[537] GFS+MapReduce

Tyler Harter
12/01/14

Google File System

GFS

Goal: present global FS that stores data across many machines. Need to handle 100's TBs.

Contrast: NFS only exports a local FS on one machine to other machines.

Google published details in 2003.

Open source implementation: Hadoop FS (HDFS)

Failure: NFS Comparison

NFS only recovers from temporary failure.
- not permanent disk/server failure
- recover means making reboot invisible
- technique: retry
  (stateless and idempotent protocol helps)

GFS needs to handle permanent failure.
- techniques: replication and failover (like RAID)

Measure Then Build

Google workload characteristics:

- huge files (GBs)
- almost all writes are appends
- concurrent appends common
- high throughput is valuable
- low latency is not

Example Workloads

MapReduce
- read entire dataset, do computation over it

Producer/consumer
- many producers append work to file concurrently
- one consumer reads and does work

Example Workloads

MapReduce
- read entire dataset, do computation over it

Producer/consumer
- many producers append work to file concurrently
- one consumer reads and does work
- append not idempotent, is work idempotent?

Co-design

Opportunity to build FS and application together.

Make sure applications can deal with FS quirks.

Avoid difficult FS features:
- read dir
- links

Special features: snapshot, atomic append

GFS Overview

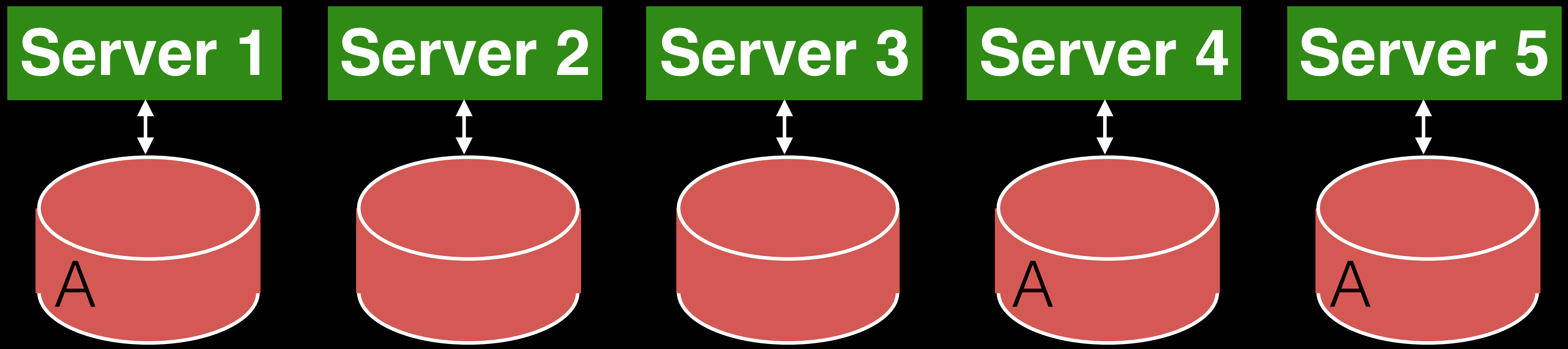
Motivation

Architecture

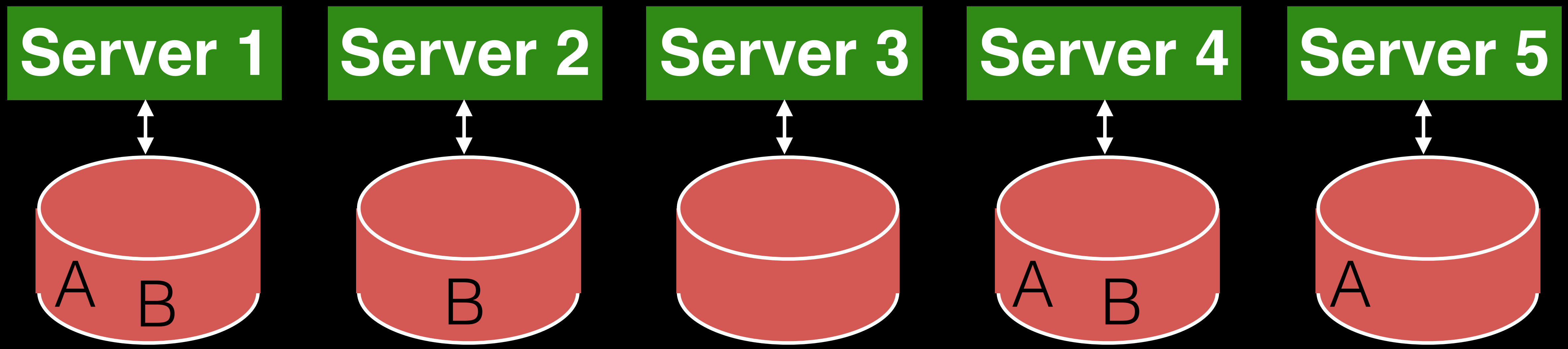
Master metadata

Worker data

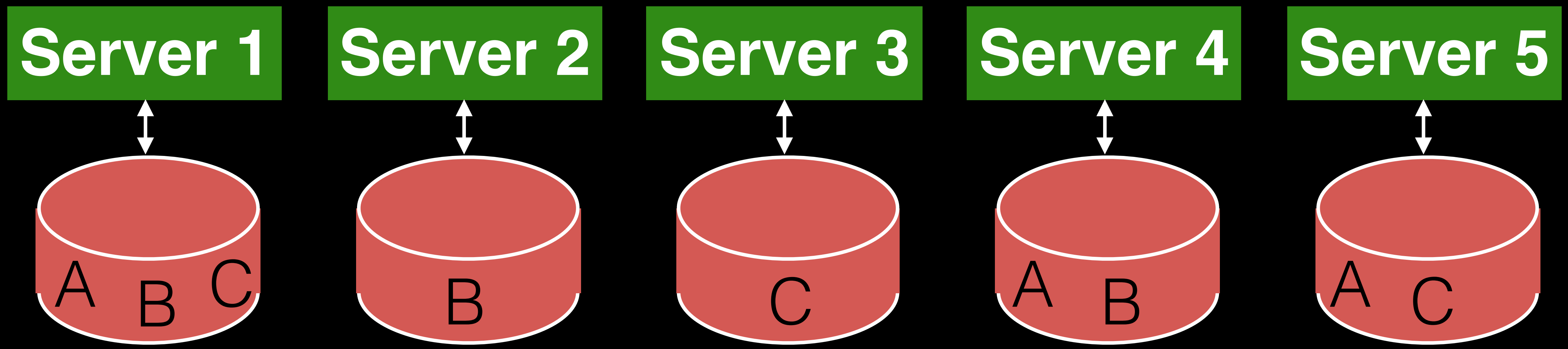
Replication



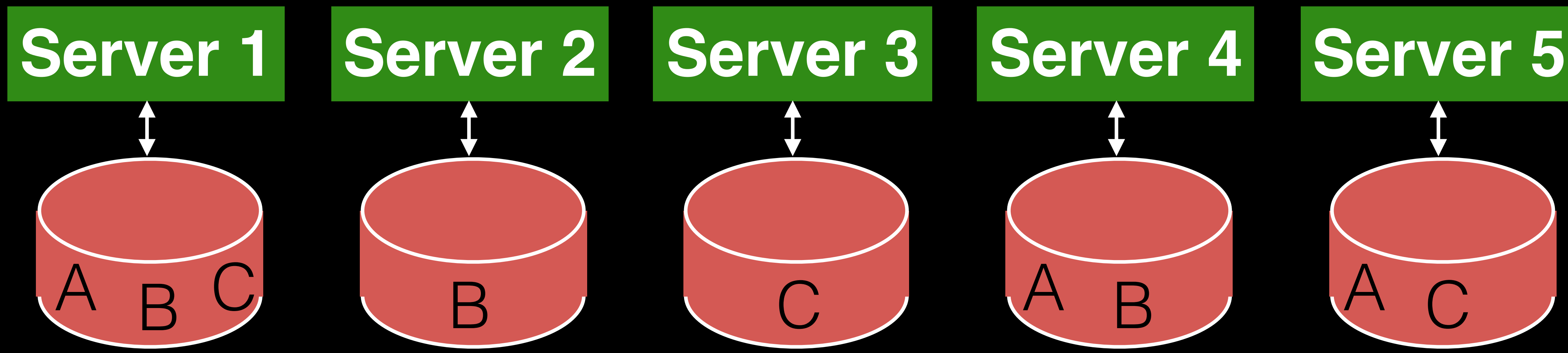
Replication



Replication

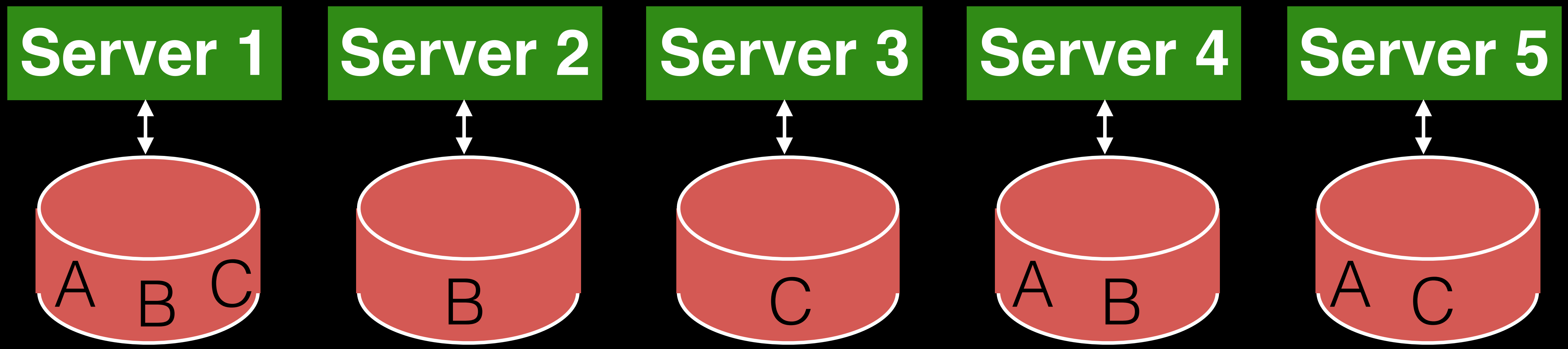


Replication



Less orderly than RAID:
- machines come and go, capacity may vary
- different data may have different replication

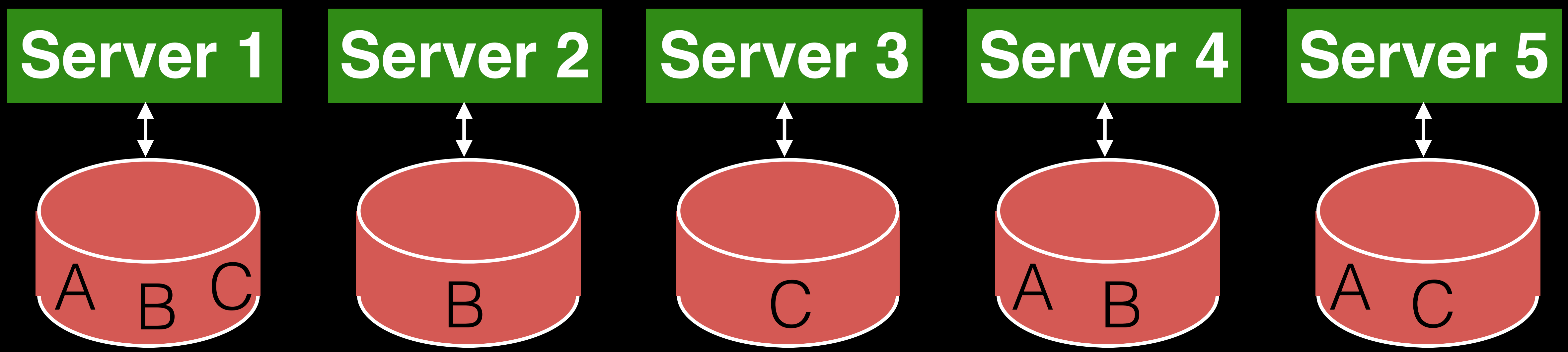
Replication



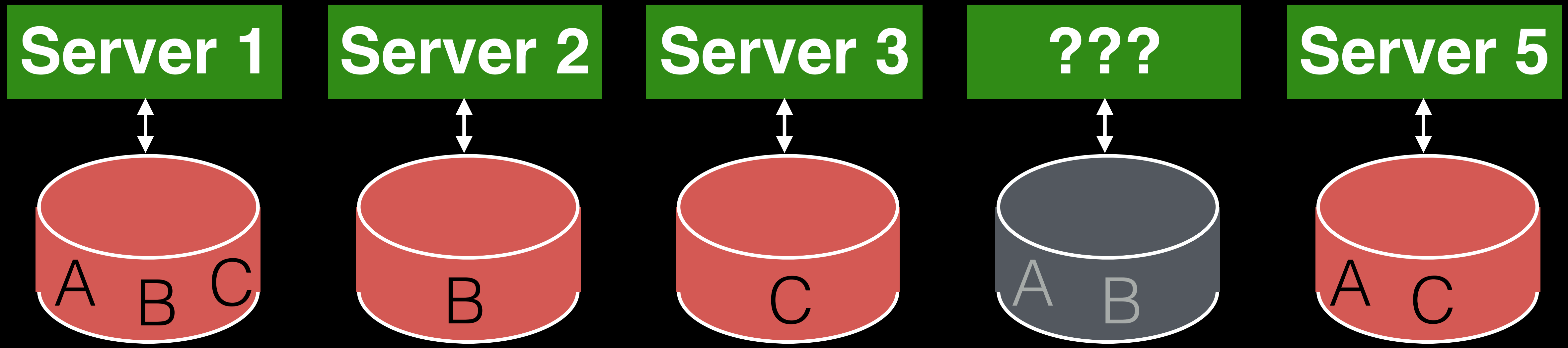
Less orderly than RAID:

- machines come and go, capacity may vary
- different data may have different replication
- how to map logical to physical?

