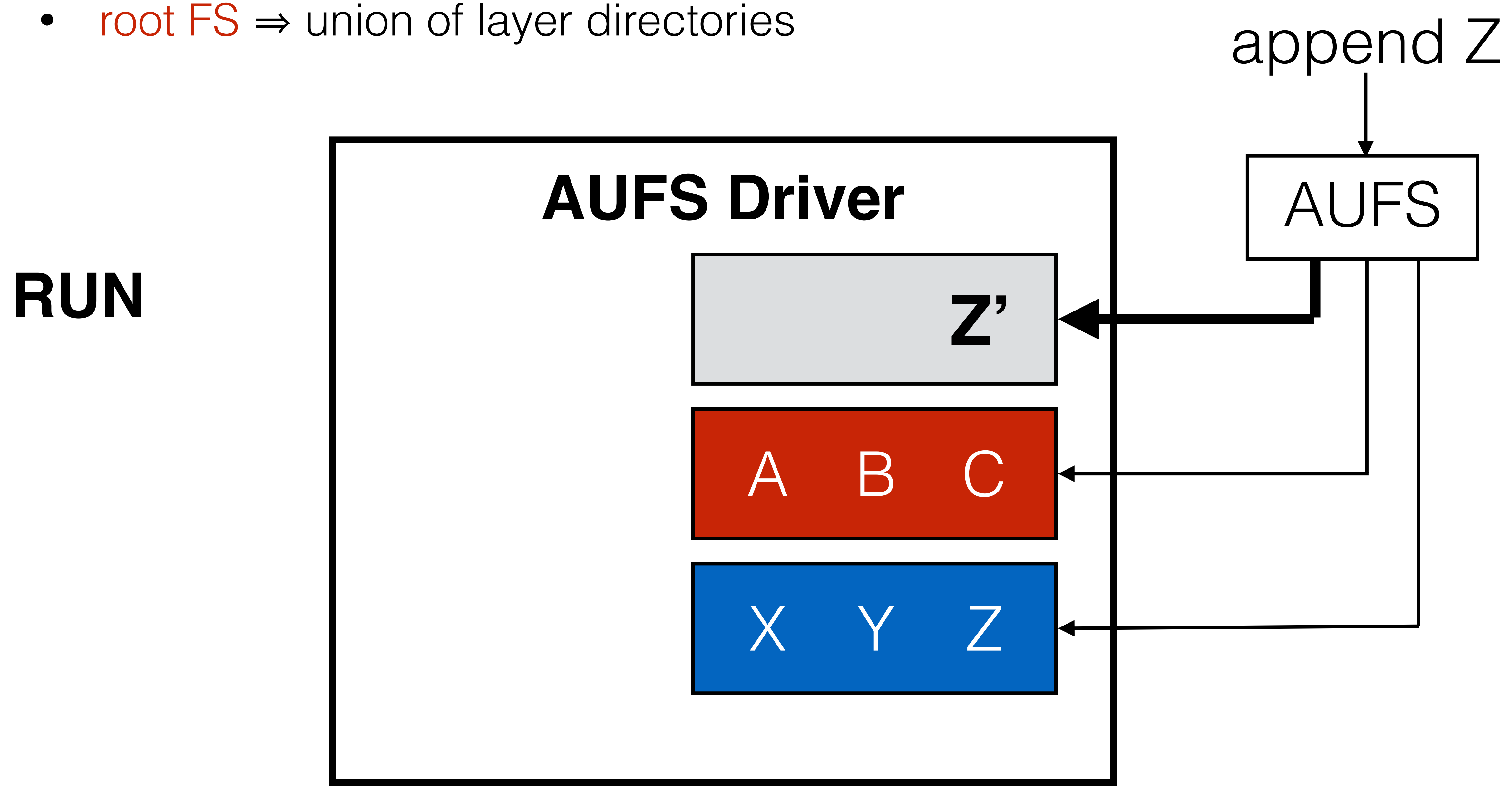
- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories



AUFS Storage Driver

Uses AUFS file system (Another Union FS)

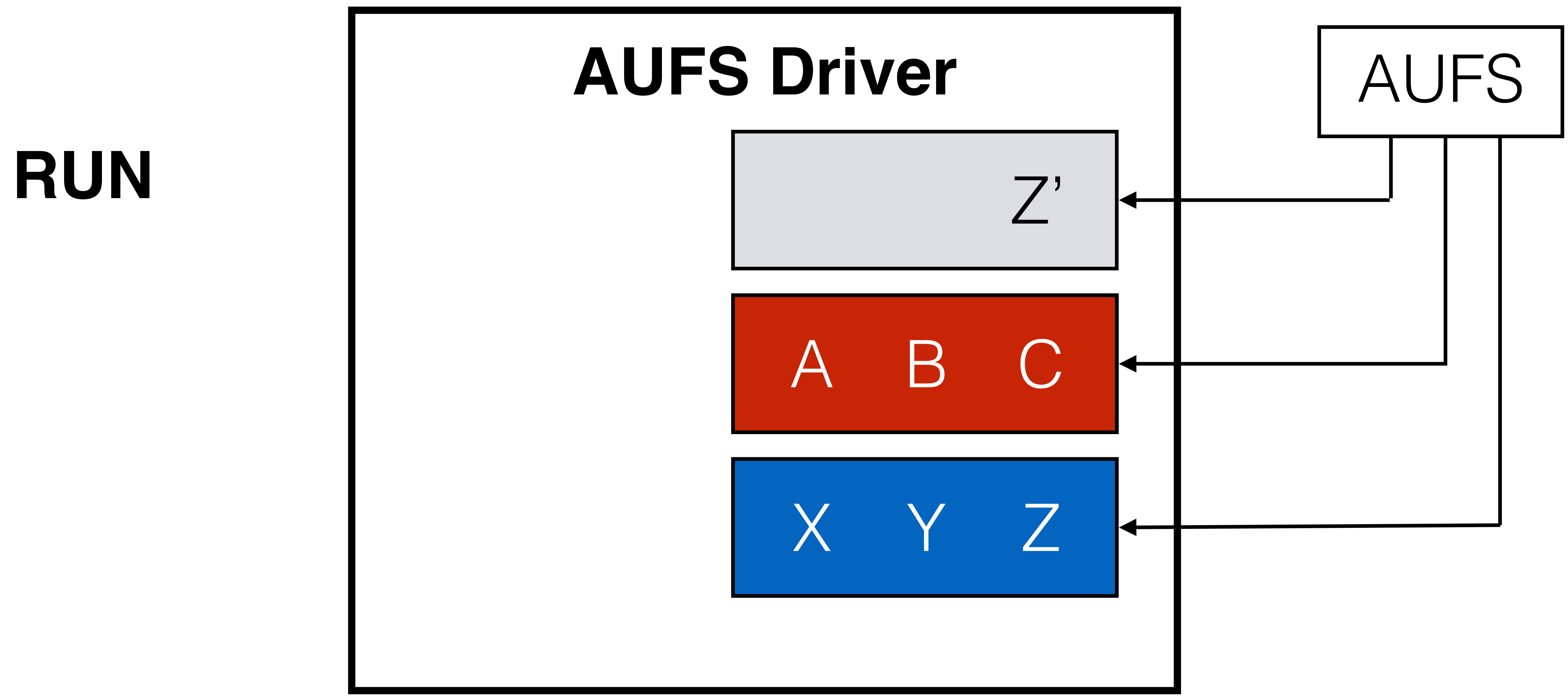
- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories



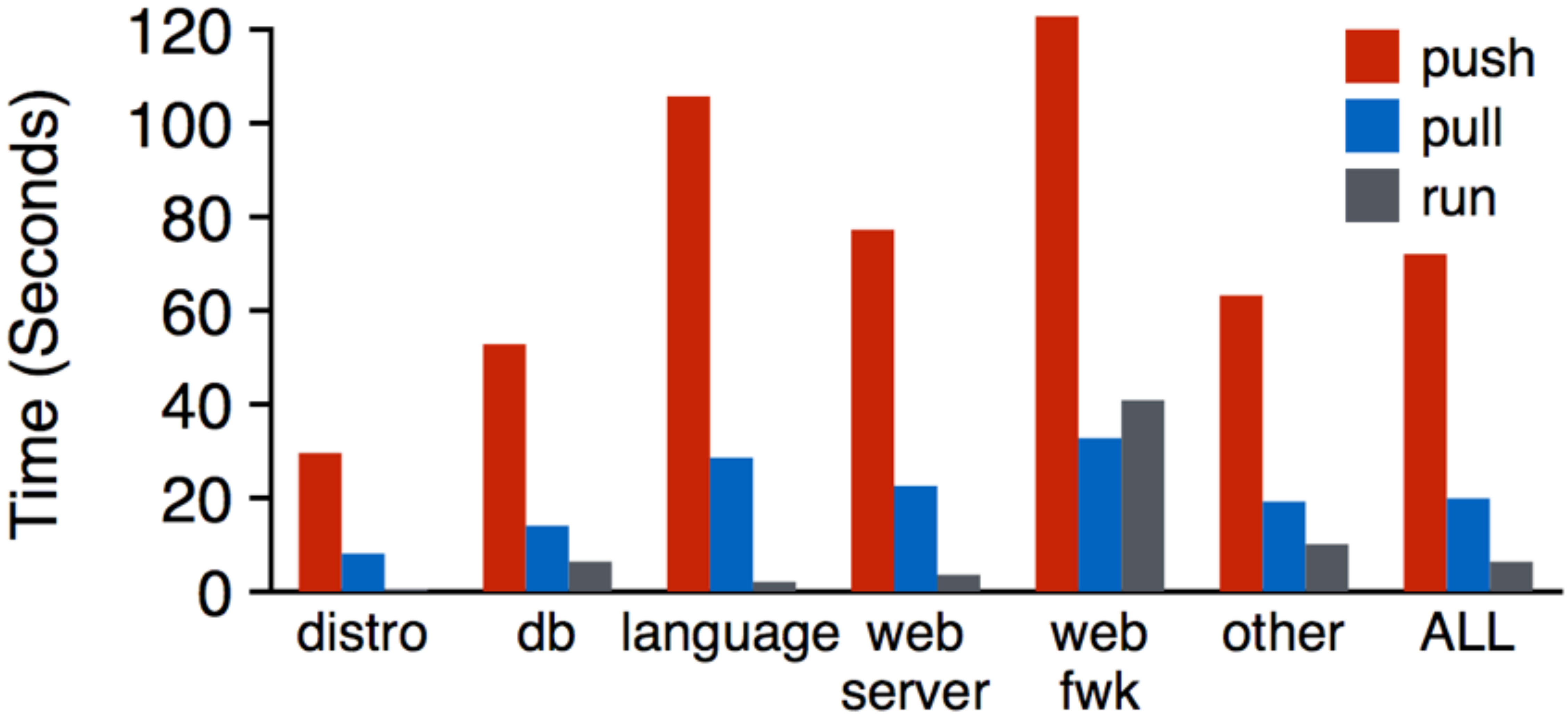
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

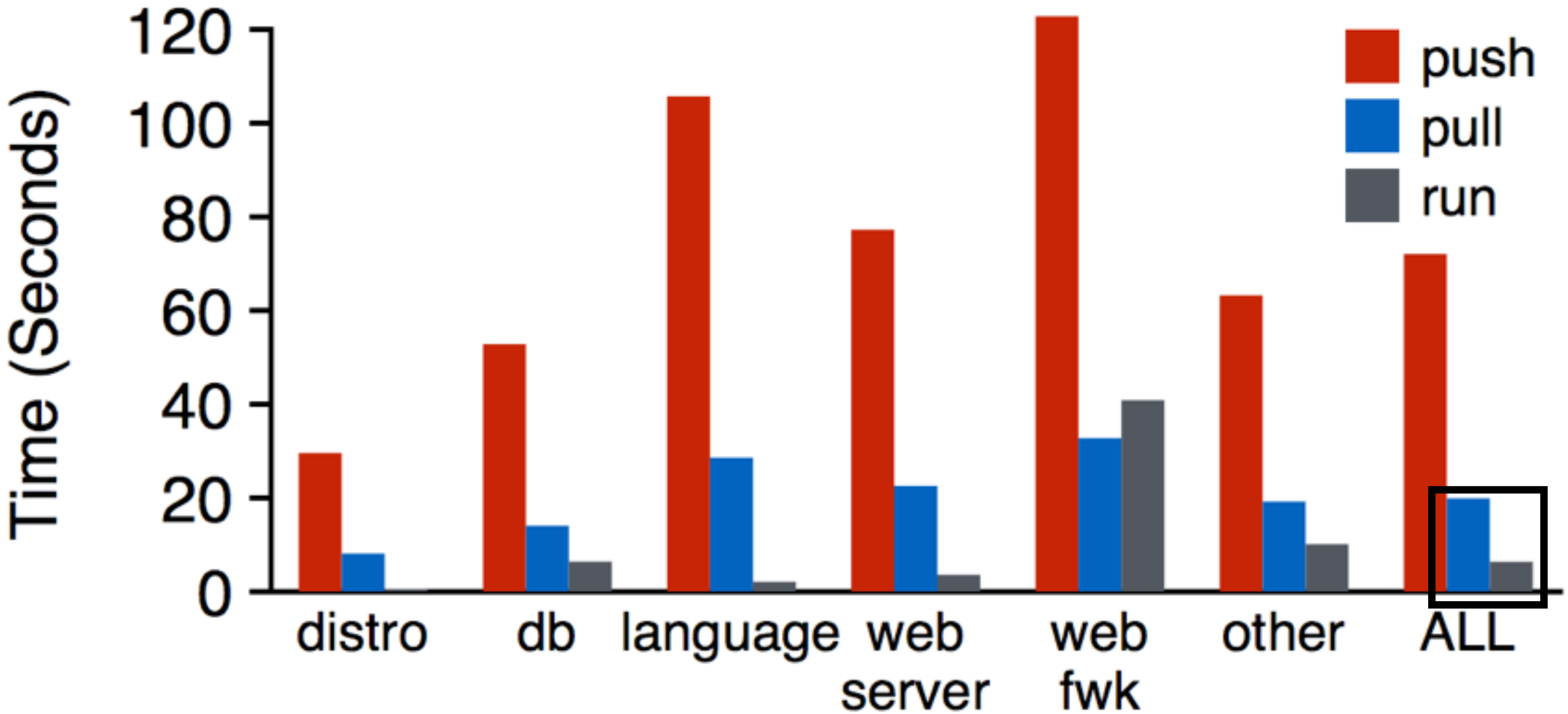
- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories