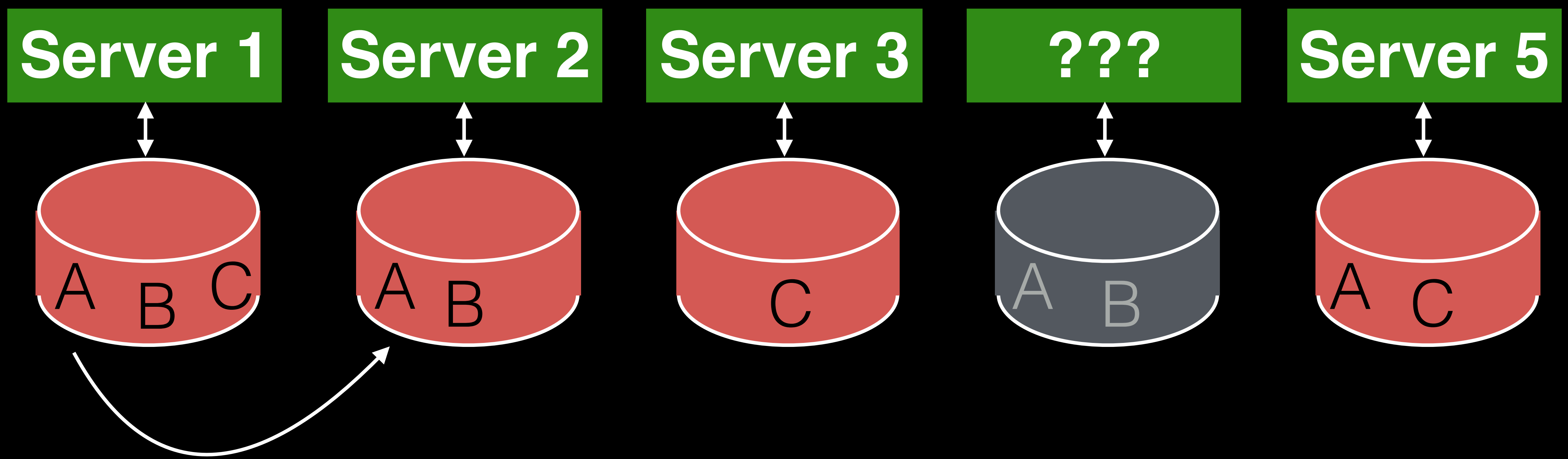
Recovery



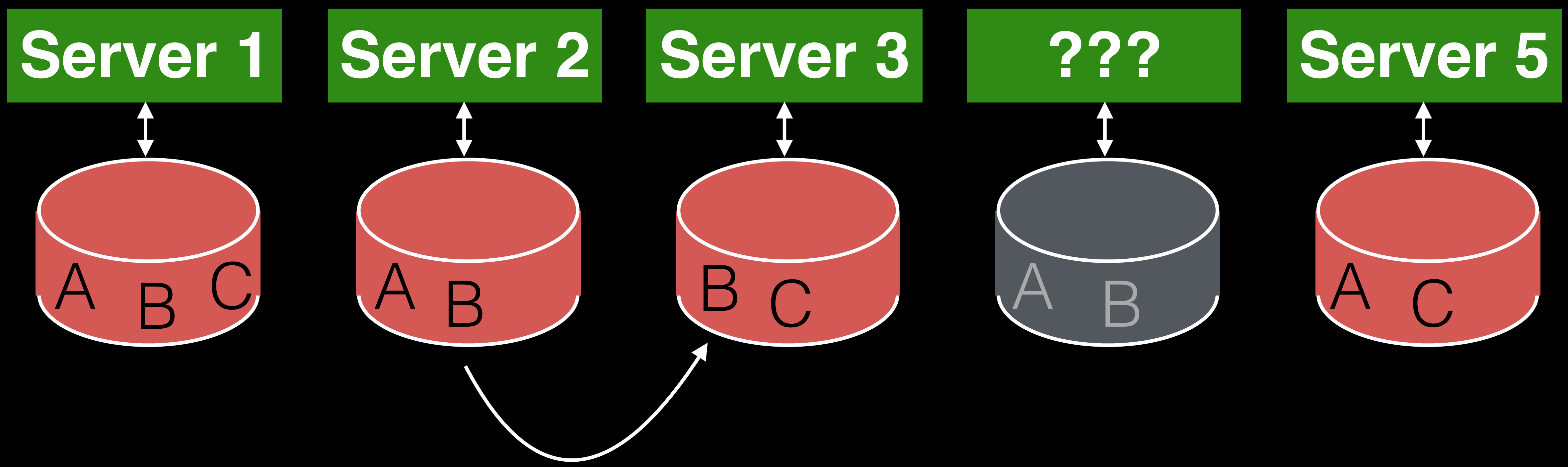
Recovery



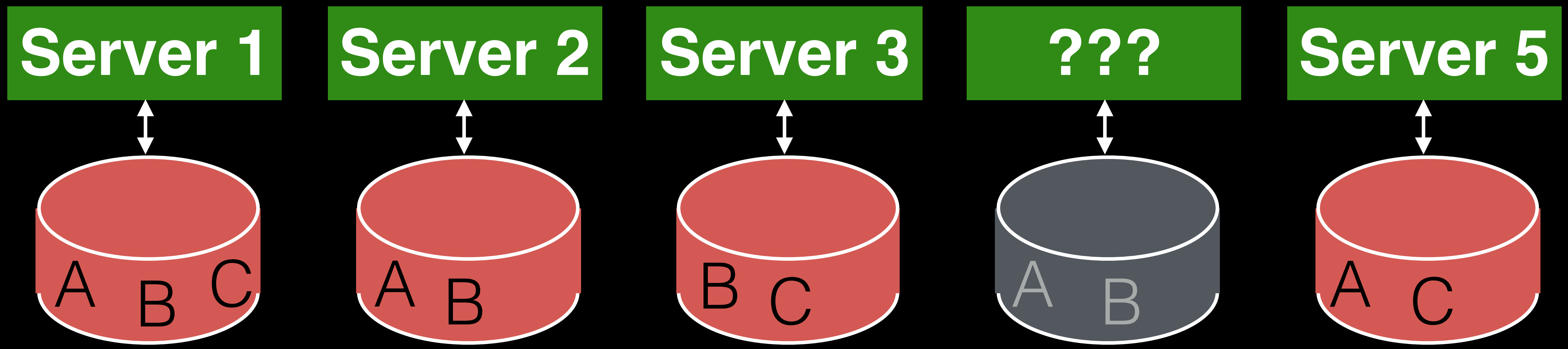
Recovery



Recovery

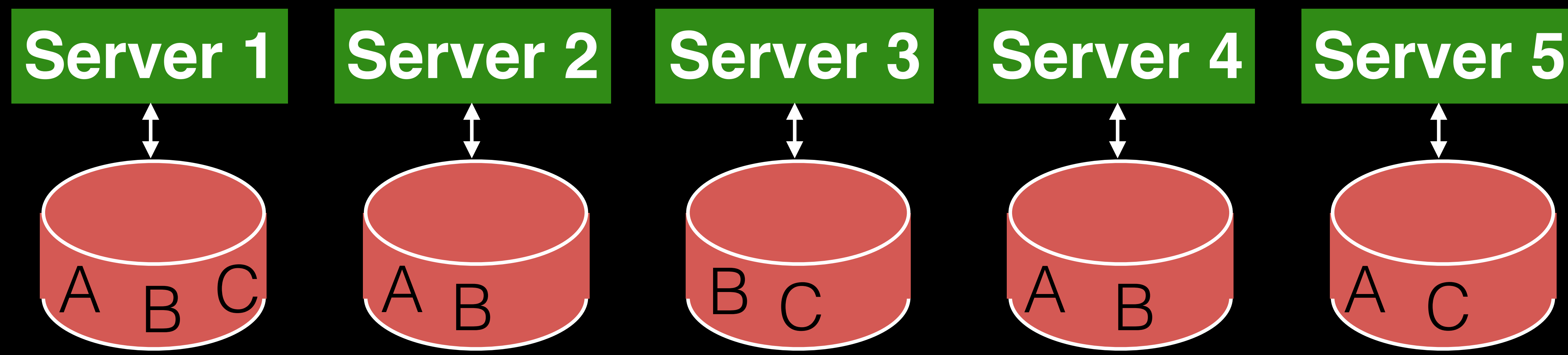


Recovery

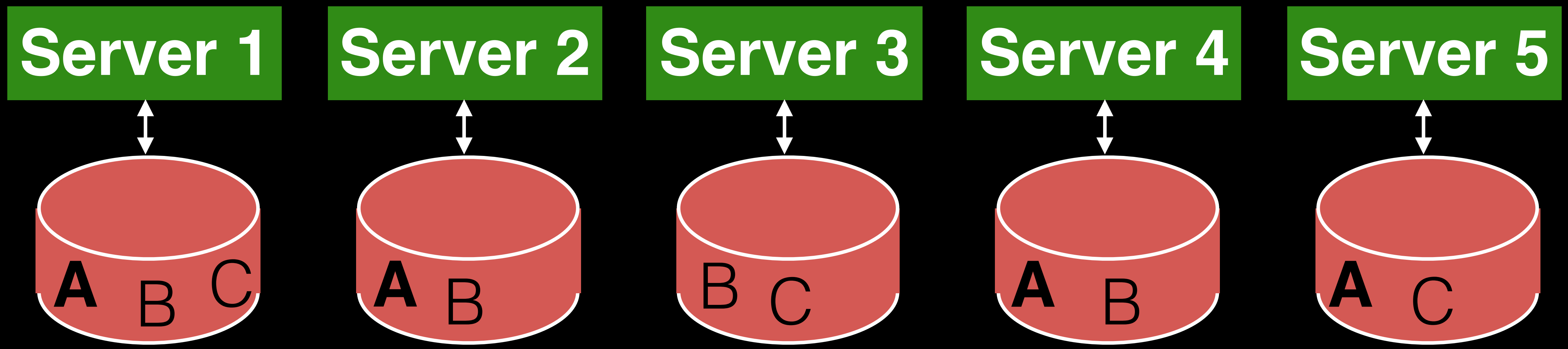


Machine may come back, or it may be dead forever.