HelloBench with AUFS



HelloBench with AUFS



76% of deployment cycle spent on pull

AUFS Problems

Deployment problem: lots of copying

- Caused by push+pull
- Compute costs: compression
- Network costs: transferring tar.gz files
- Storage I/O costs: installing packages
- Pull+run = 26 seconds

Execution problem: coarse-grained COW

- Iterate over directories on lookup
- Large copies for small writes
- *more in dissertation*

Slacker Outline

AUFS Storage Driver Background

Slacker Design

Evaluation

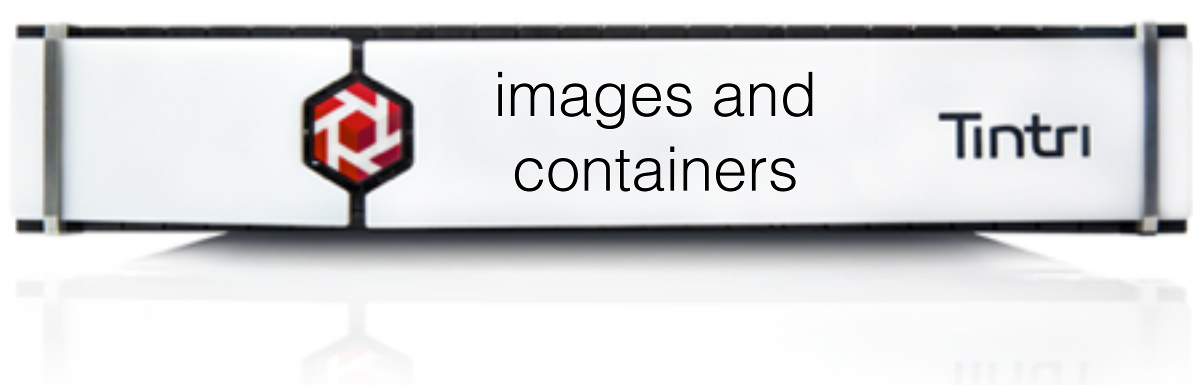
Slacker Driver

Goals

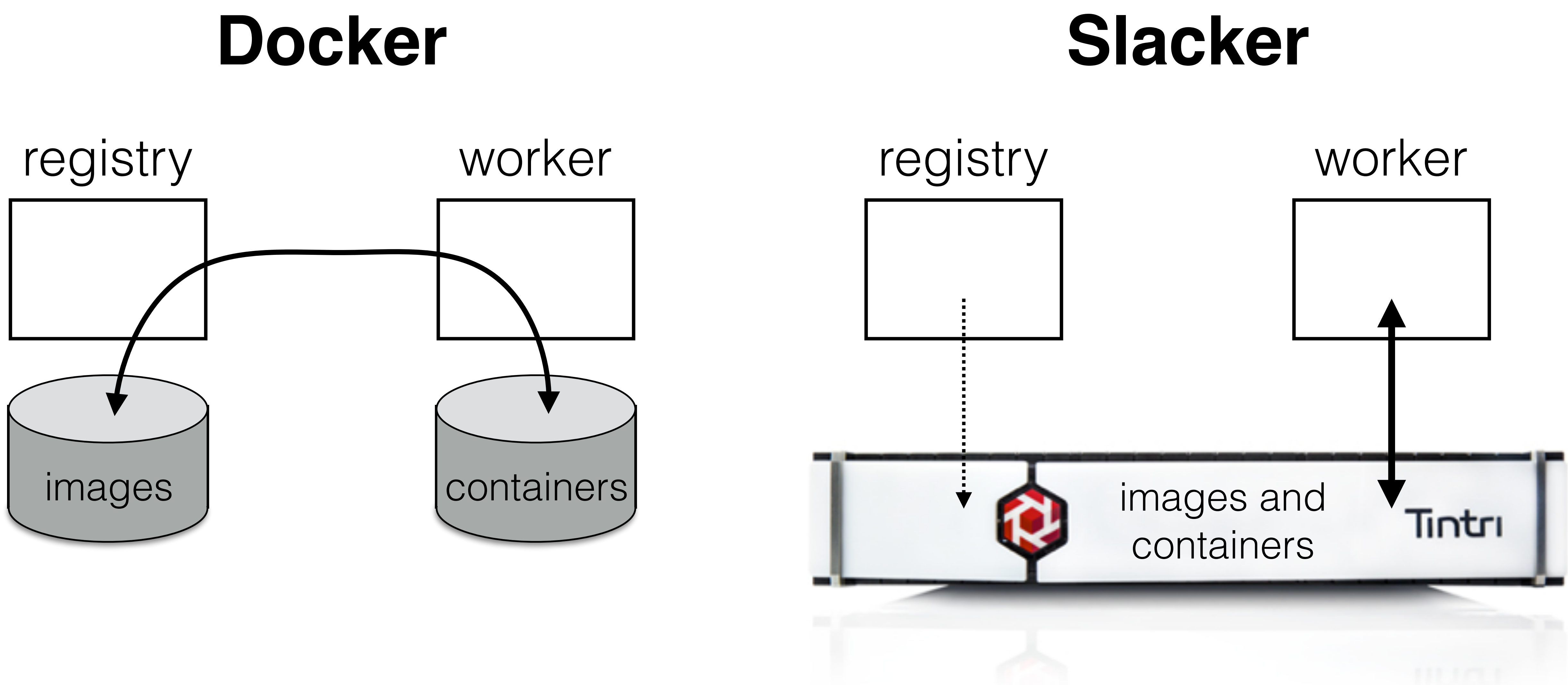
- make push+pull very fast
- create drop-in replacement; don't change Docker framework itself

Design

- lazily pull image data (like Nicolae *et al.* do for VMs)
- utilize COW primitives of Tintri VMstore backend (block level)