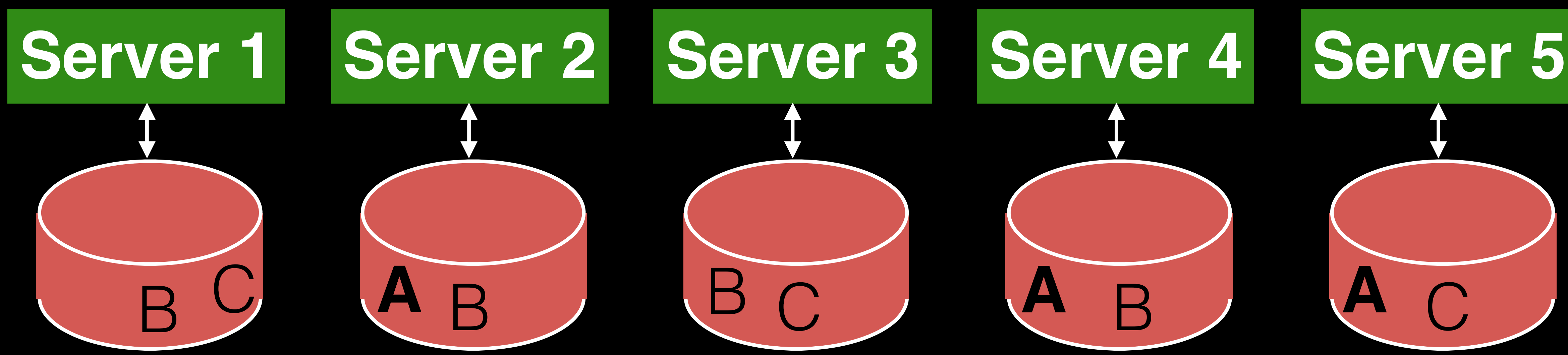
Recovery



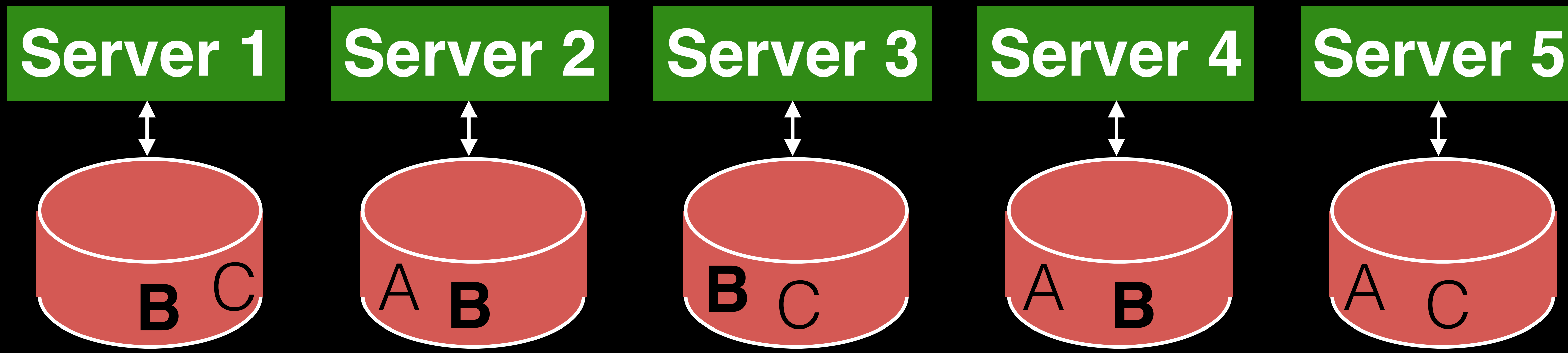
Recovery



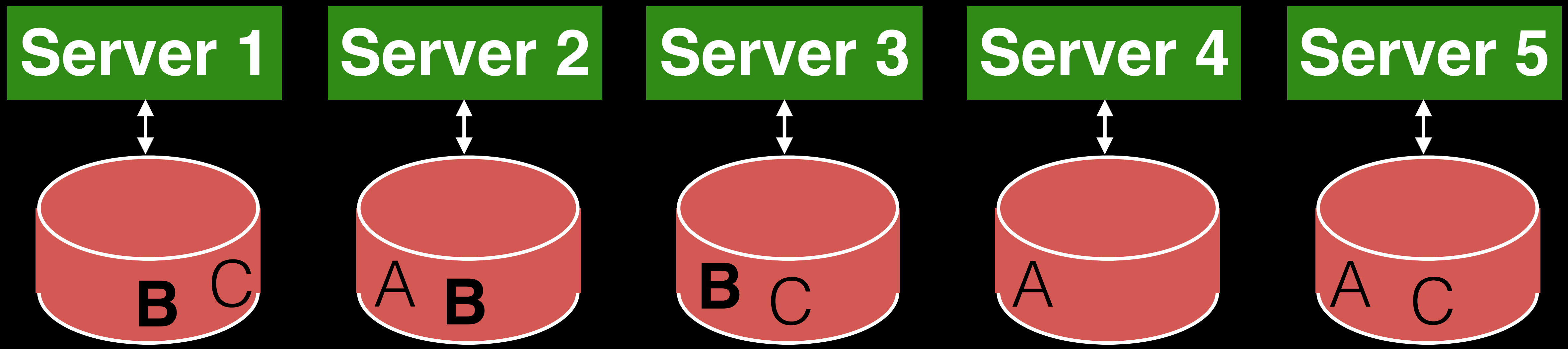
Recovery



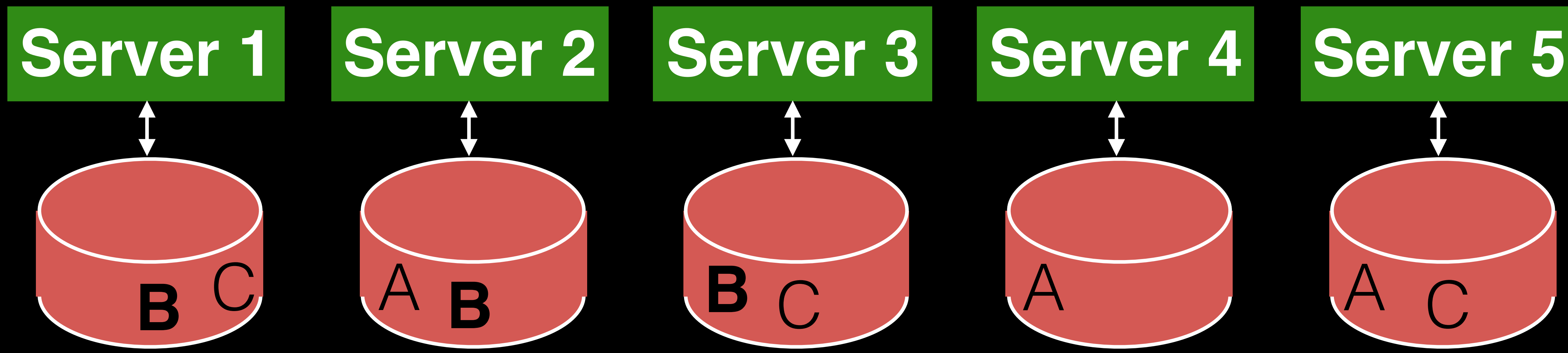
Recovery



Recovery

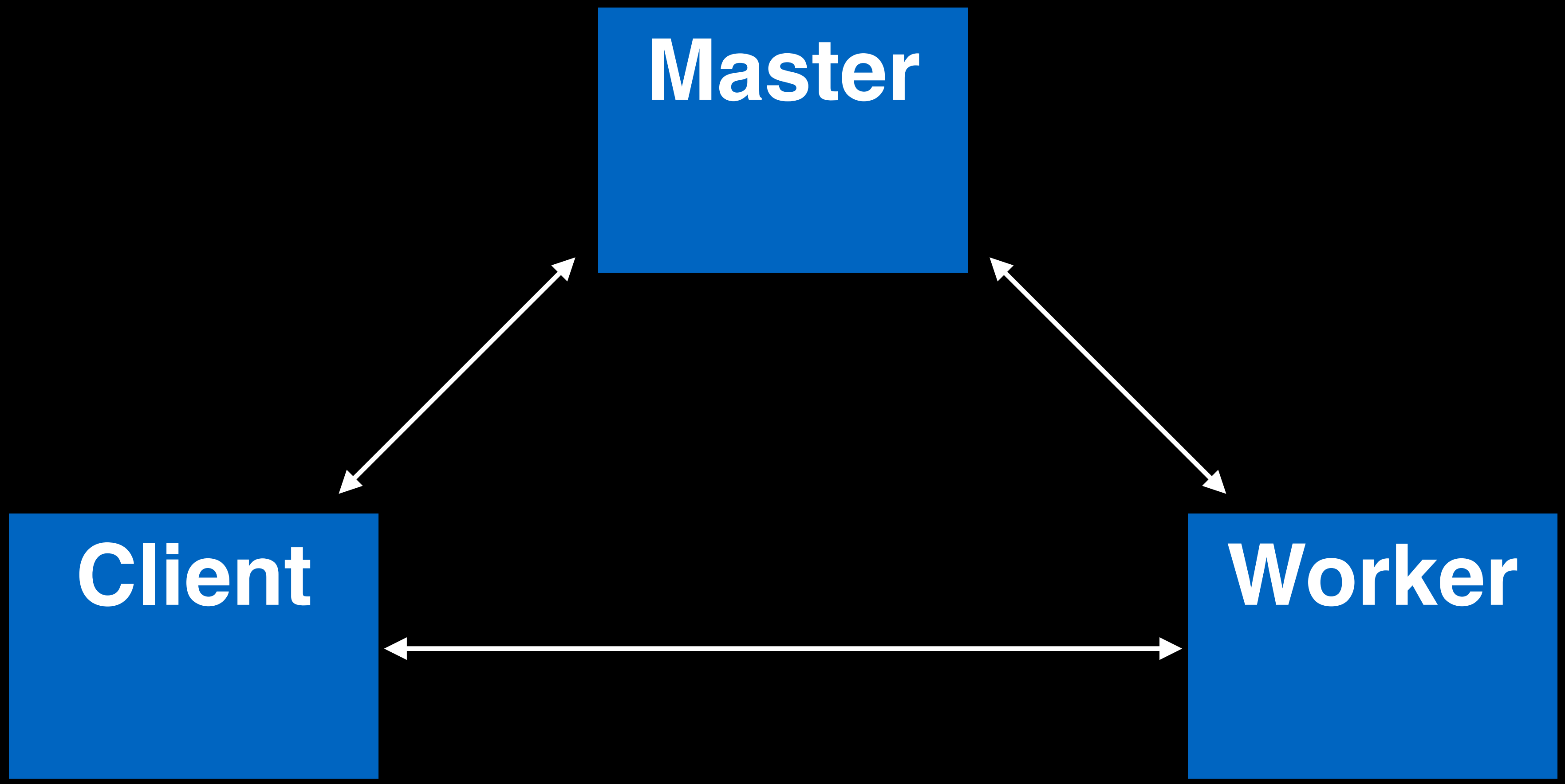


Observation

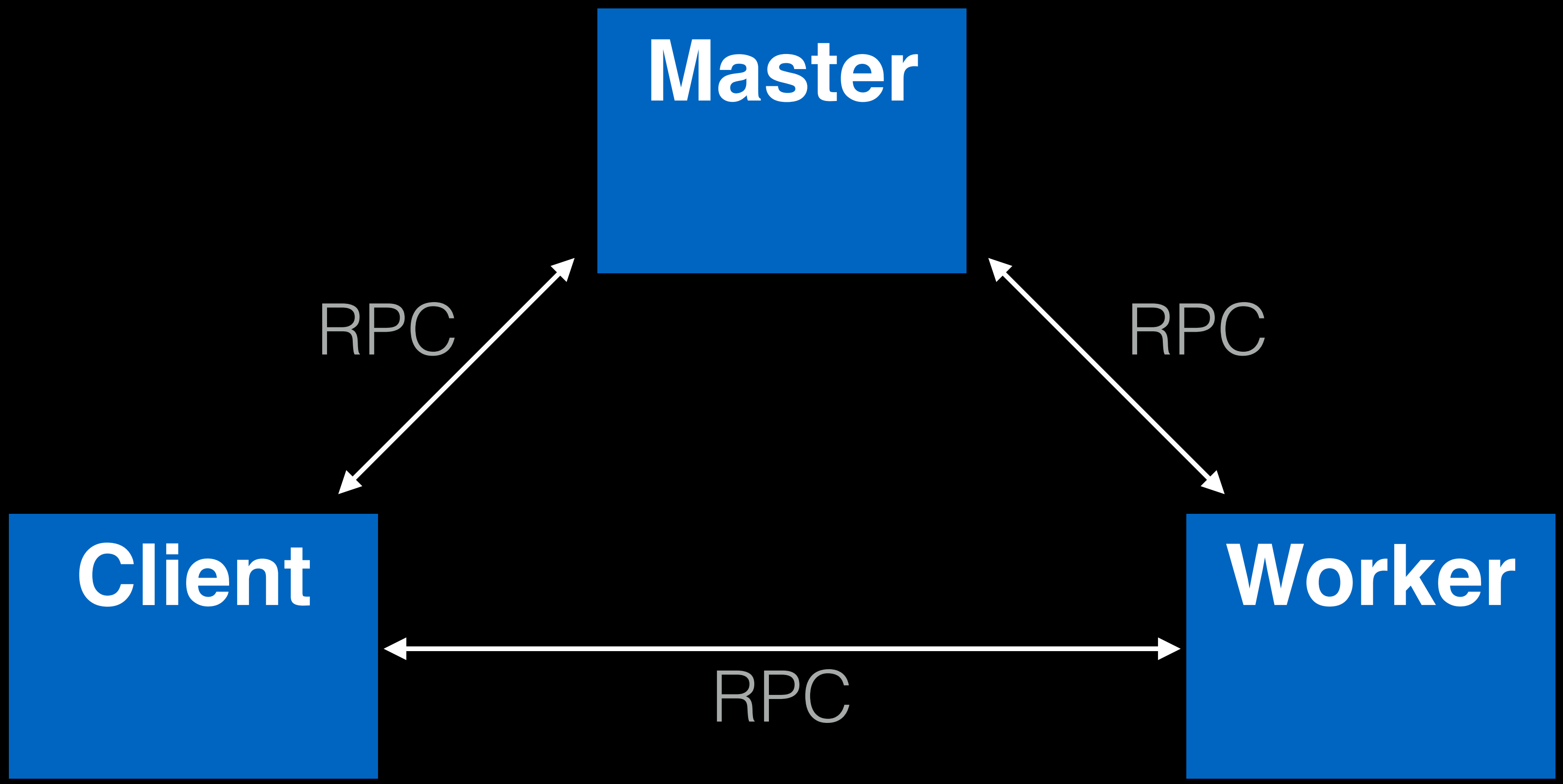


Maintaining replication and finding data will be difficult unless we have a global view of the data.

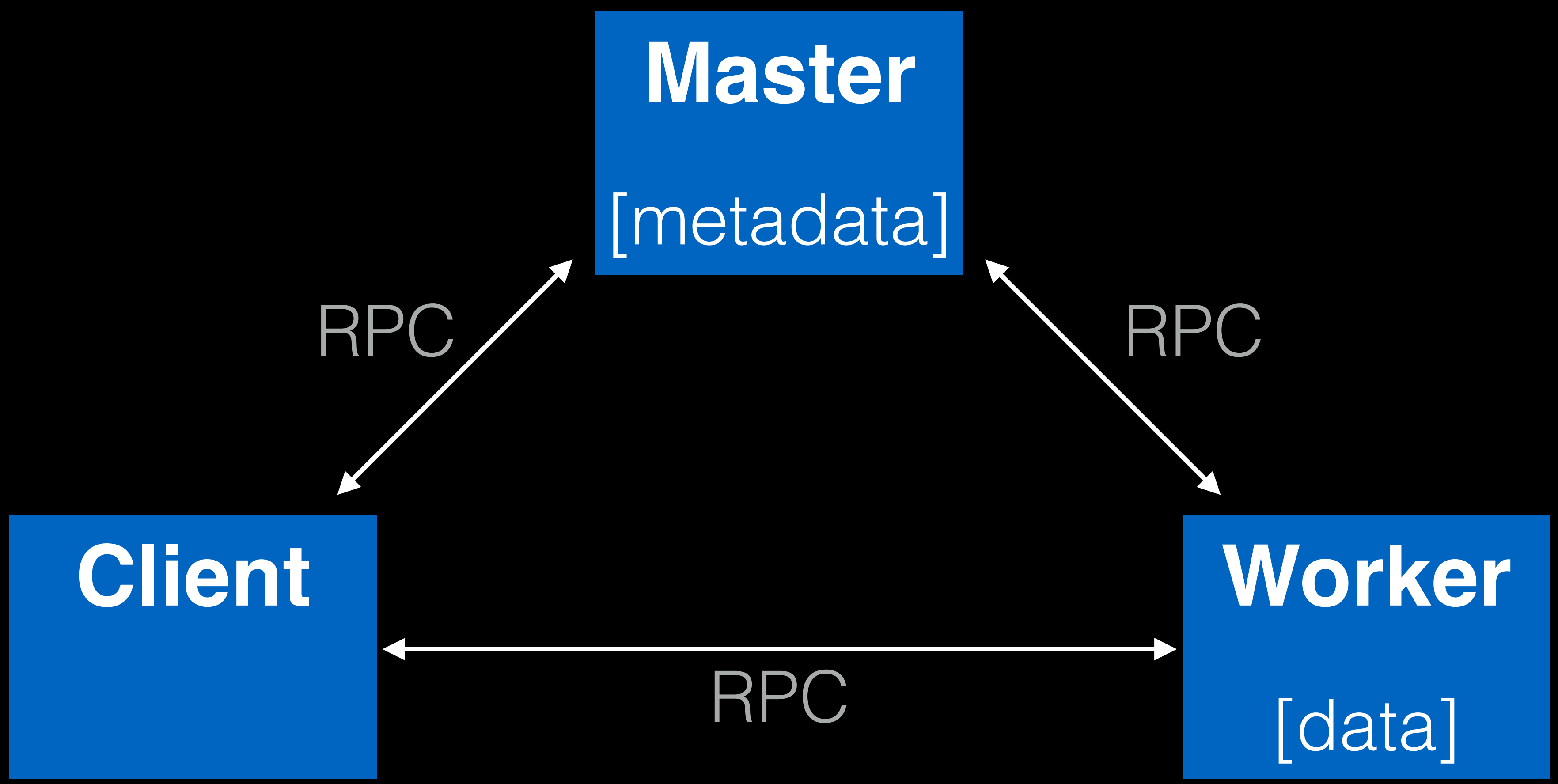
Architecture



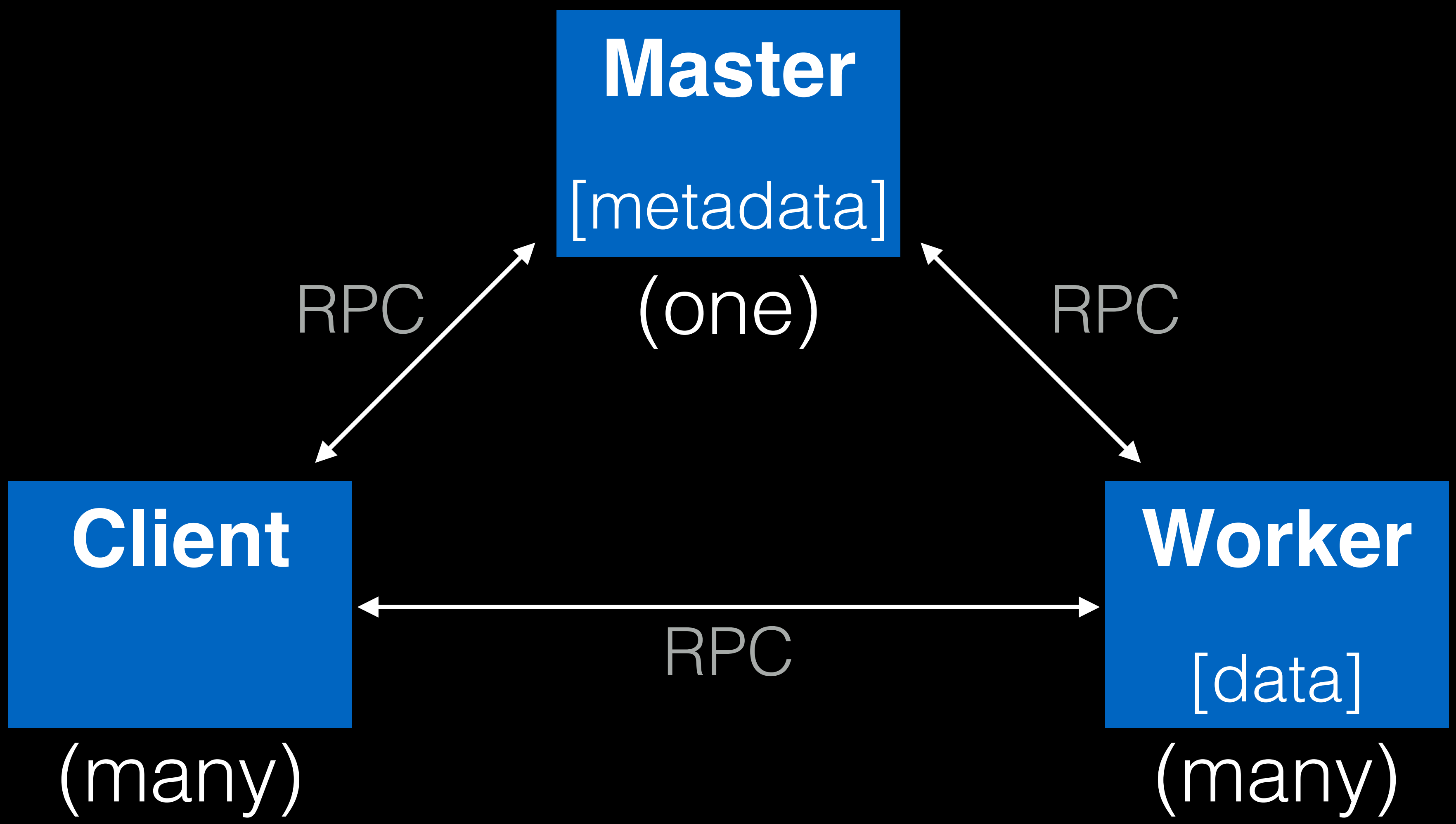
Architecture



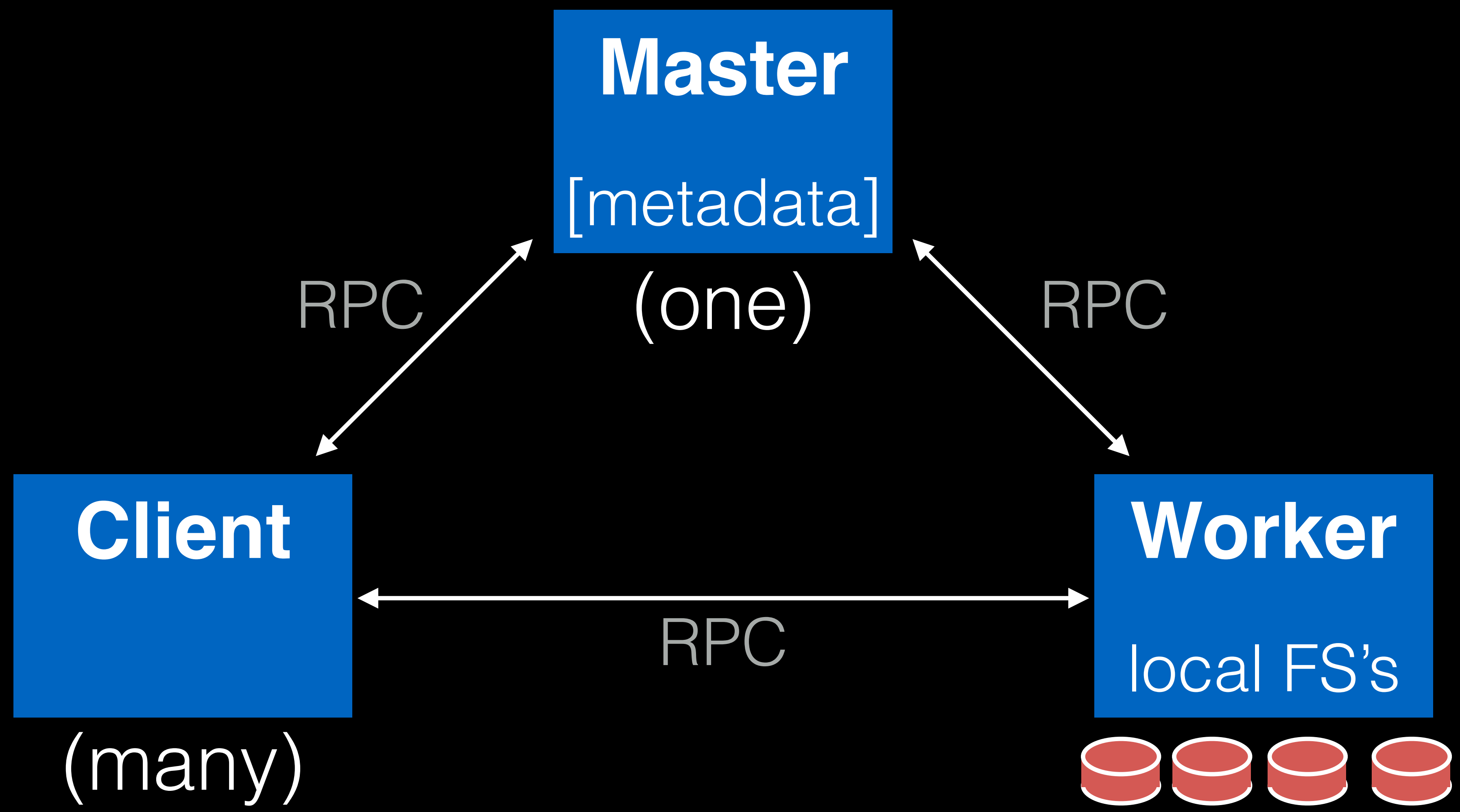
Architecture



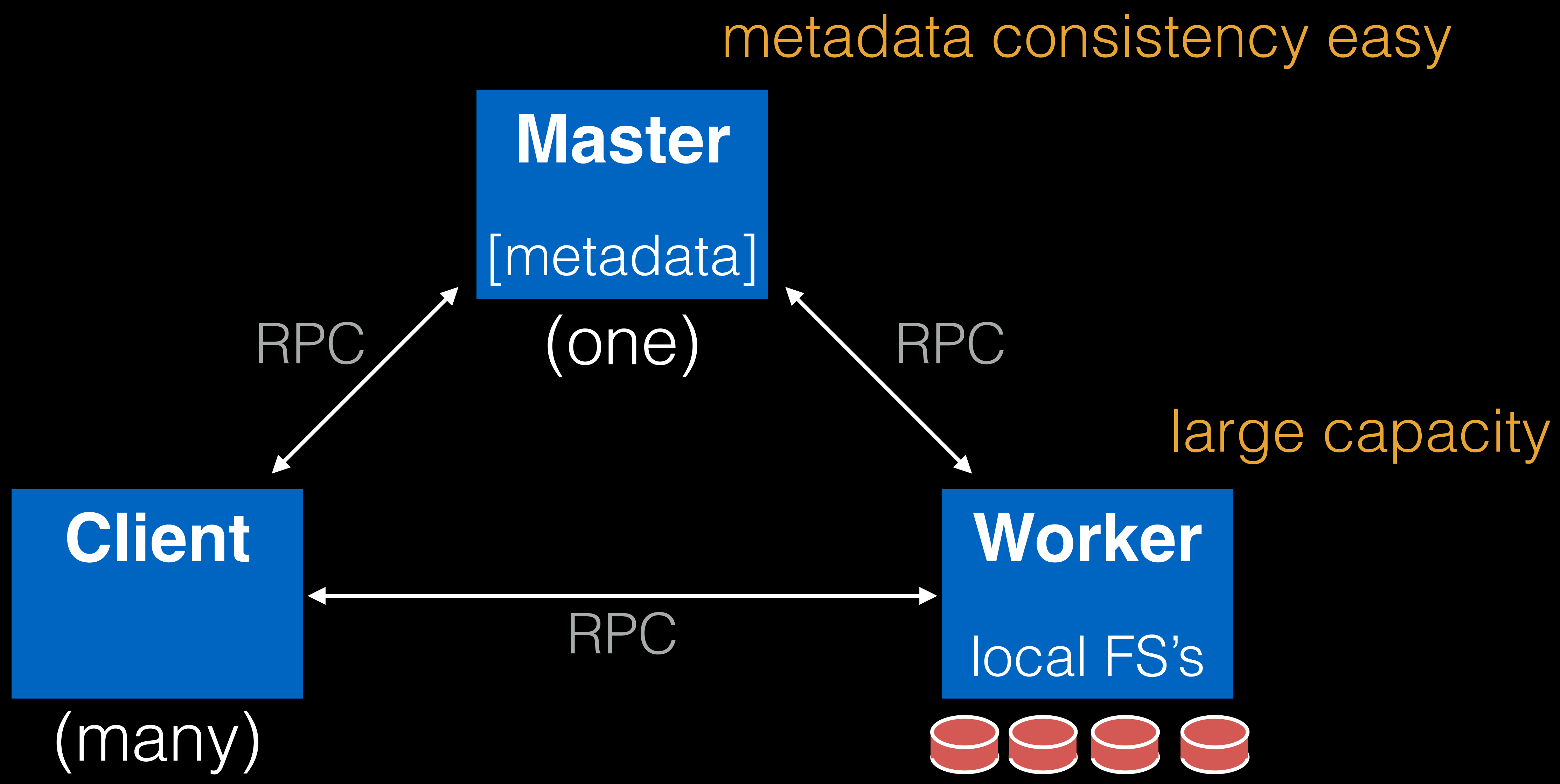
Architecture



Architecture



Architecture



Chunk Layer

Break GFS files into large chunks (e.g., 64MB).