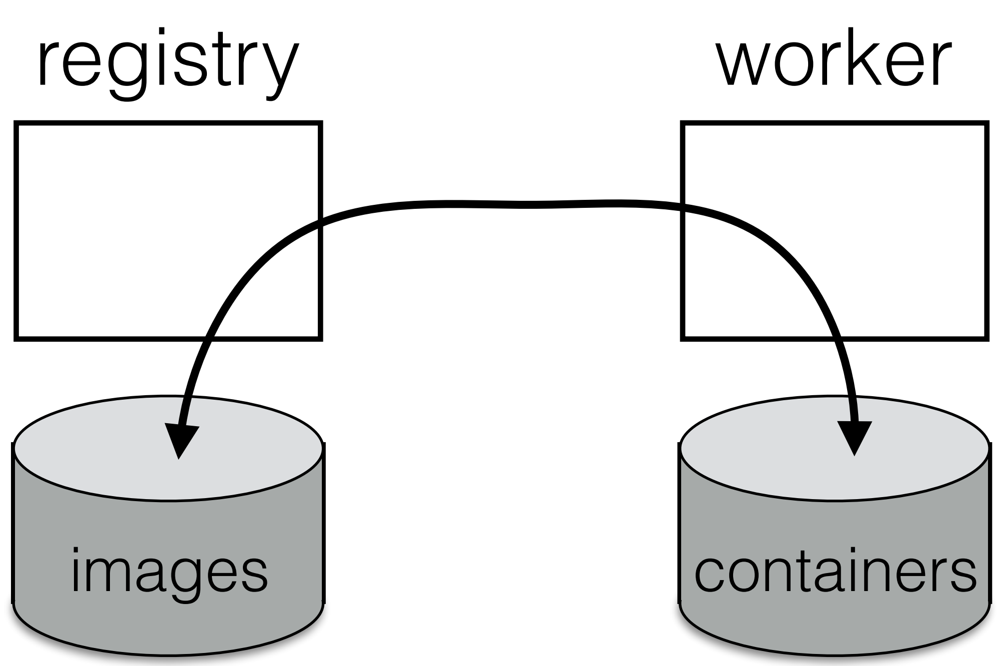


Prefetch vs. Lazy Fetch



Prefetch vs. Lazy Fetch

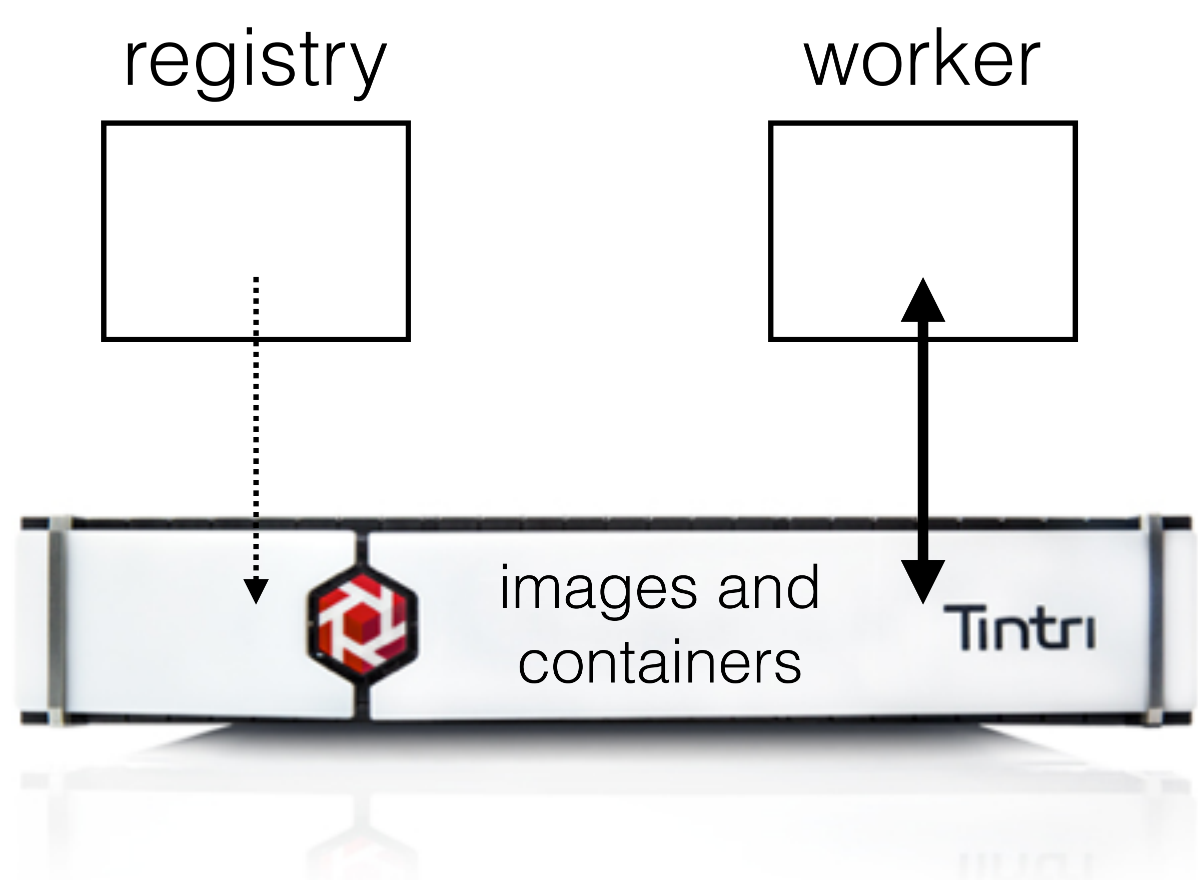
Docker



significant copying

- over network
- to/from disk

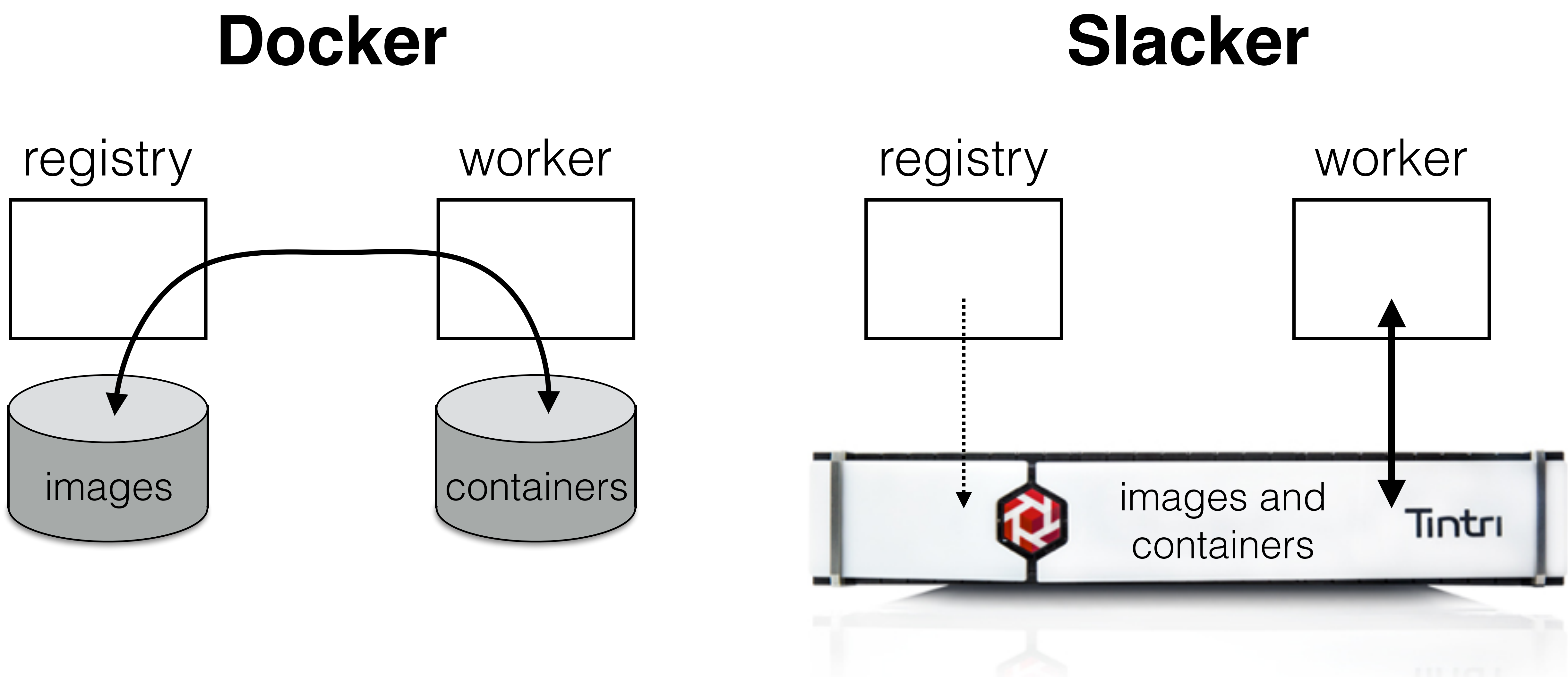
Slacker



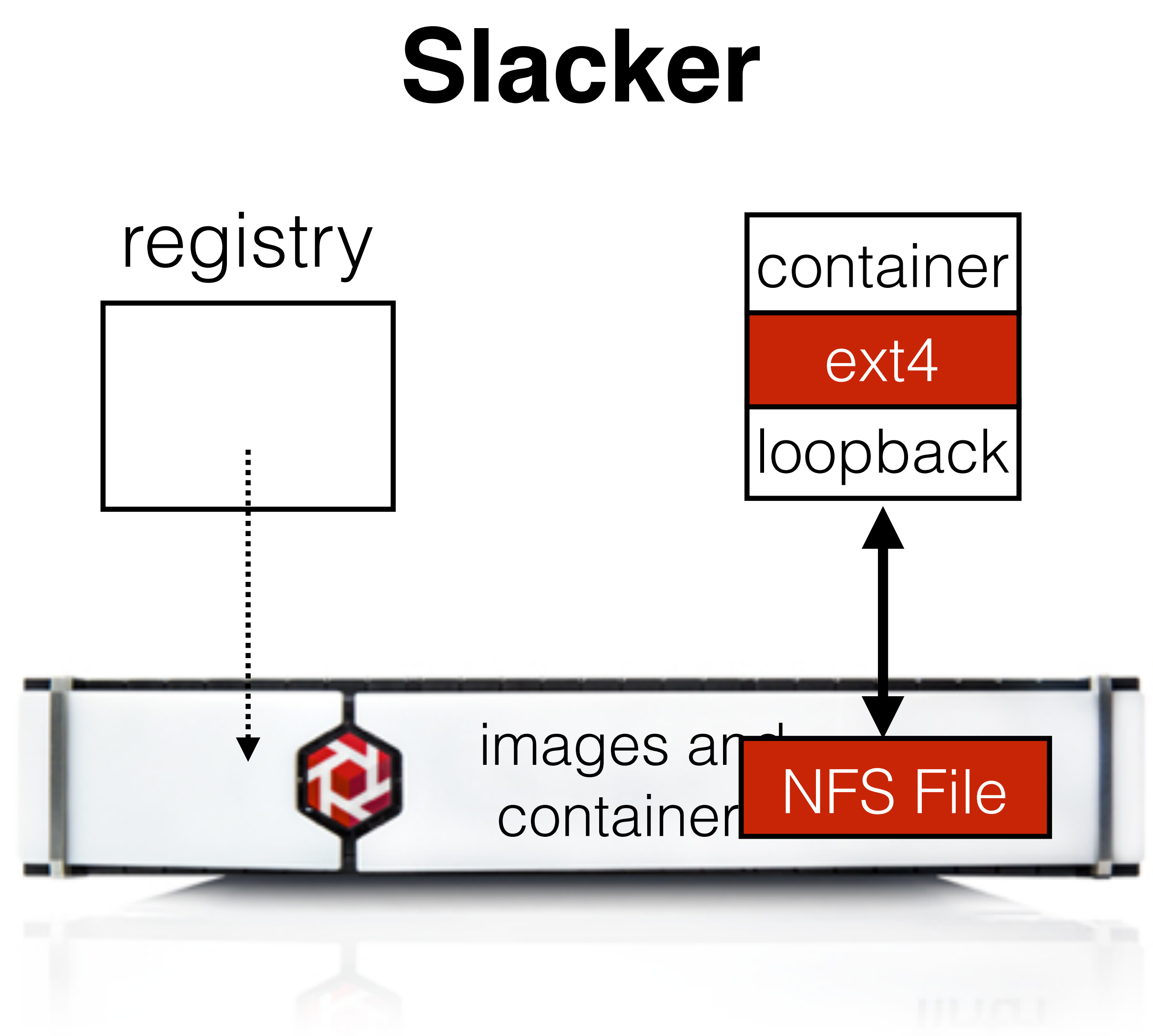
centralized storage

- easy sharing

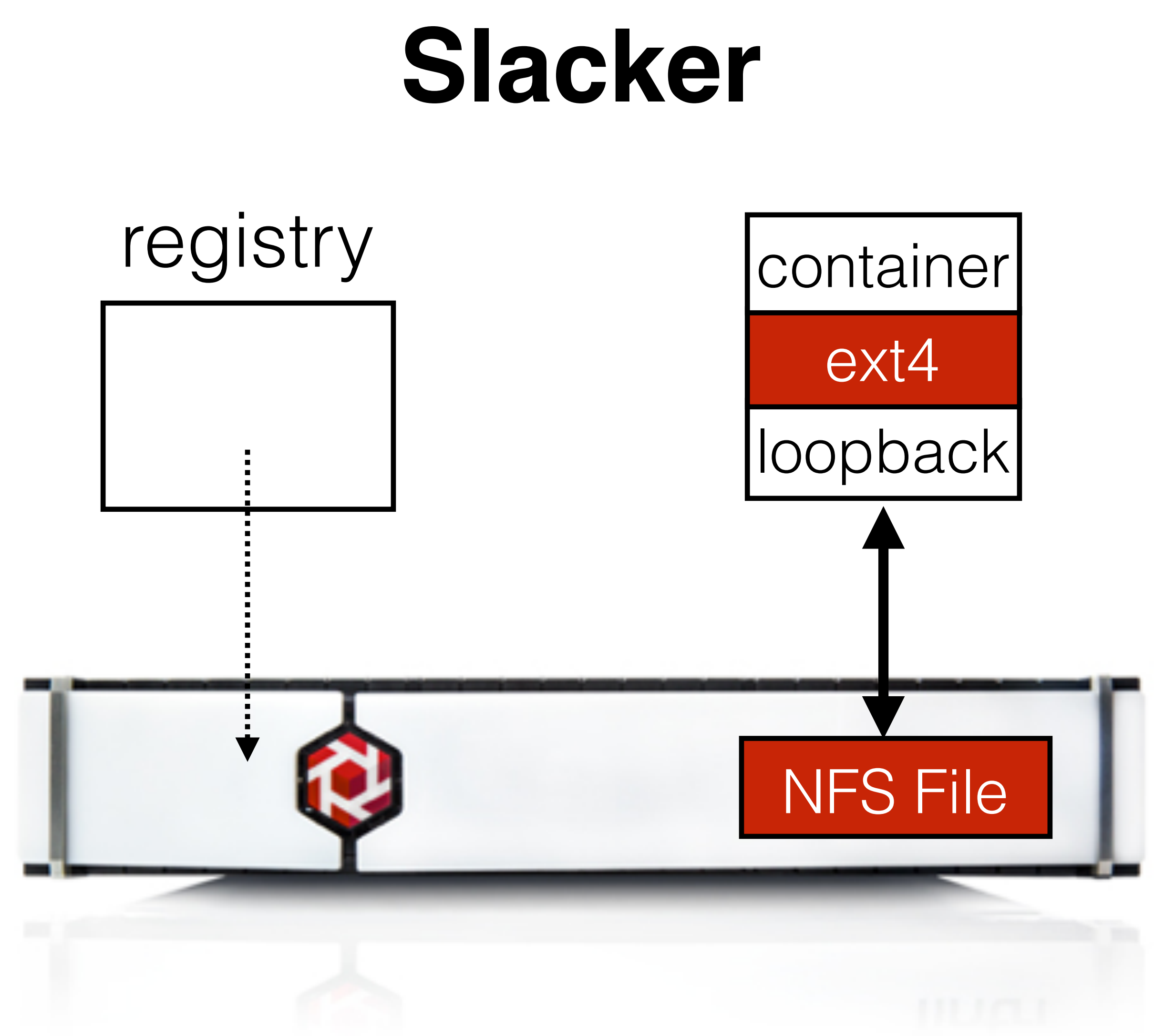
Prefetch vs. Lazy Fetch



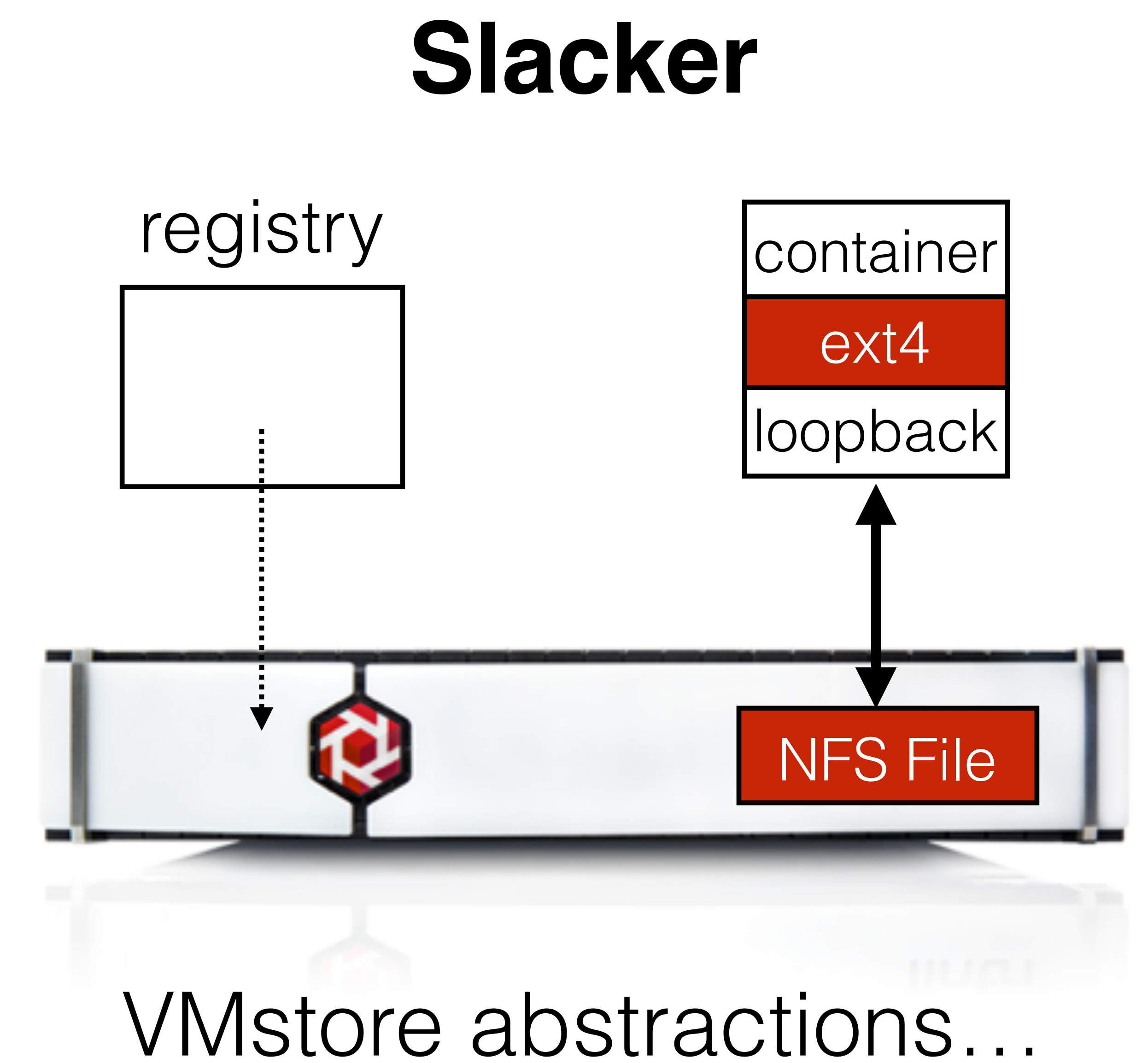
Prefetch vs. Lazy Fetch



Prefetch vs. Lazy Fetch



Prefetch vs. Lazy Fetch



VMstore Abstractions

Copy-on-Write

- VMstore provides `snapshot()` and `clone()`

`snapshot(nfs_path)`

- create read-only copy of NFS file
- return snapshot ID

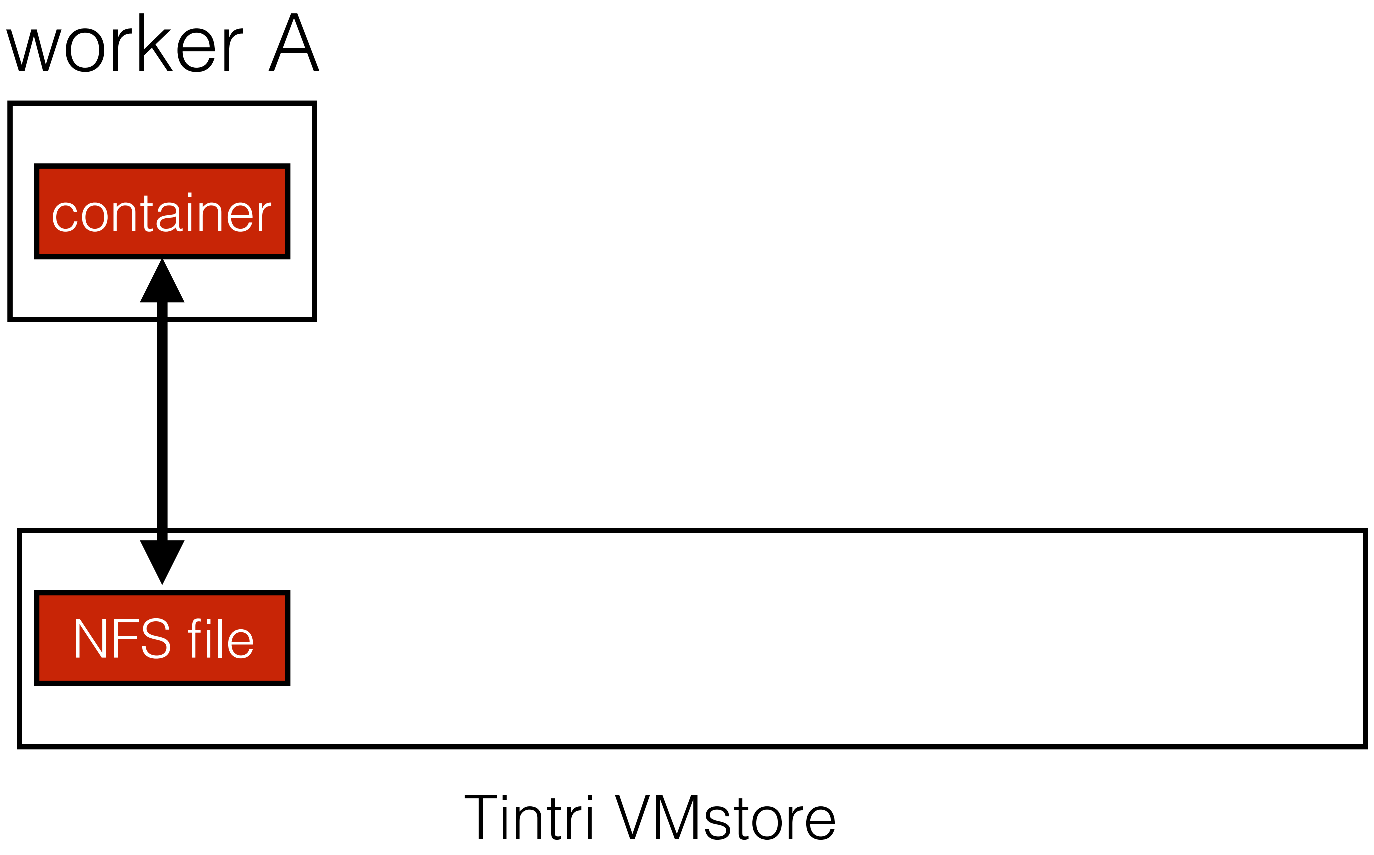
`clone(snapshot_id)`

- create r/w NFS file from snapshot

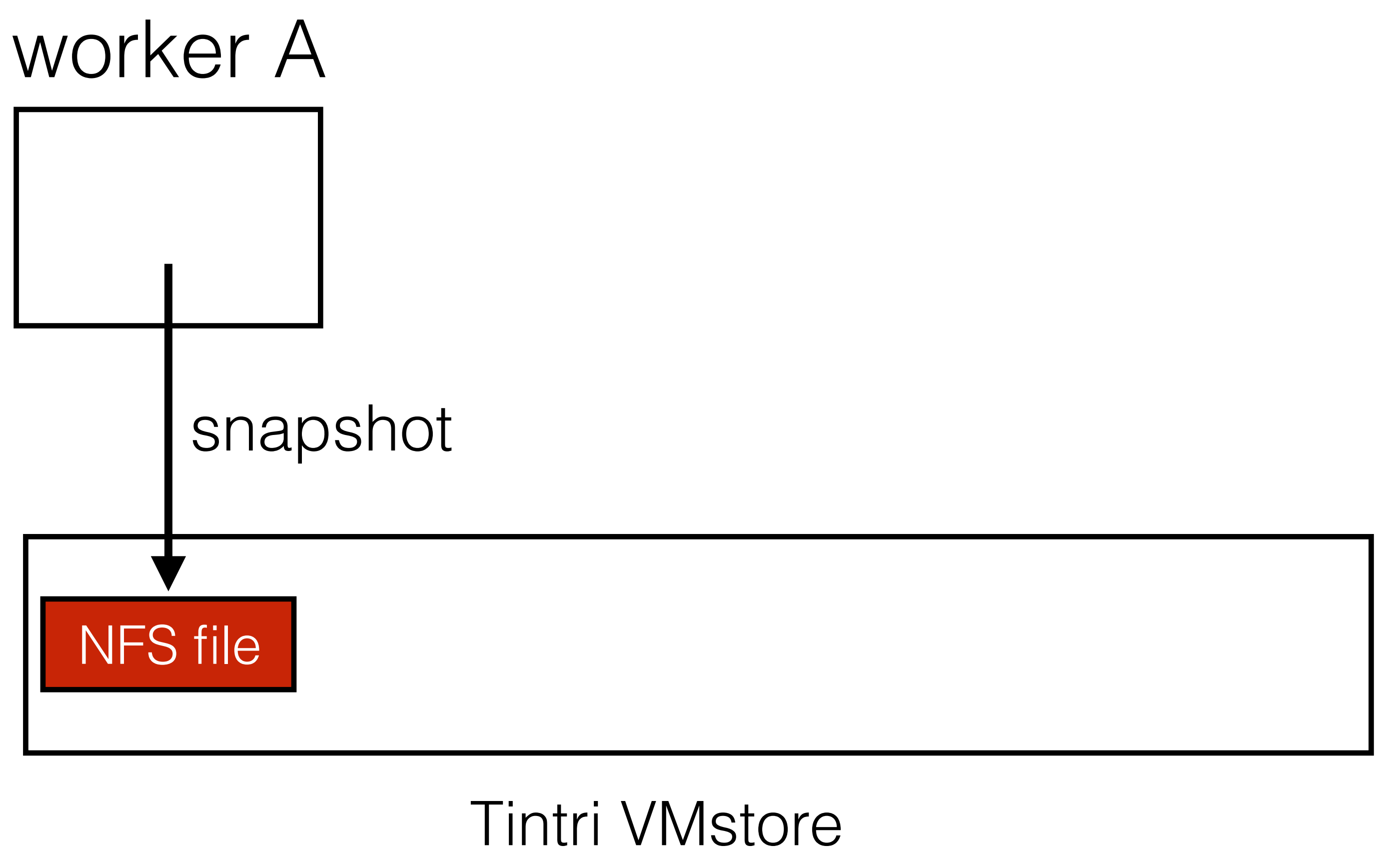
Slacker Usage

- NFS files ⇒ container storage
- snapshots ⇒ image storage
- `clone()` ⇒ provision container from image
- `snapshot()` ⇒ create image from container