Workers store physical chunks in Linux files.

Master maps logical chunk to physical chunk locations.

GFS Overview

Motivation

Architecture

Master metadata

Worker data

Chunk Map

Master

chunk map:

logical	phys
924	w2,w5,w7
521	w2,w9,w11
...	...

Worker w2

Master

chunk map:

logical	phys
924	w2,w5,w7
521	w2,w9,w11
…	…

Worker w2

Local FS

/chunks/942 => data1
/churks/521 => data2
…

Client Reads a Chunk

Master

chunk map:

logical	phys
924	w2,w5,w7
521	w2,w9,w11
…	…

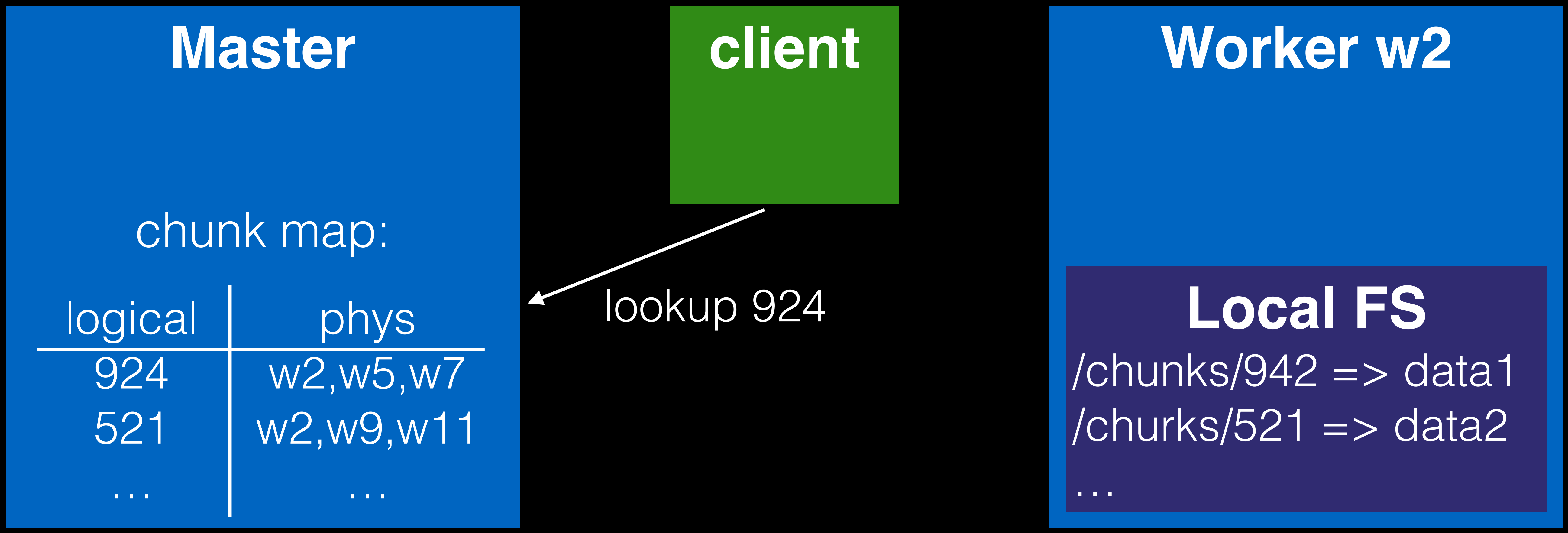
client

Worker w2

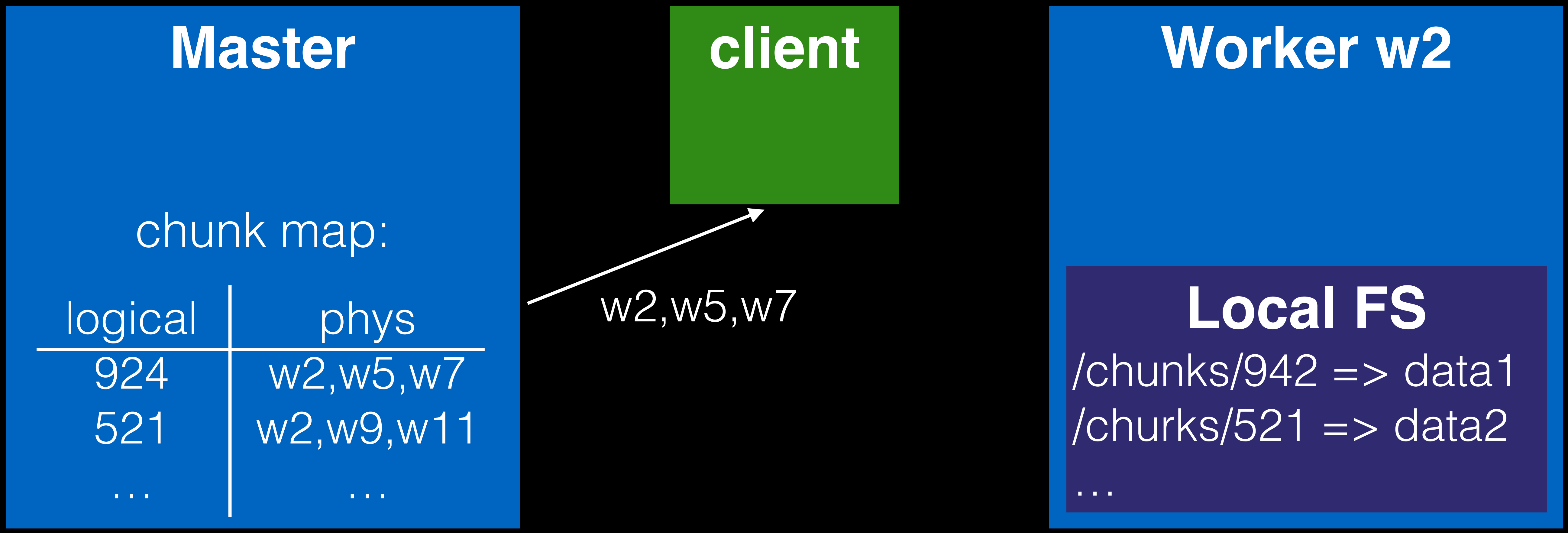
Local FS