Lazy Allocation

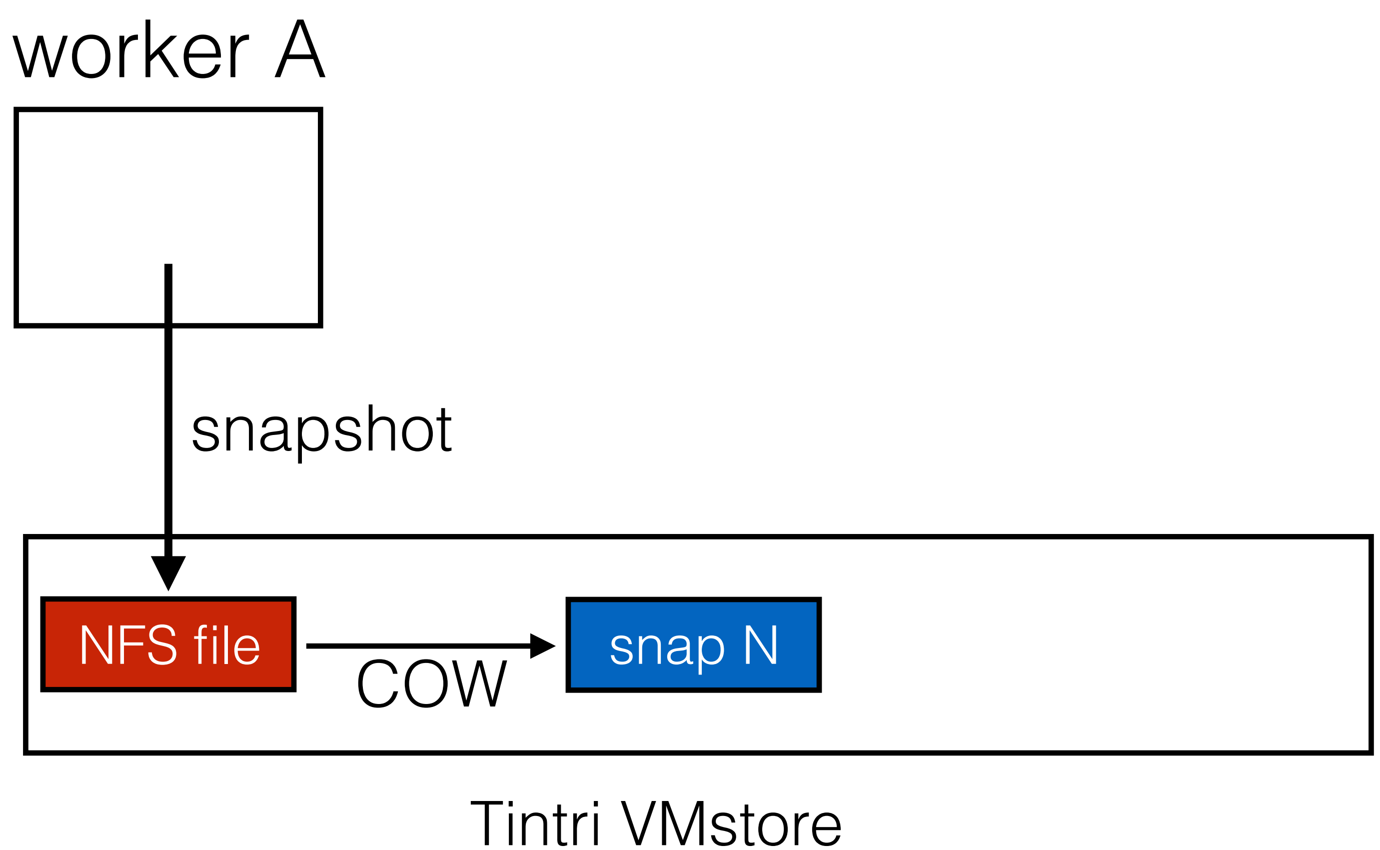


Lazy Allocation



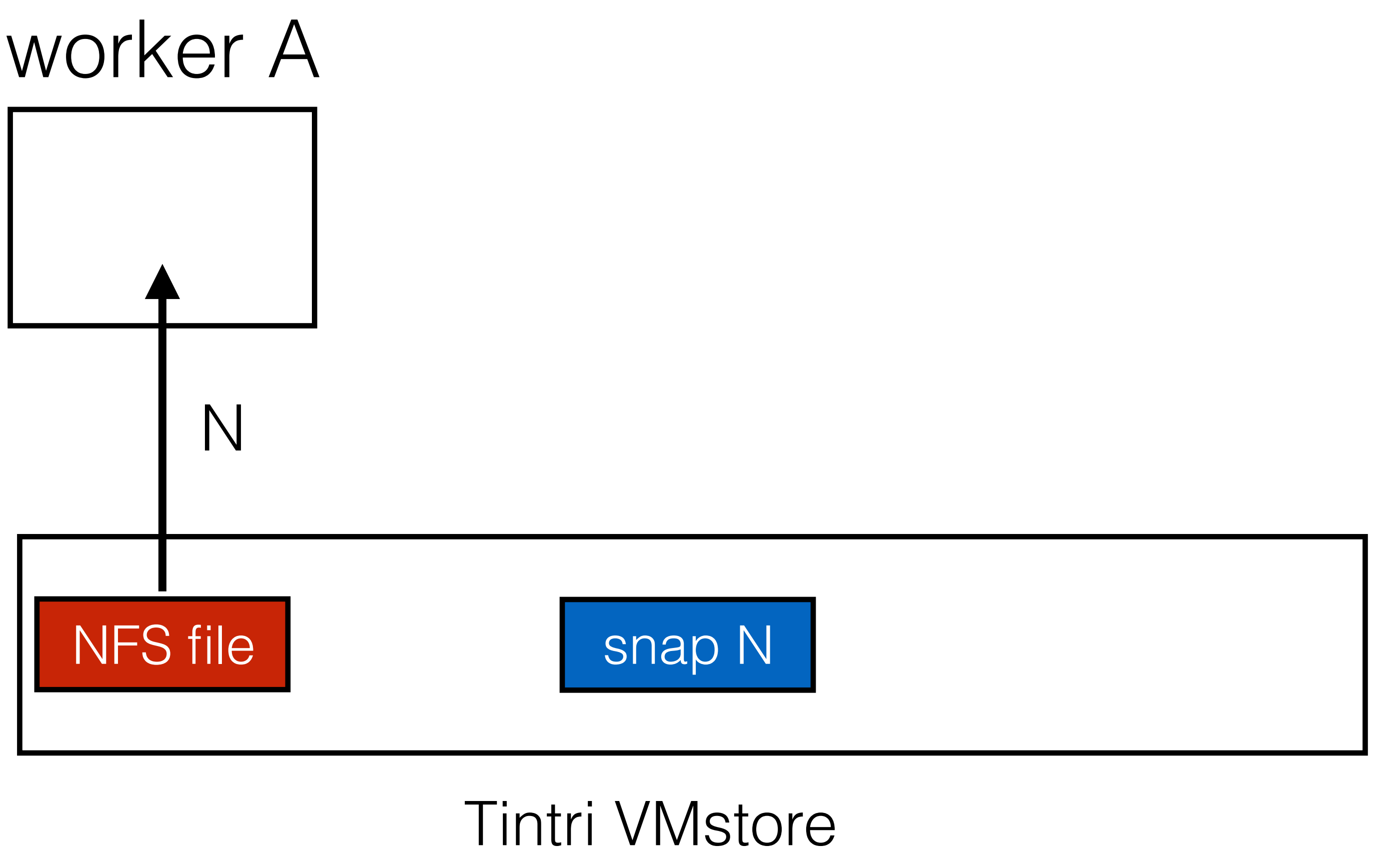
Worker A: push

Lazy Allocation



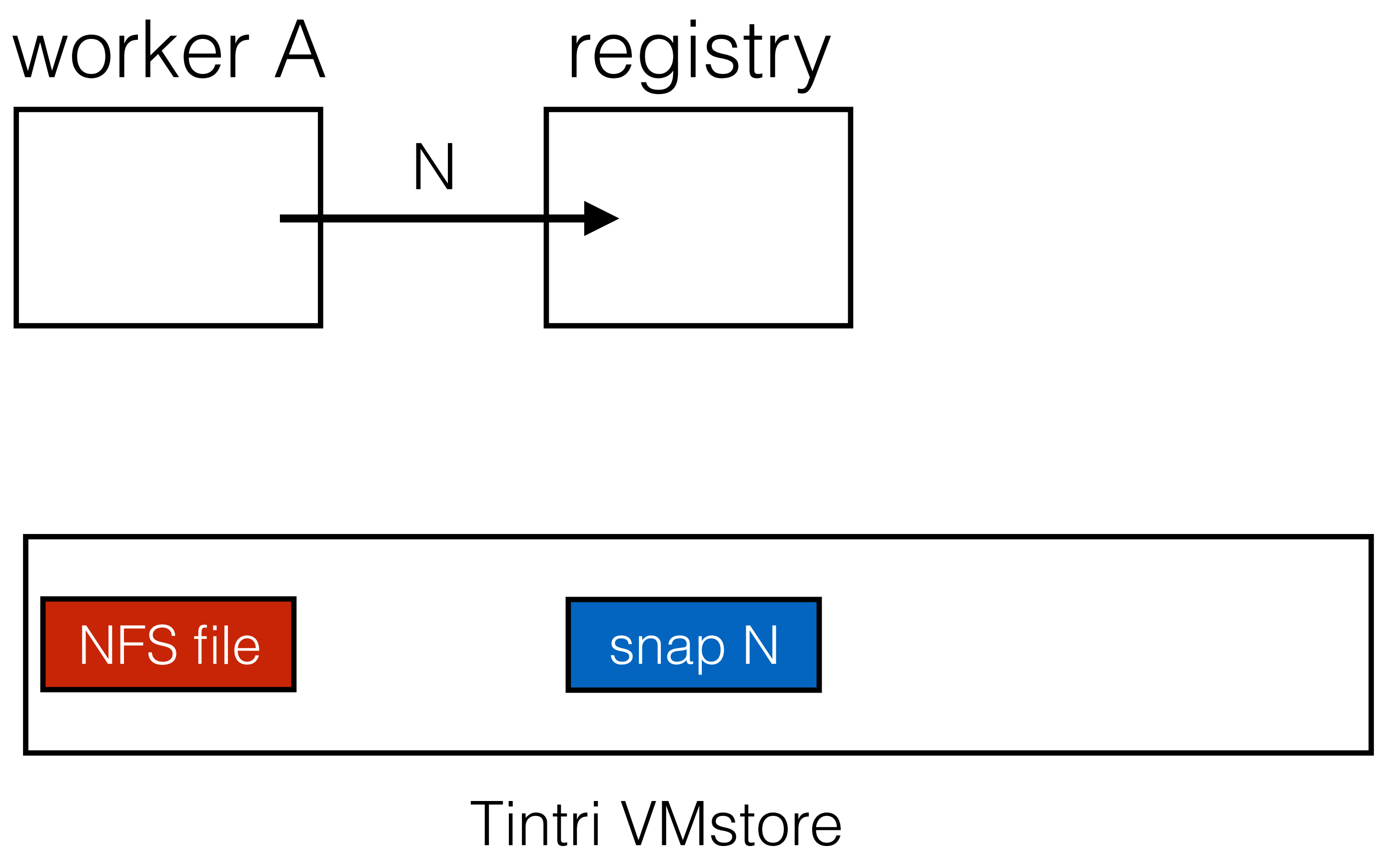
Worker A: push

Lazy Allocation



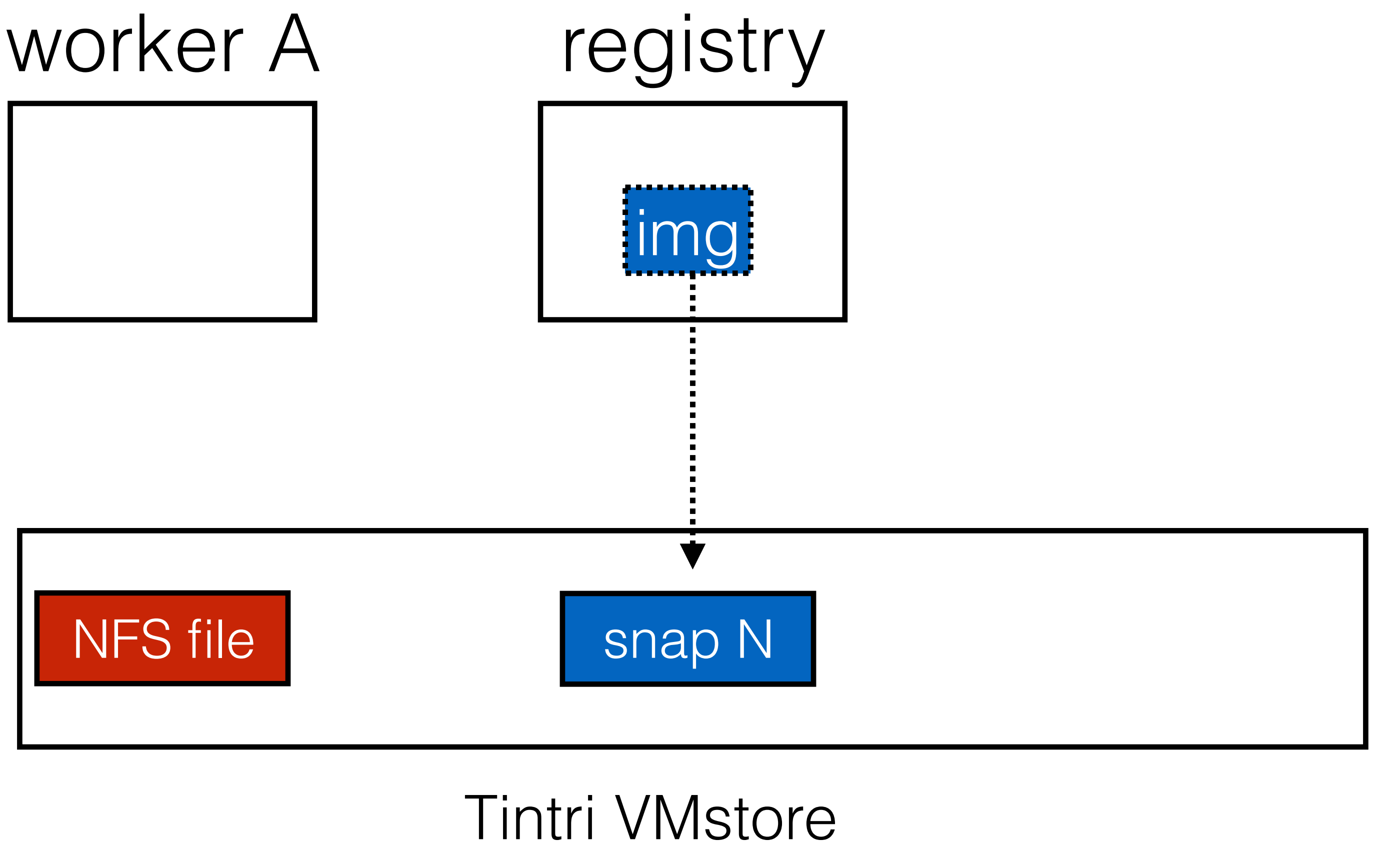
Worker A: push

Lazy Allocation



Worker A: push

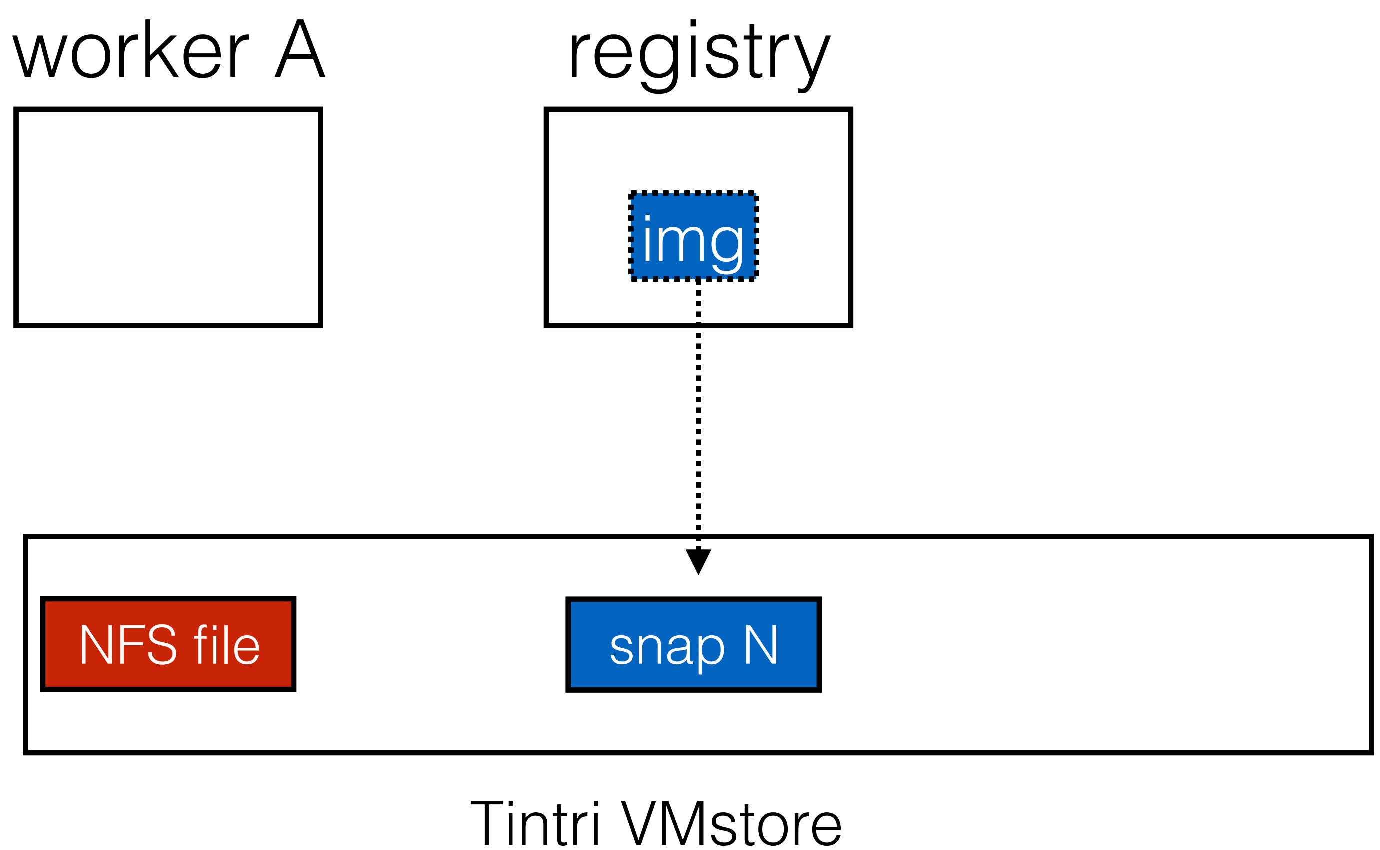
Lazy Allocation



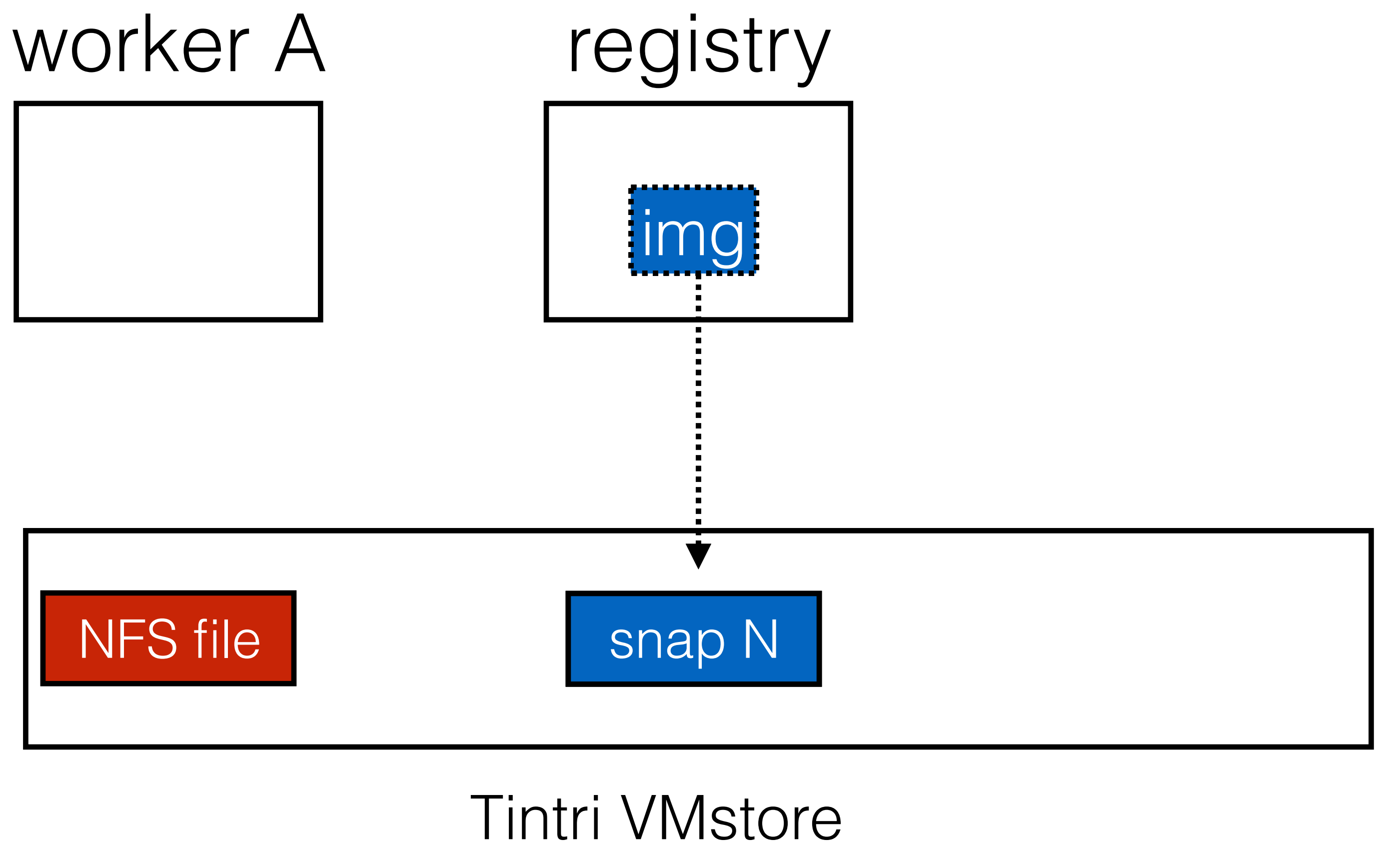
Worker A: push

Lazy Allocation

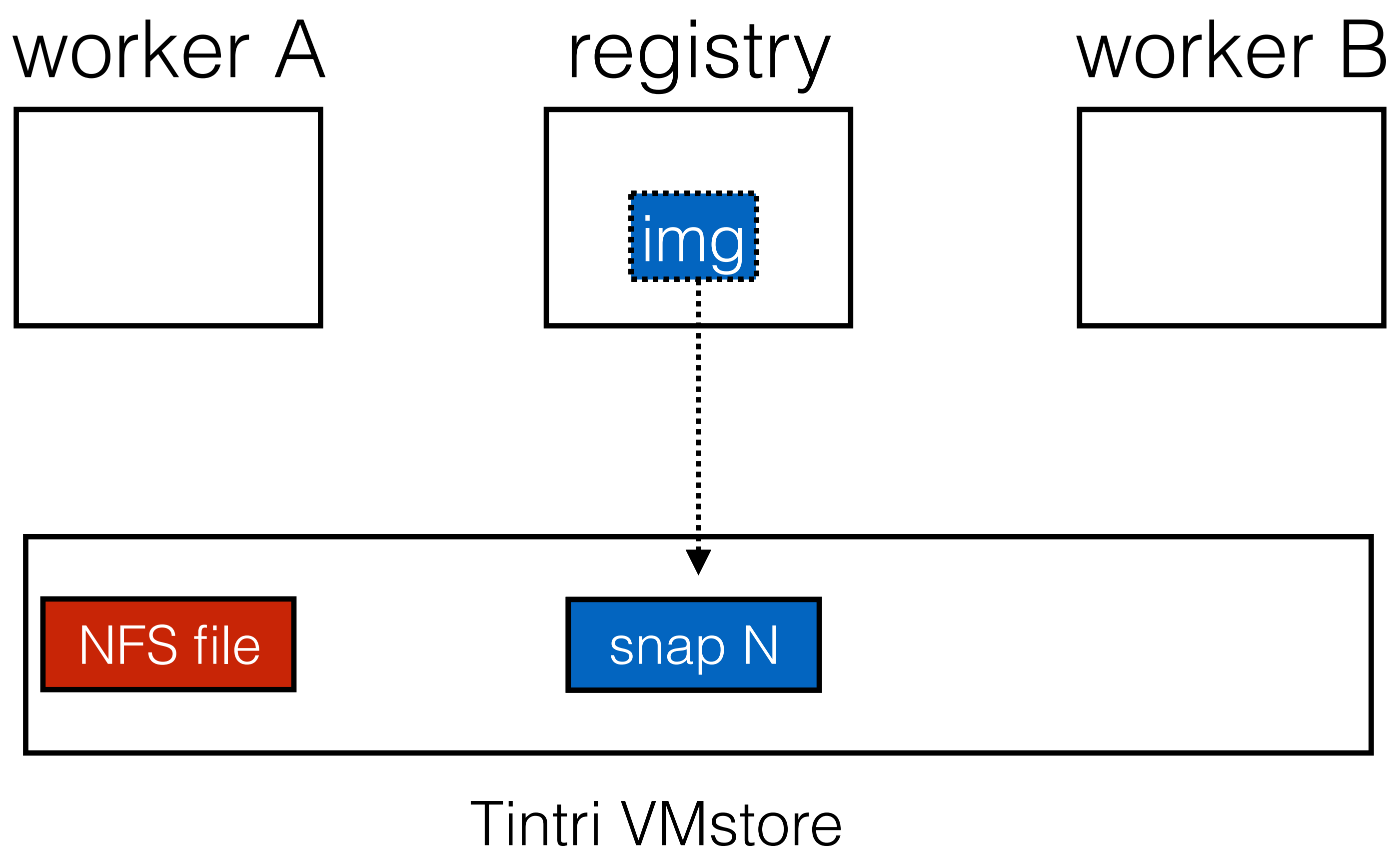
Note: registry is only a name server.
Maps layer metadata ⇒ snapshot ID