/chunks/942 => data1
/churks/521 => data2
…

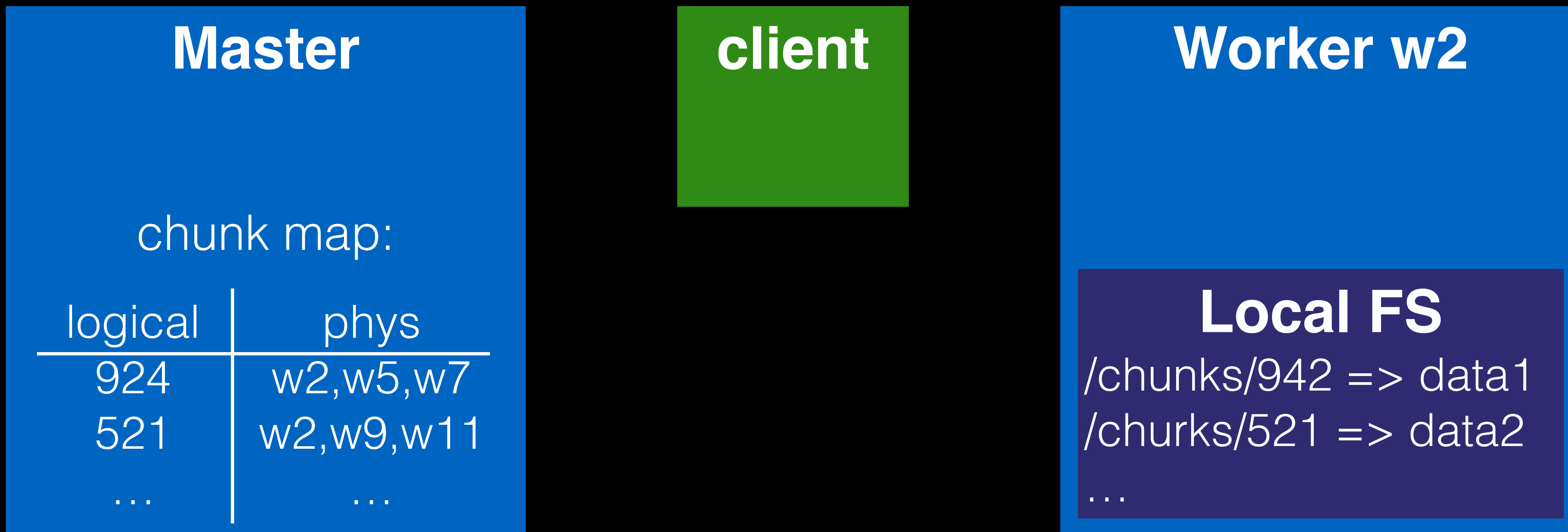
Client Reads a Chunk



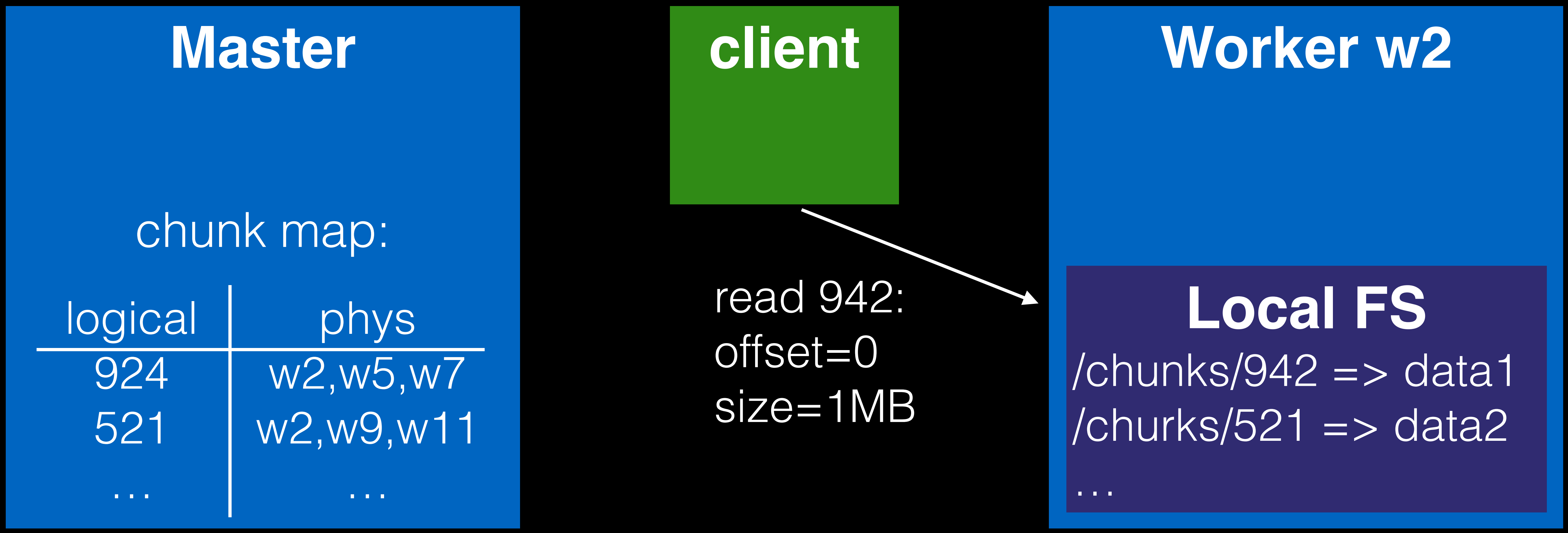
Client Reads a Chunk



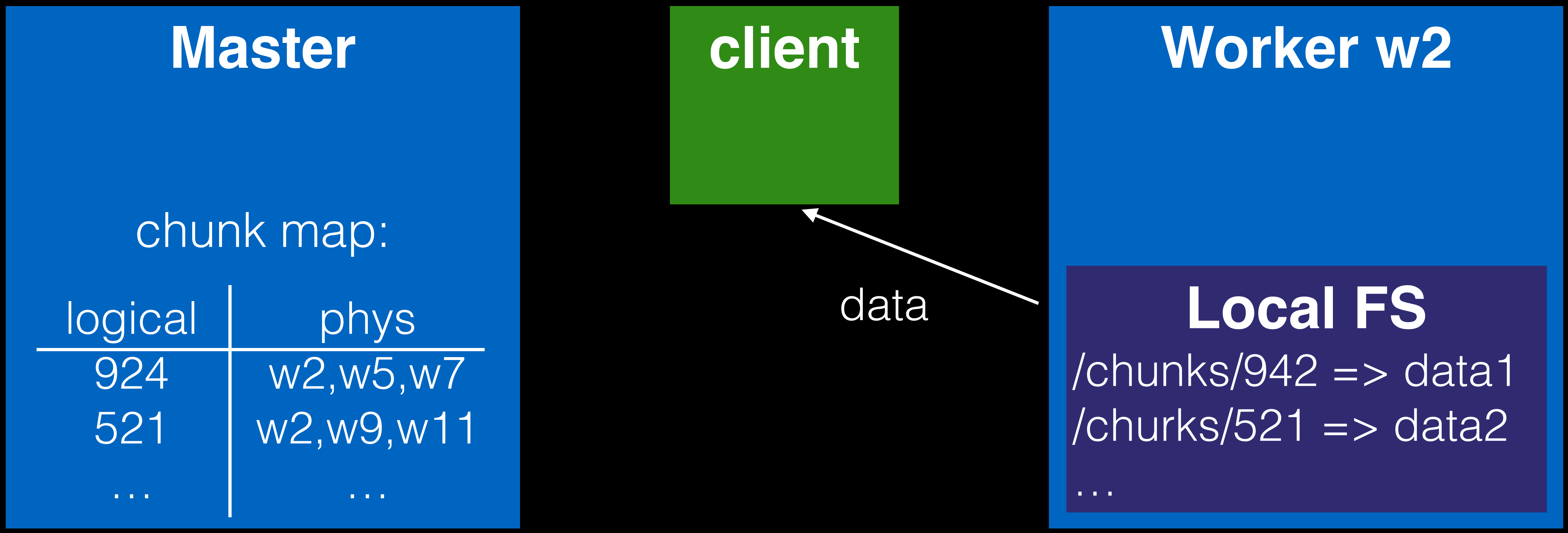
Client Reads a Chunk



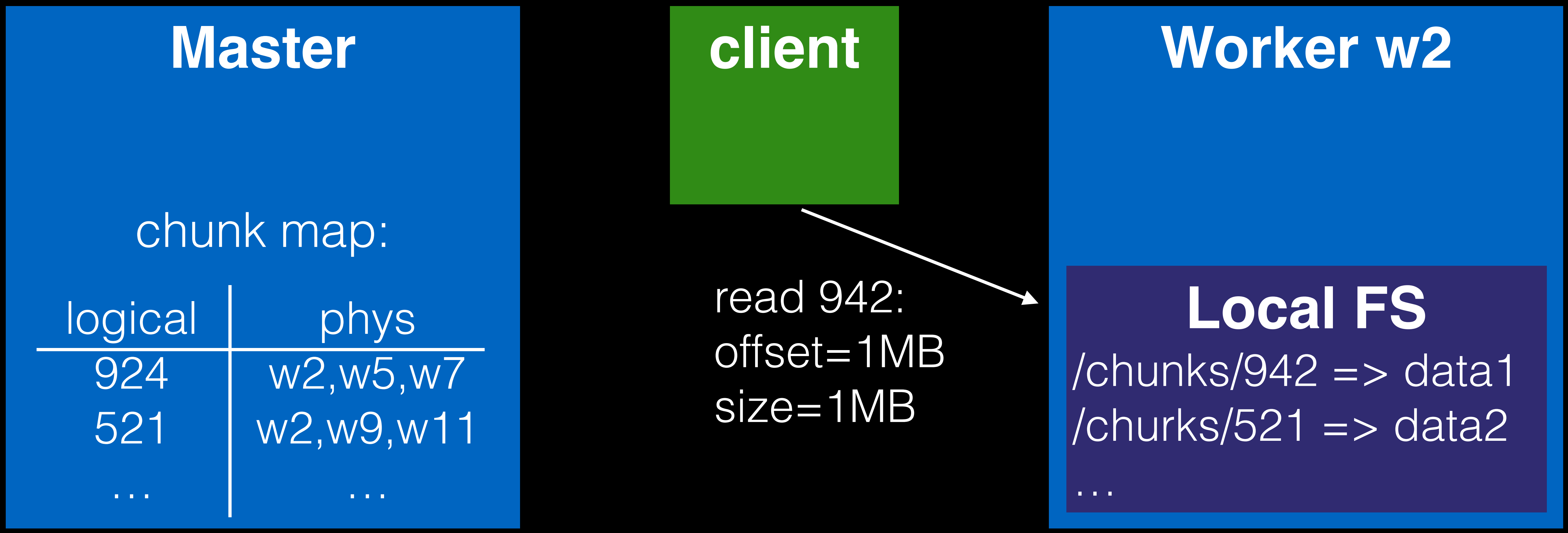
Client Reads a Chunk



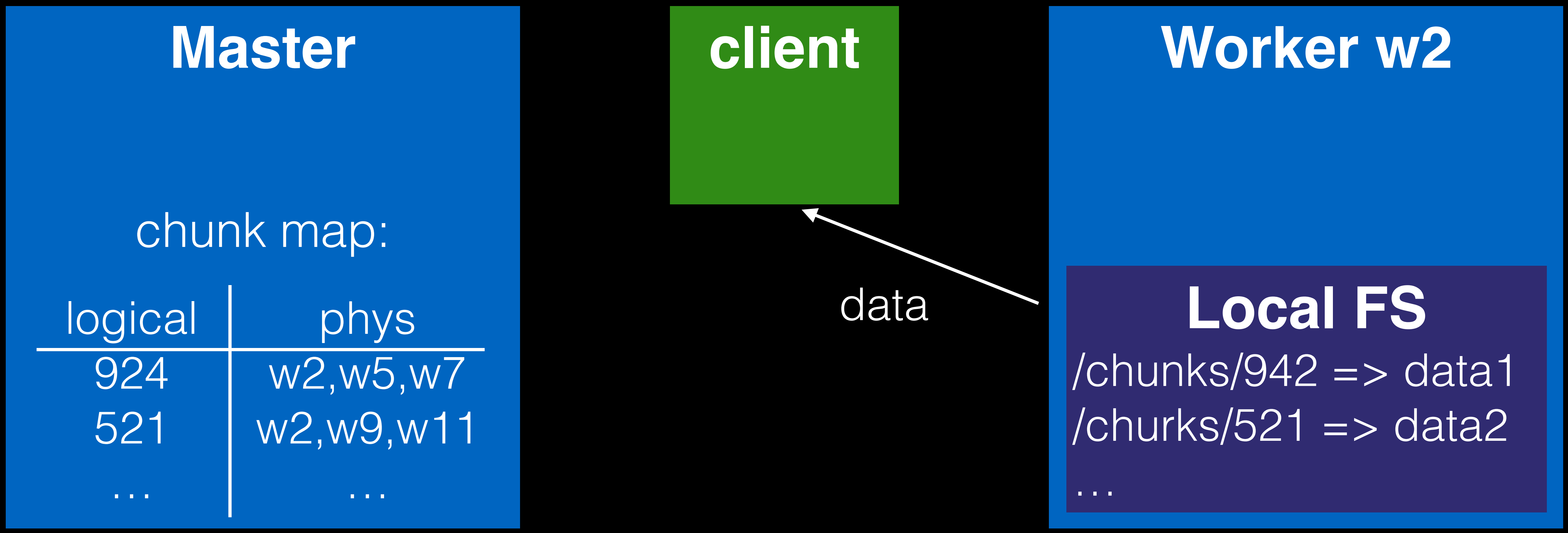
Client Reads a Chunk



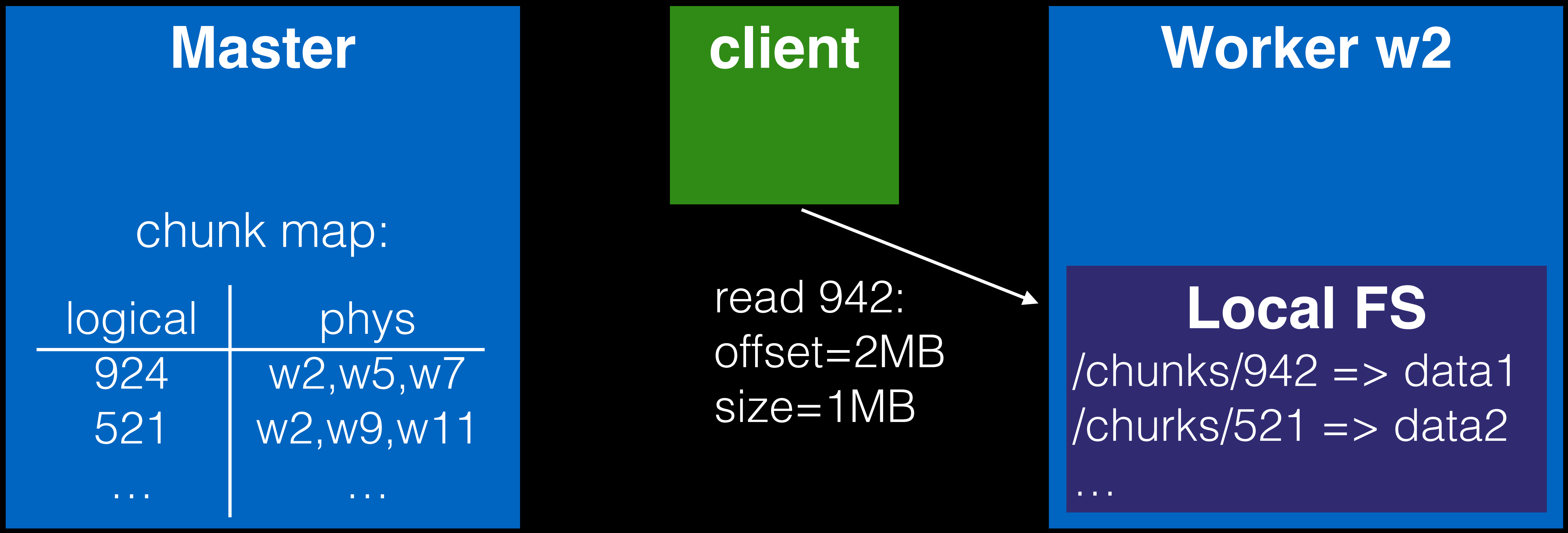
Client Reads a Chunk



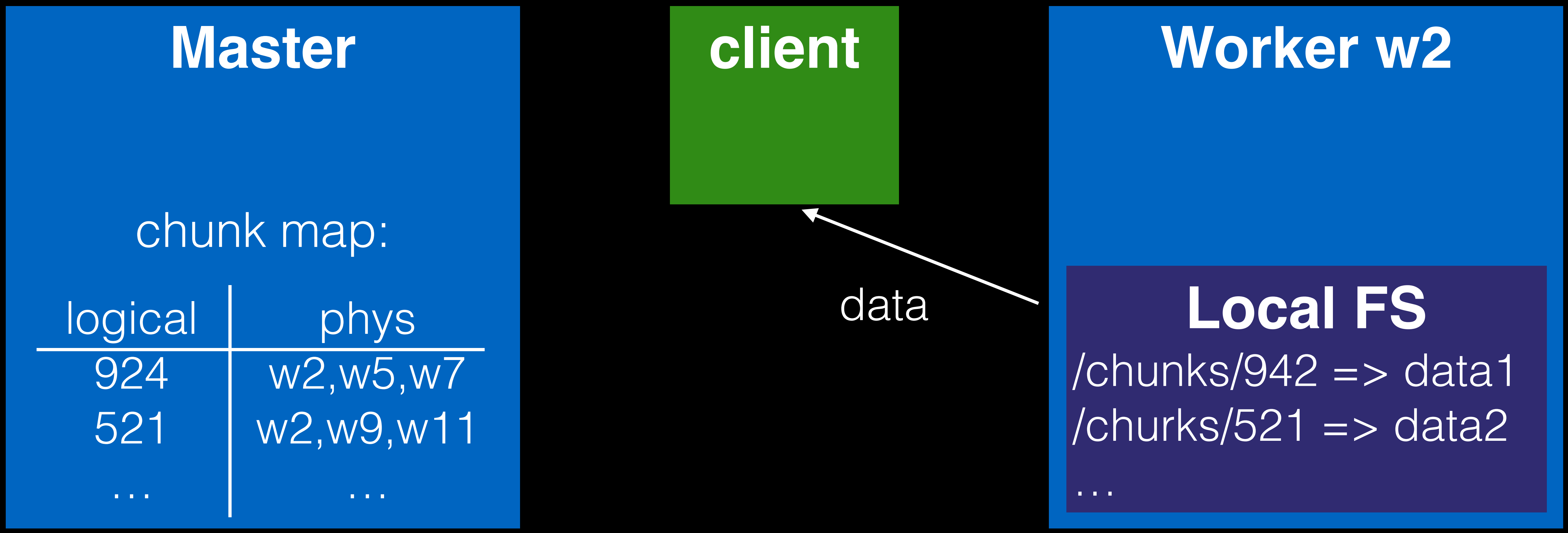
Client Reads a Chunk



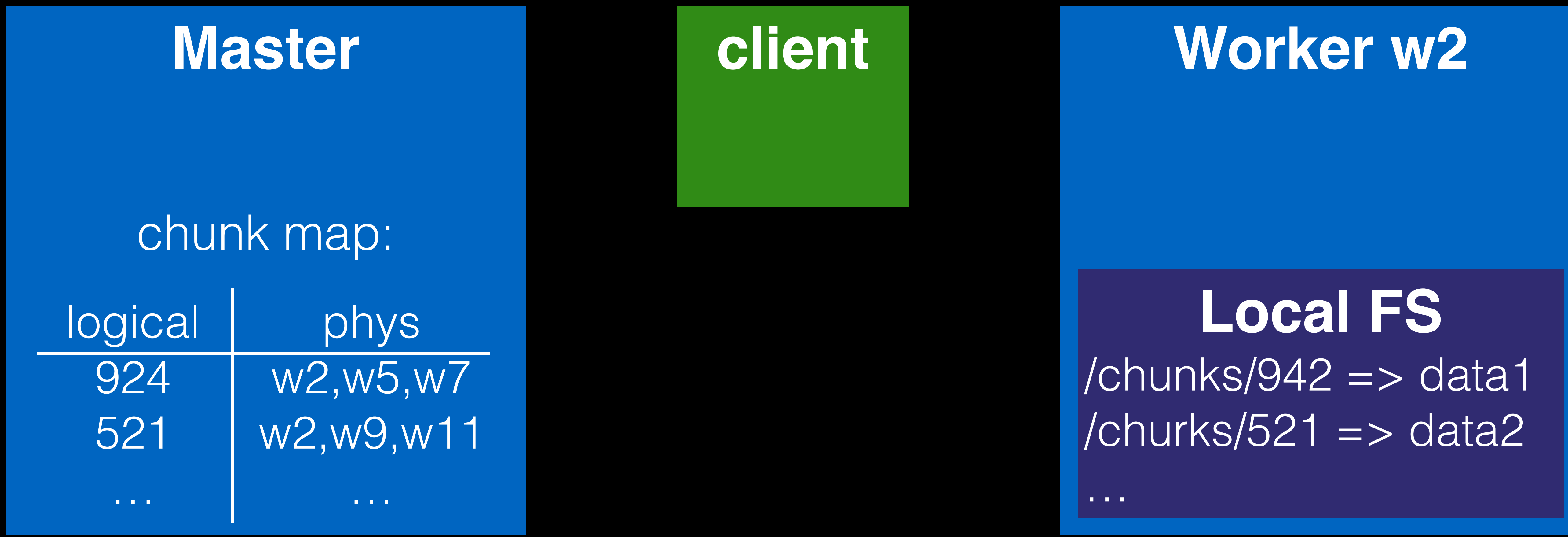
Client Reads a Chunk



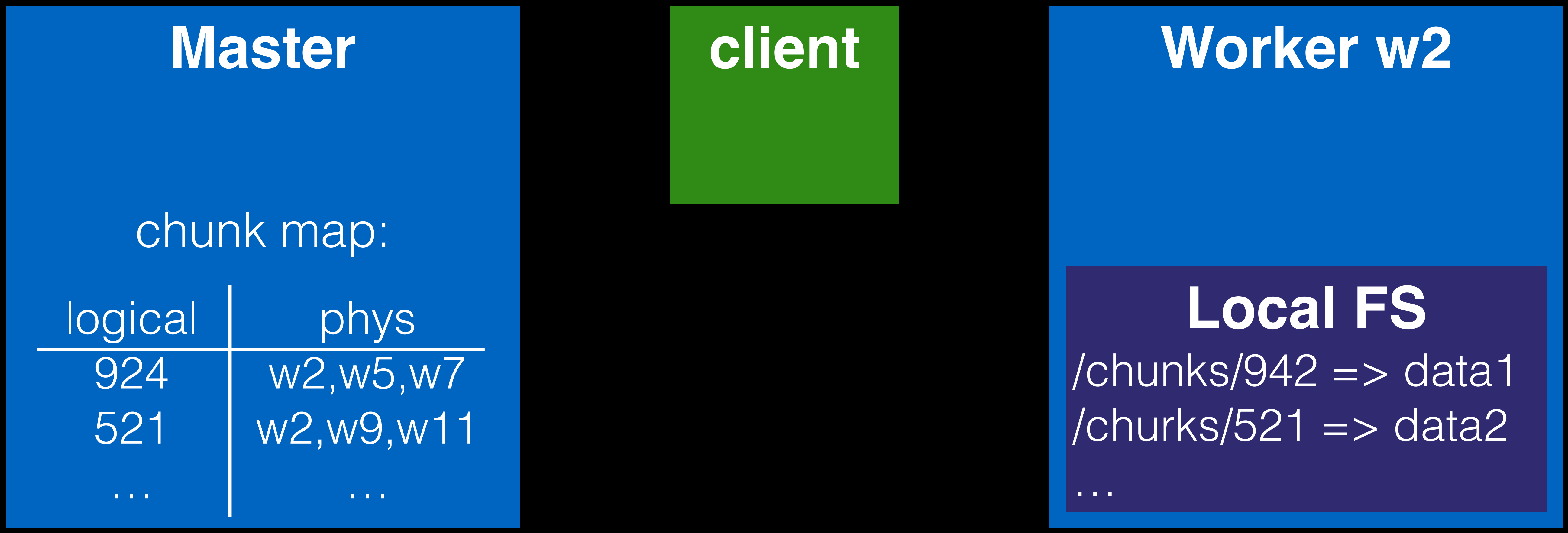
Client Reads a Chunk



Client Reads a Chunk



Client Reads a Chunk



Master is not bottleneck because not involved in most reads.

Client Reads a Chunk

Master

chunk map:

logical	phys
924	w2,w5,w7
521	w2,w9,w11
…	…

client

Worker w2

Local FS

/chunks/942 => data1
/churks/521 => data2
…

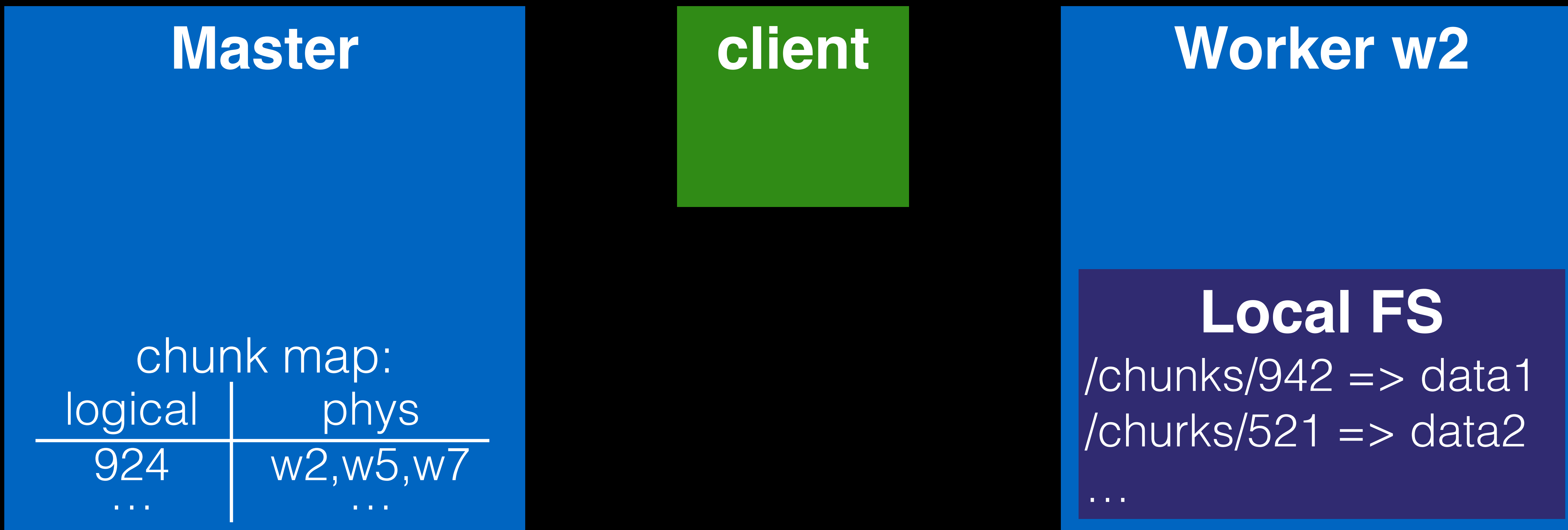
How does client know what chunk to read?

File Namespace

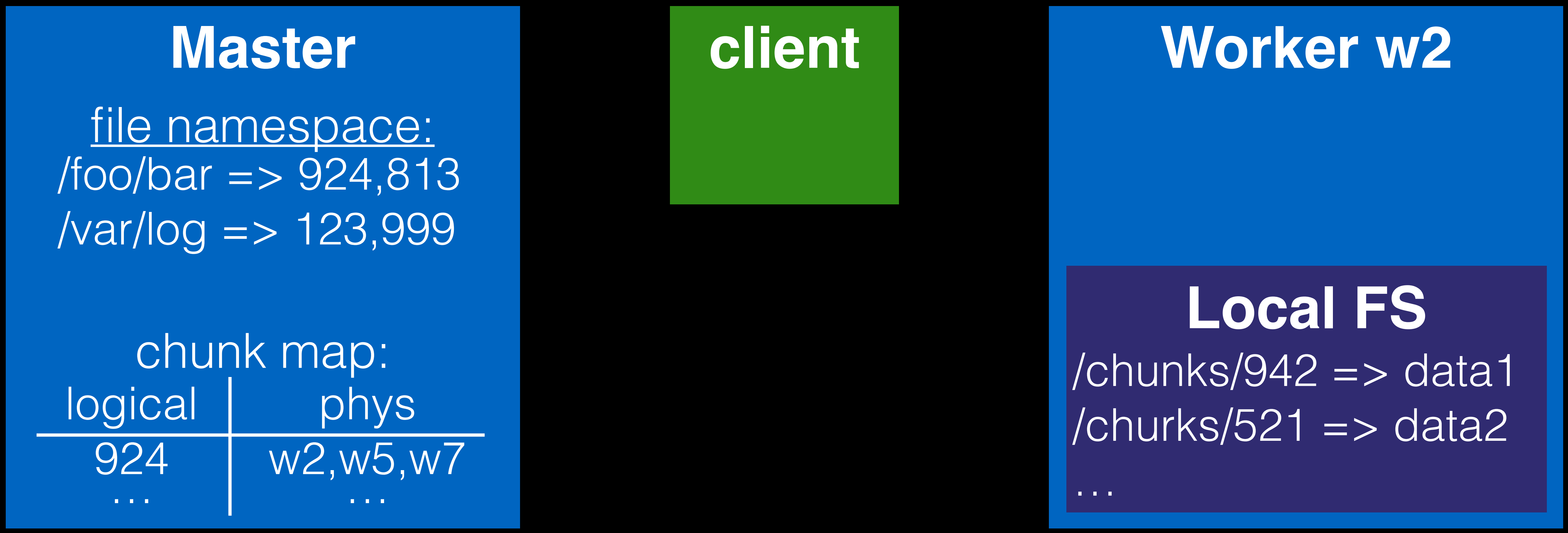
Map path names to logical chunk lists.

1. Client sends path name to master.
2. Master sends chunk locations to client.
3. Client reads/writes to workers directly.