Lazy Allocation

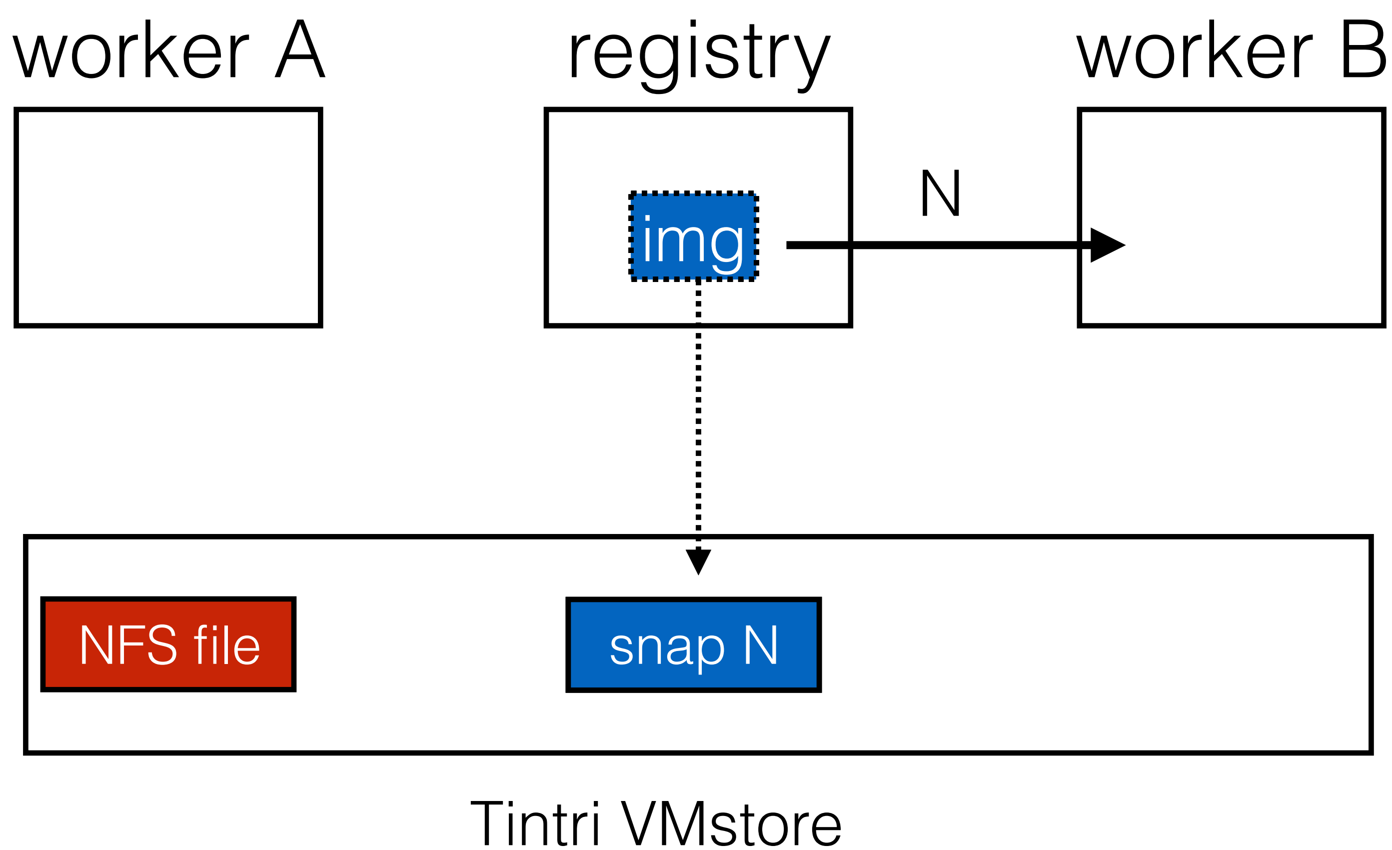


Lazy Allocation



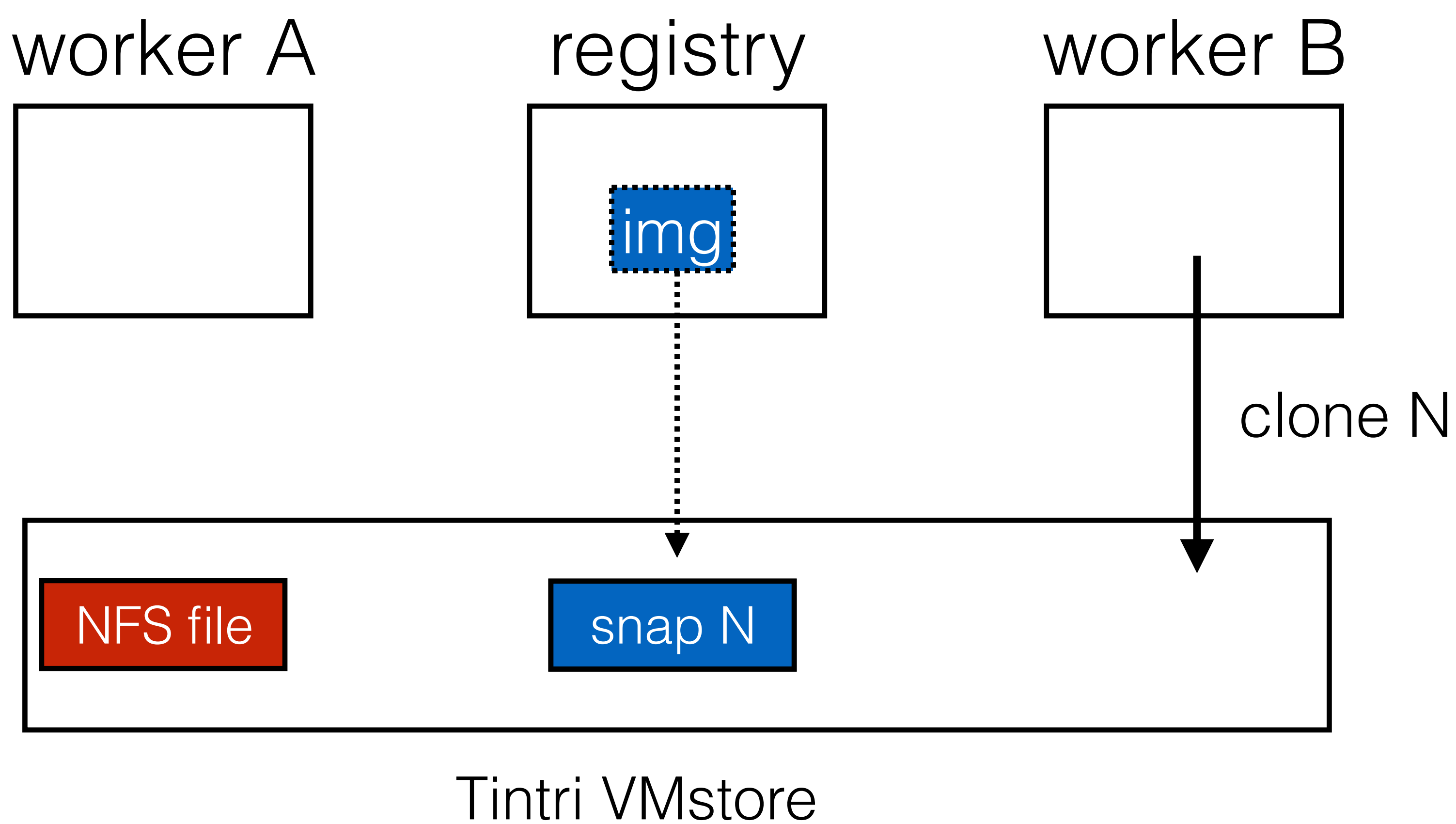
Worker B: pull and run

Lazy Allocation



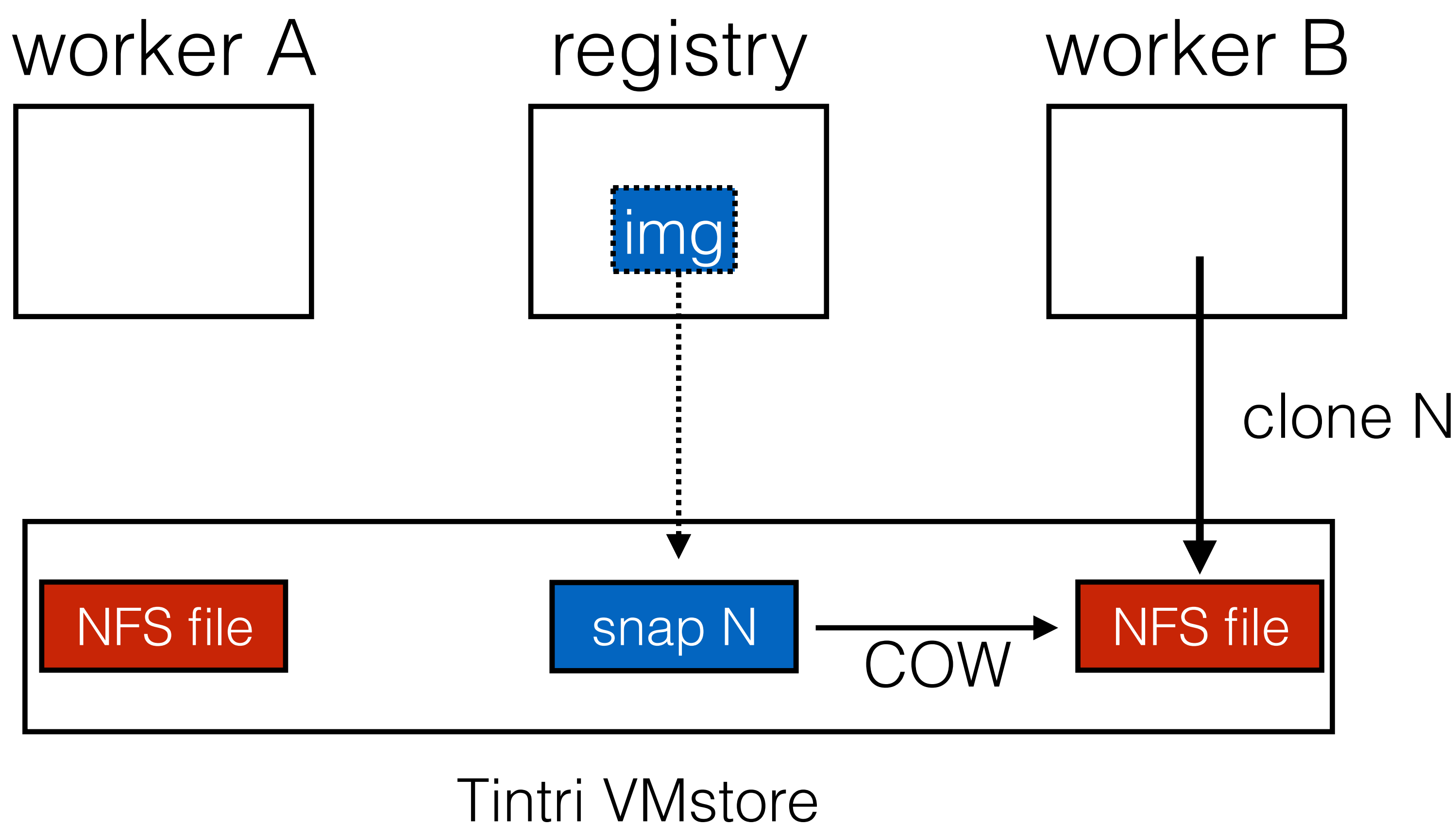
Worker B: pull and run

Lazy Allocation



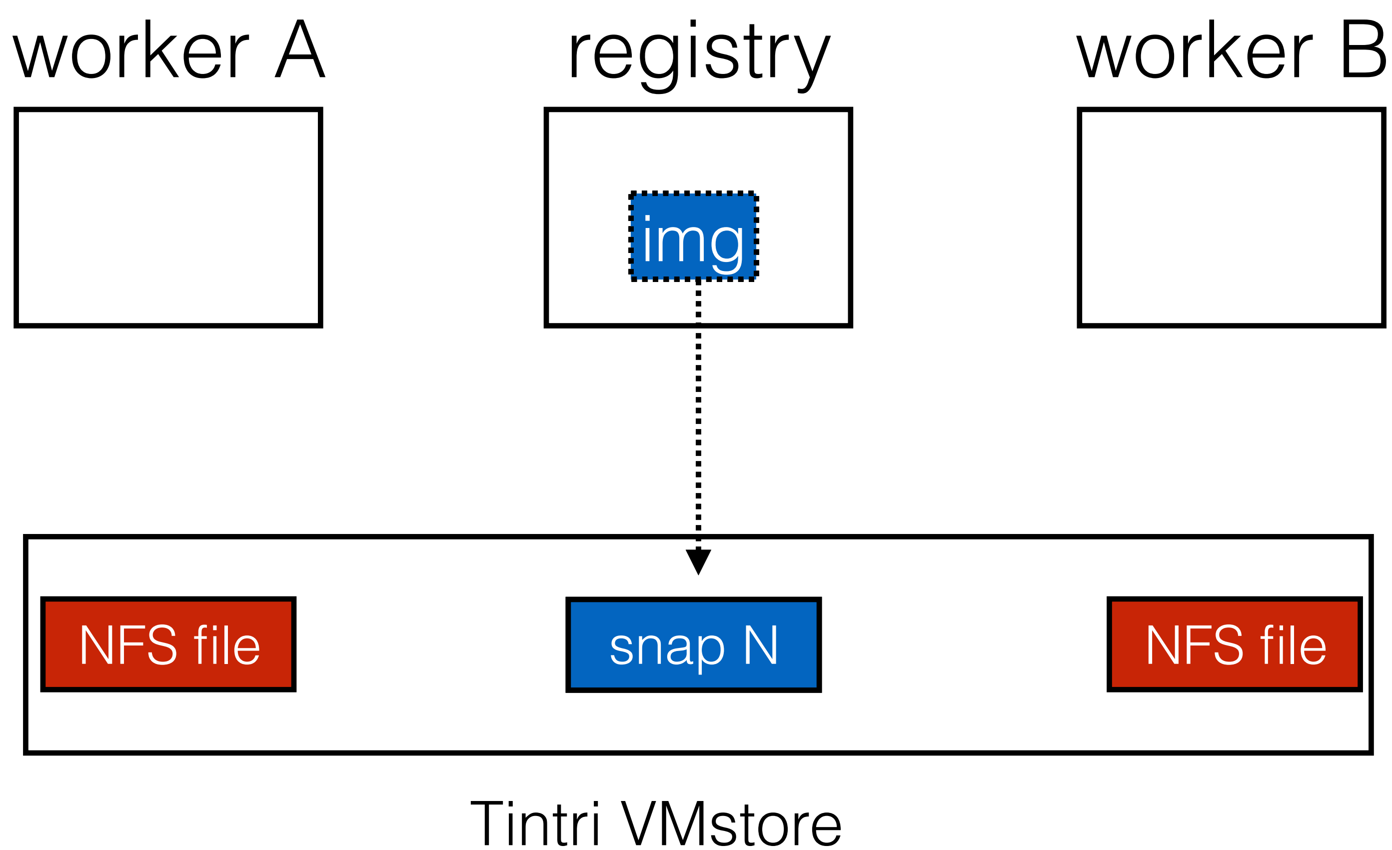
Worker B: pull and run

Lazy Allocation



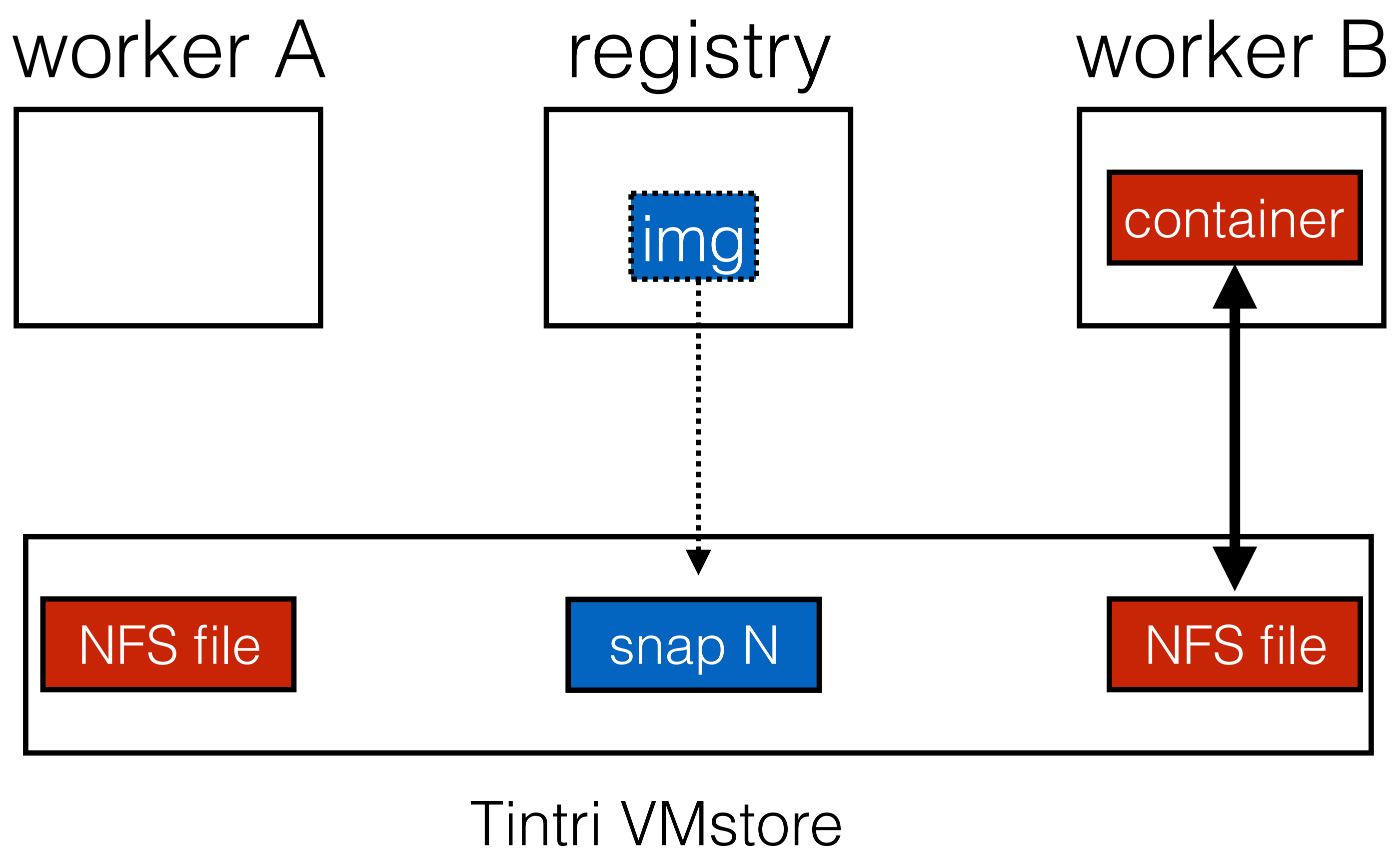
Worker B: pull and run

Lazy Allocation



Worker B: pull and run

Lazy Allocation



Worker B: pull and run

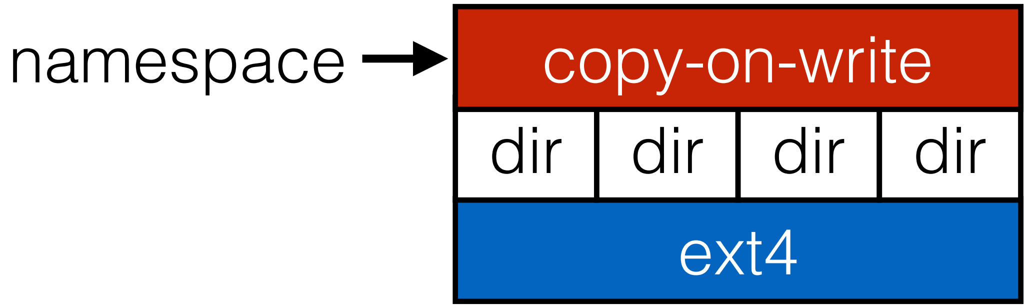
Indirection Discussion

File namespace level

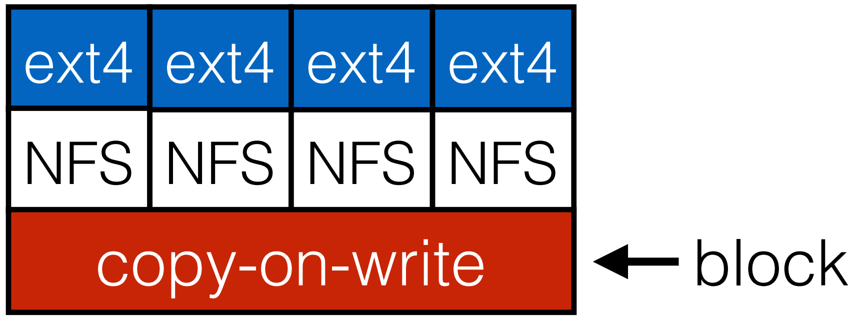
- flatten layers
- if B is child of A, then "copy" A to B to start. Don't make B empty

Block level

- do COW+dedup beneath NFS files, inside VMstore



AUFS



Slacker

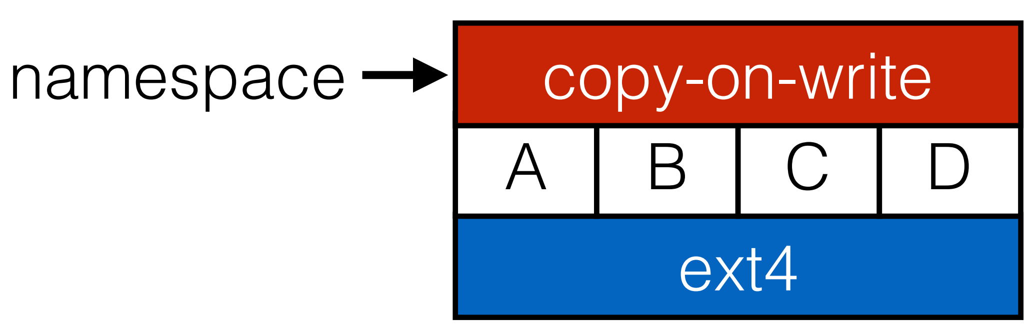
Indirection Discussion

File namespace level

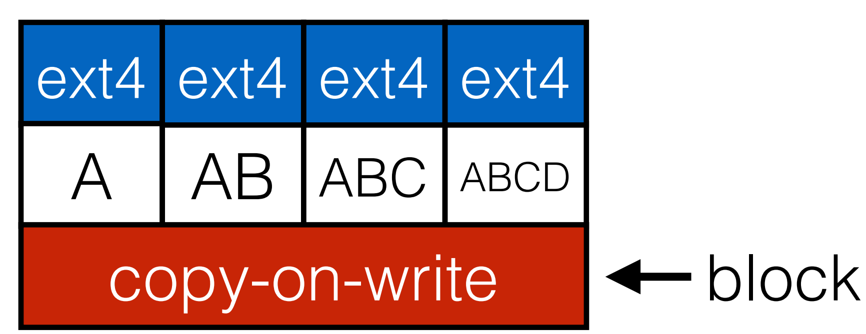
- flatten layers
- if B is child of A, then “copy” A to B to start. Don’t make B empty

Block level

- do COW+dedup beneath NFS files, inside VMstore

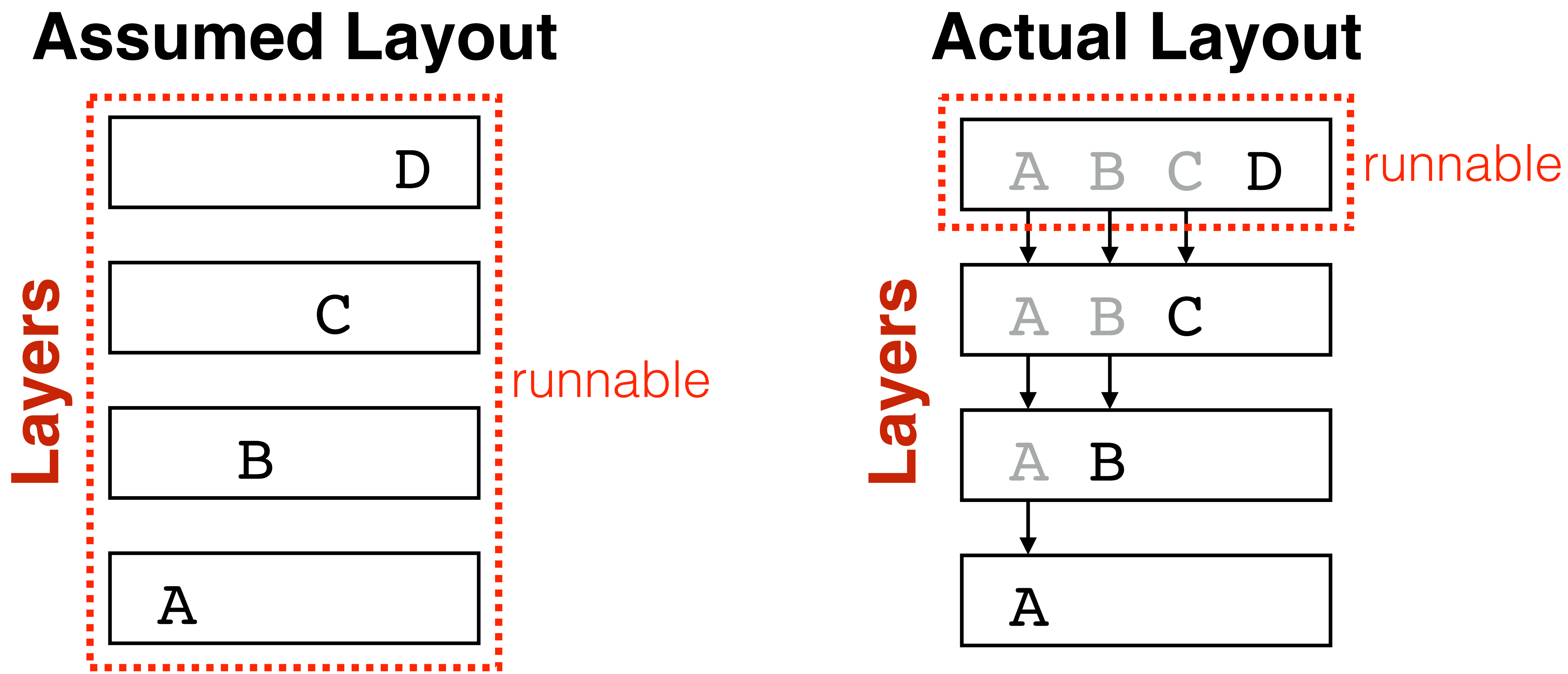


AUFS

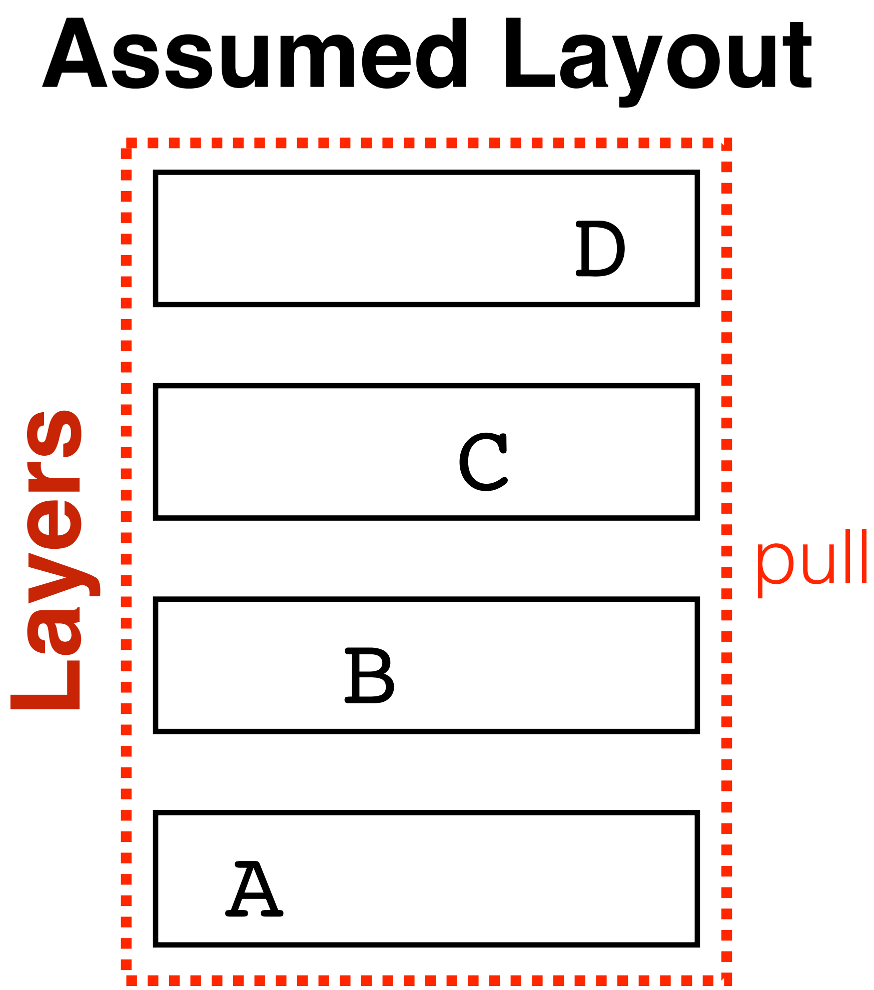


Slacker

Challenge: Framework Assumptions

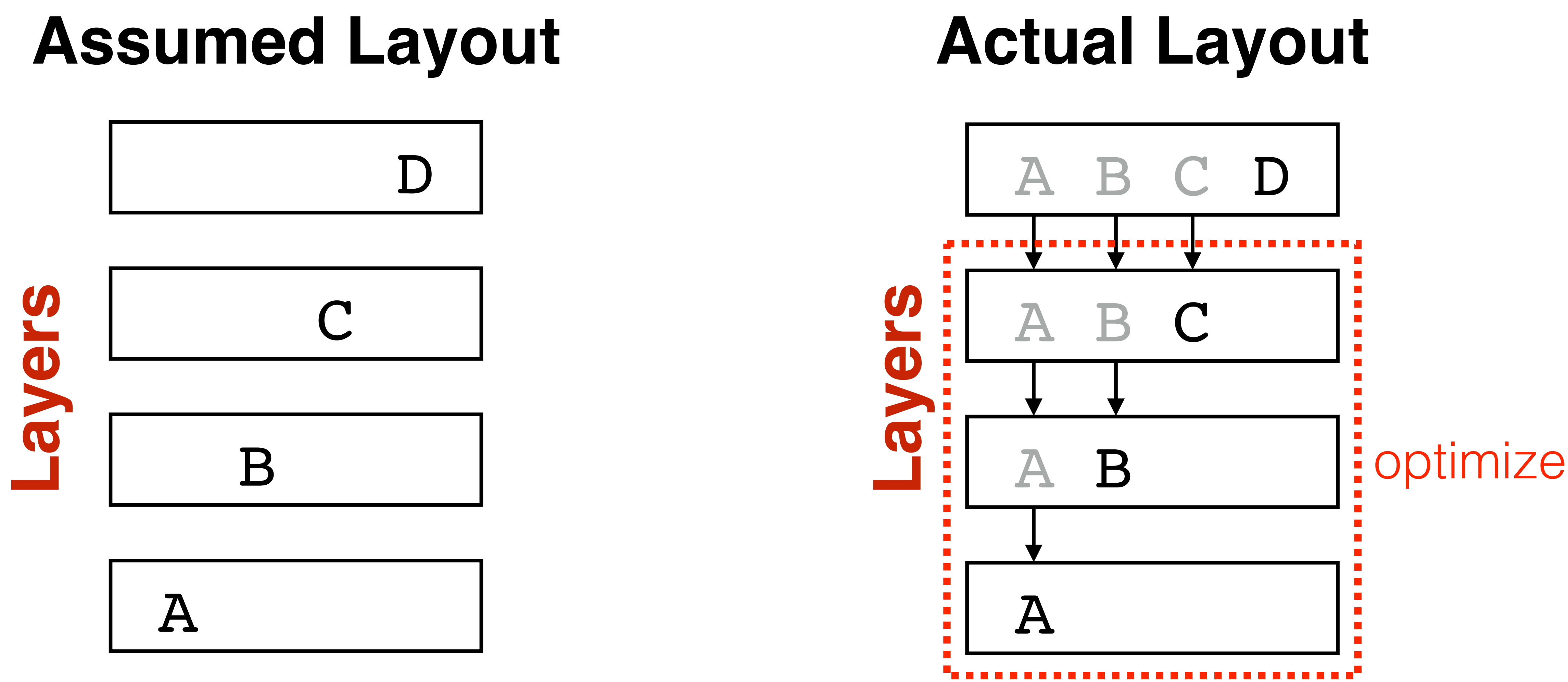


Challenge: Framework Assumptions



Challenge: Framework Assumptions

Strategy: **lazy cloning**. Don't clone non-top layers until Docker tries to mount them.



Slacker Outline

AUFS Storage Driver Background

Slacker Design

Evaluation

Questions

What are deployment and development speedups?

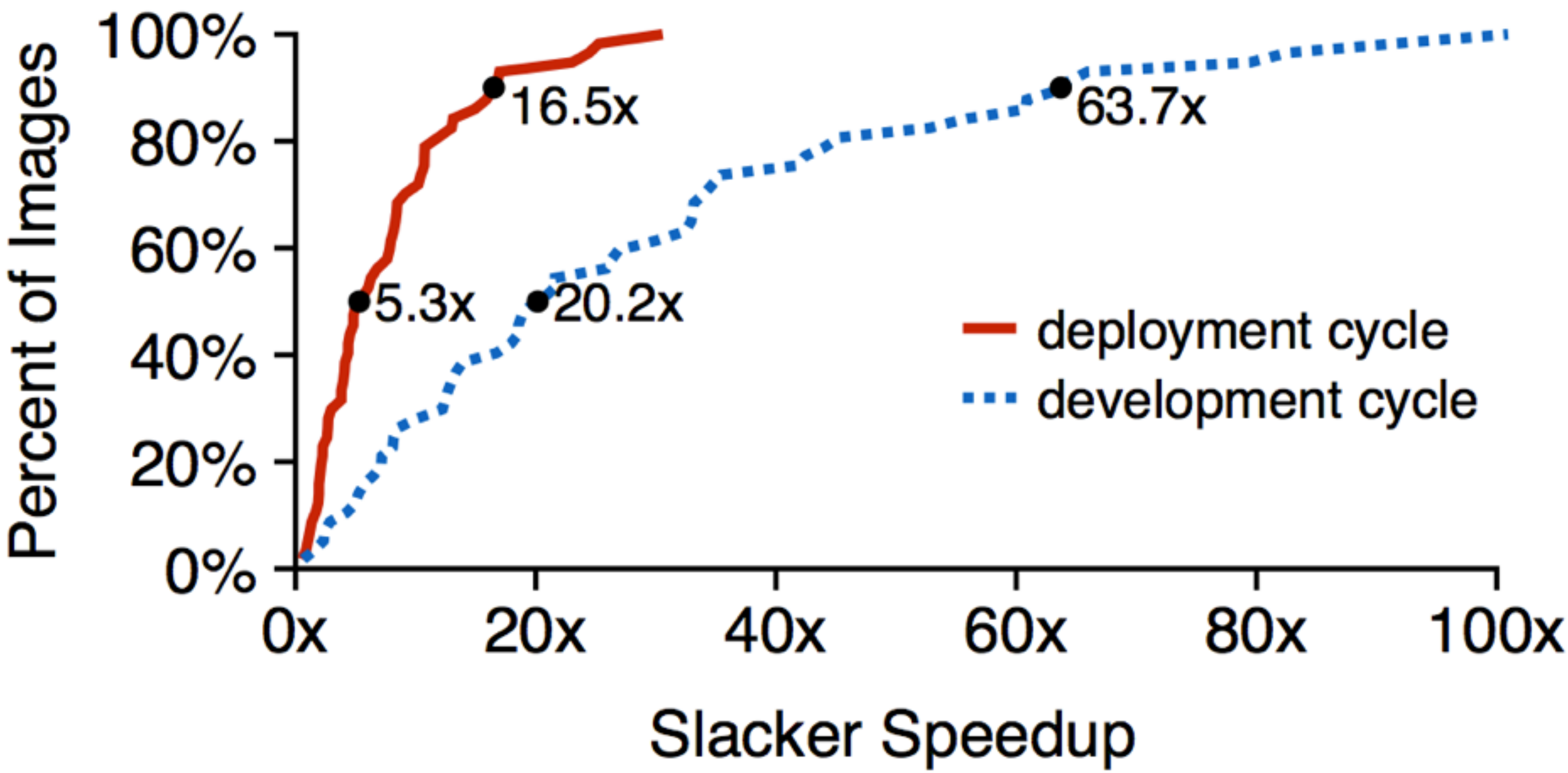
How is long-term performance?

Questions

What are deployment and development speedups?

How is long-term performance?

HelloBench Performance



deployment: pull+run
development: push+pull+run

Questions

What are deployment and development speedups?

- 5x and 20x faster respectively (median speedup)

How is long-term performance?

Questions

What are deployment and development speedups?

- 5x and 20x faster respectively (median speedup)

How is long-term performance?

Server Benchmarks

Databases and web servers

- PostgreSQL
- Redis
- Apache web server (static)
- io.js Javascript server (dynamic)

Experiment

- measure throughput (after startup)
- run 5 minutes

Server Benchmarks

Databases and web servers

- PostgreSQL
- Redis
- Apache web server (static)
- io.js Javascript server (dynamic)

Experiment

- measure throughput (after startup)
- run 5 minutes

Result: Slacker is always at least as fast as AUFS

Questions

What are deployment and development speedups?

- 5x and 20x faster respectively (median speedup)

How is long-term performance?

- there is no long-term penalty for being lazy

Slacker Conclusion

Containers are inherently lightweight

- but existing frameworks are not

COW between workers is necessary for fast startup

- use shared storage
- utilize VMstore snapshot and clone

Slacker driver

- **5x** deployment speedup
- **20x** development speedup

Outline

Motivation: Modularity in Modern Storage

Overview: Types of Modularity

Library Study: Apple Desktop Applications

Layer Study: Facebook Messages

Microservice Study: Docker Containers

Slacker: a Lazy Docker Storage Driver

Conclusions

Modularity Often Causes Unnecessary I/O

Measurement exposed undesirable emergent properties

Libraries cause iBench applications to excessively flush

Layers cause Facebook Messages to waste network I/O

Microservice provisioning unnecessary copying

Layers Mask Costs

Apple desktop

- Key/value layer causes excessive fsync/rename
- SQLite use caused excessive fine-grained locking, rendered unnecessary by higher-level exclusion

Facebook Messages

- composition of layers amplifies writes from 1% to 64% of total I/O

Docker containers

- AUFS access surprisingly expensive to deep data

Simple Integration Surprisingly Useful

Measurement-driven optimizations are surprisingly effective at mitigating the cost of modularity

Local compaction

- reduces network I/O by **2.7x**

Combined logging

- reduces log latency by **6x**

Lazy propagation

- reduces container startup latency by **5x**

Thank you!