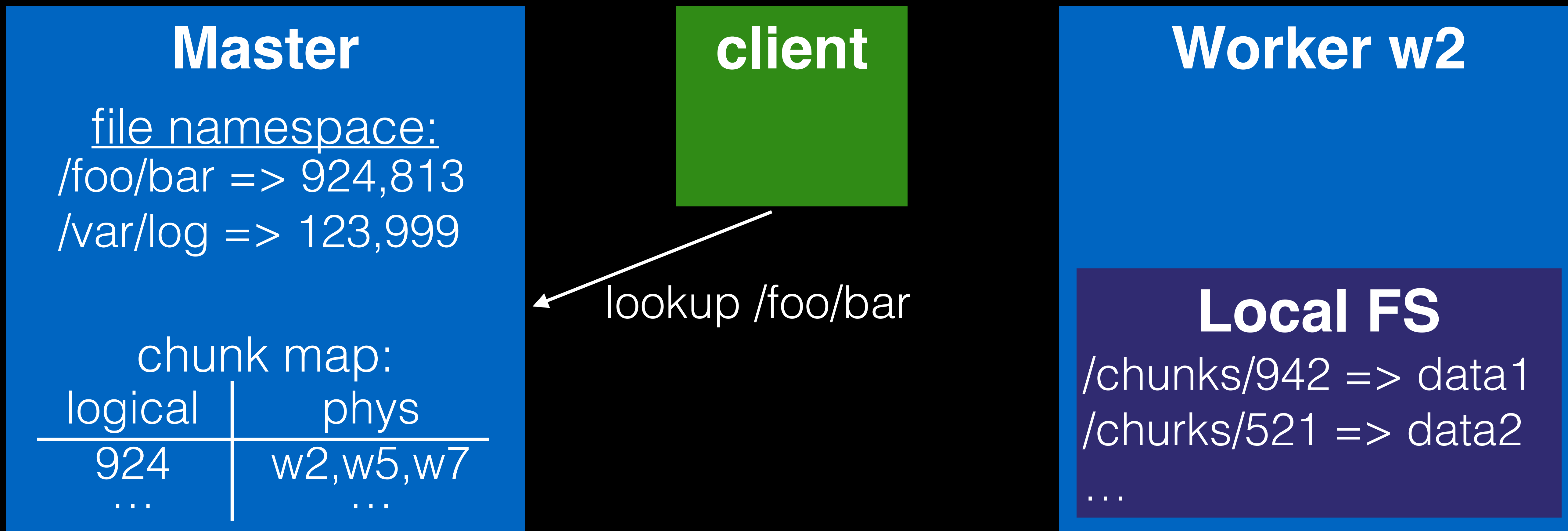
File Namespace



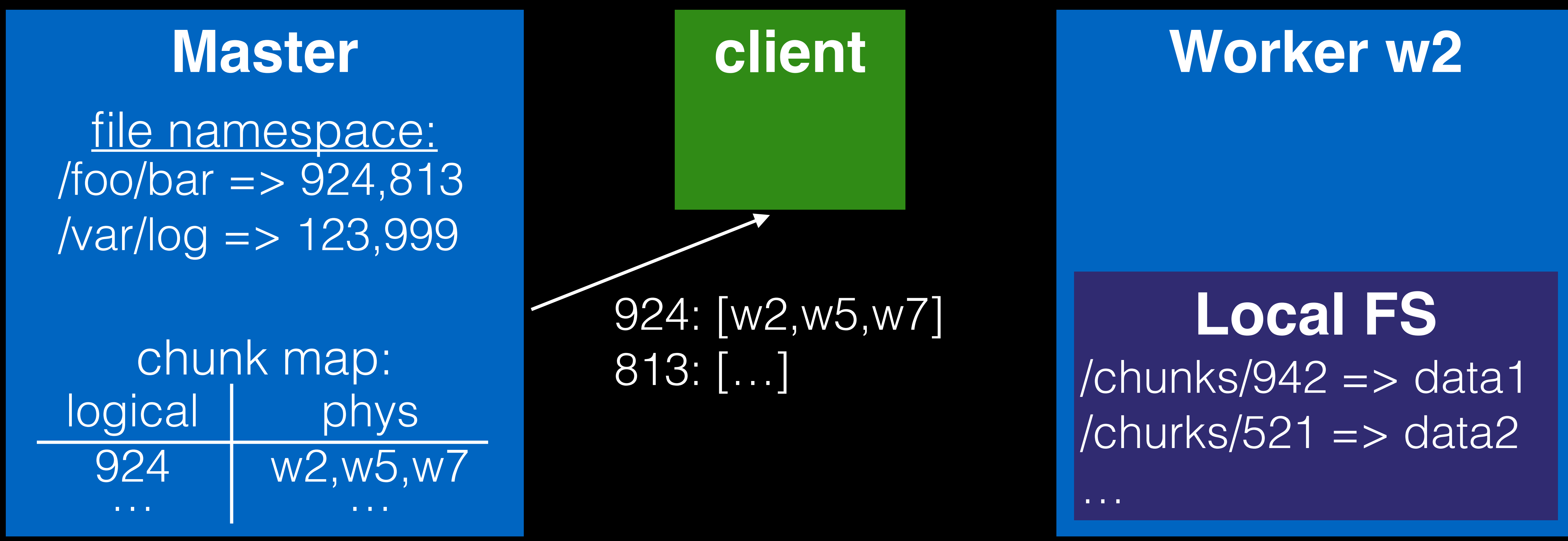
File Namespace



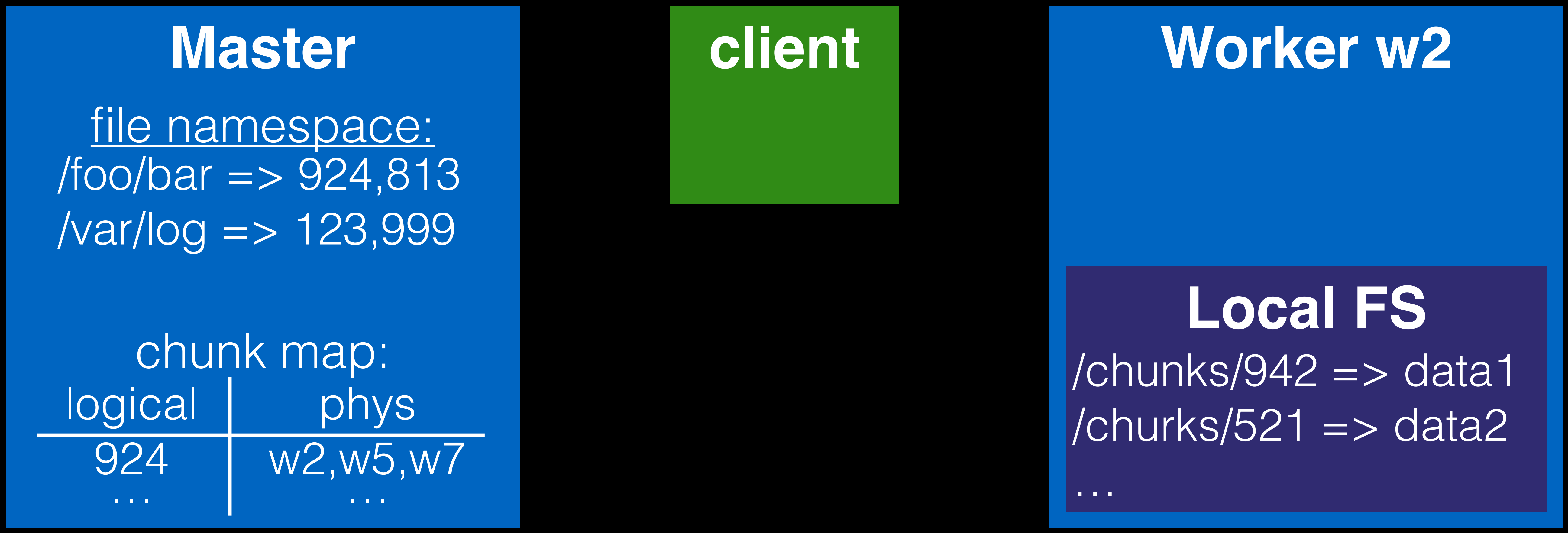
File Namespace



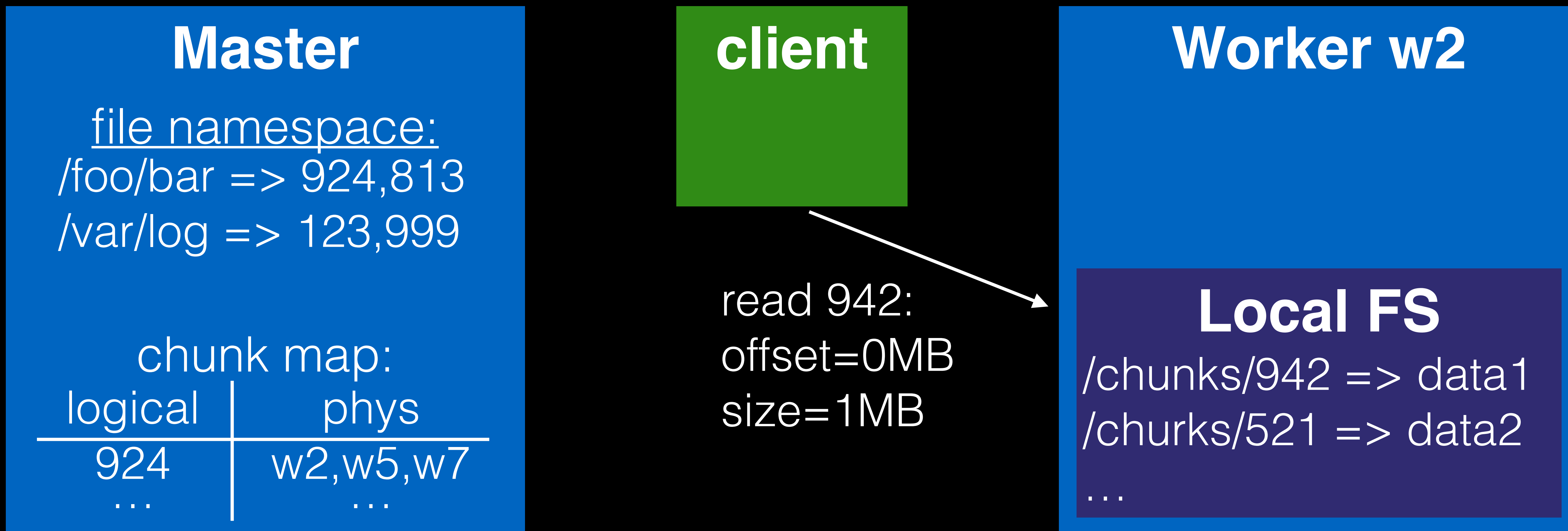
File Namespace



File Namespace



File Namespace



Chunk Size

GFS uses very large chunks, i.e., 64MB.

How does chunk size affect size of data structs?

What if Chunk Size Doubles?

Master

file namespace:
/foo/bar => 924,813
/var/log => 123,999

chunk map:

logical	phys
924	w2,w5,w7
813	w1,w8,w9
...	...

What if Chunk Size Doubles?

Master

file namespace:
/foo/bar => 924
/var/log => 123

chunk map:

logical	phys
924	w2,w5,w7
813	w1,w8,w9
...	...

lists half as long

What if Chunk Size Doubles?

Chunk Size

GFS uses very large chunks, i.e., 64MB.

How does chunk size affect size of data structs?

A: logical-block lists halved, chunk map halved

Any disadvantages to making chunks huge?

Chunk Size

GFS uses very large chunks, i.e., 64MB.

How does chunk size affect size of data structs?

A: logical-block lists halved, chunk map halved

Any disadvantages to making chunks huge?
- sometimes slow. Cannot parallelize I/O as much.

Master: Crashes + Consistency

File namespace and chunk map are 100% in RAM.
- allows master to work with 1000’s of workers
- what if master crashes?

Master: Crashes + Consistency

File namespace and chunk map are 100% in RAM.

- allows master to work with 1000’s of workers
- what if master crashes?

File Namespace

Write namespace updates to two types of logs:
- local disk (disk is never read except for crash)
- disk on backup master (in case permanent fail)

Occasionally dump entire state to checkpoint.
- use format that can be directly mapped for fast recovery (i.e., no parsing).
- why can't we use pointers?

Master: Crashes + Consistency

File namespace and chunk map are 100% in RAM.

- allows master to work with 1000's of workers
- what master crashes?

Master: Crashes + Consistency

File namespace and chunk map are 100% in RAM.

- allows master to work with 1000’s of workers
- what master crashes?

Chunk Map

Don’t persist on master. Just ask workers which chunks they have.

What if worker dies too?

Chunk Map

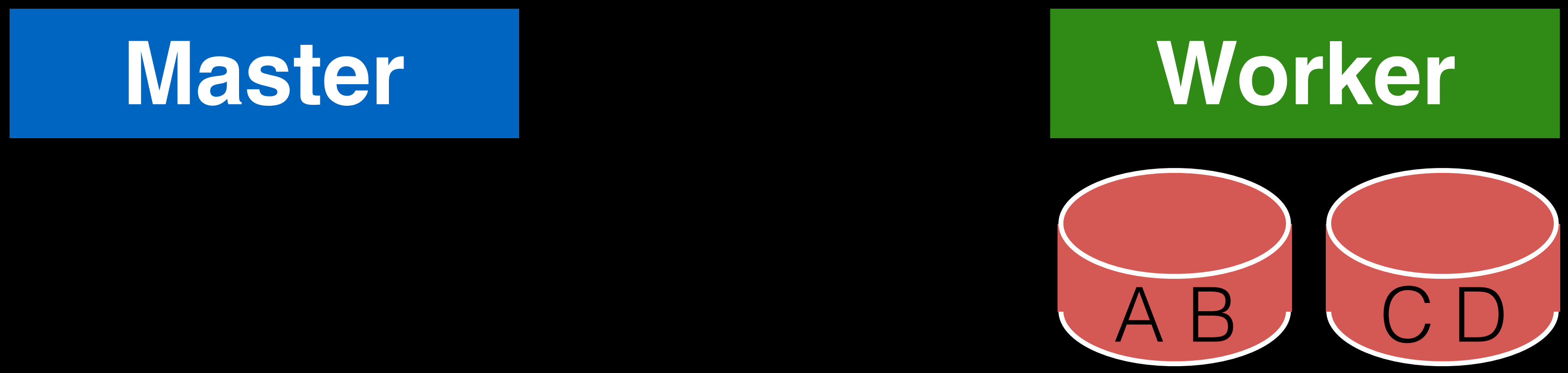
Don't persist on master. Just ask workers which chunks they have.

What if worker dies too? Doesn't matter, then that worker can serve chunks in the map anyway.

Chunk Map

Don't persist on master. Just ask workers which chunks they have.

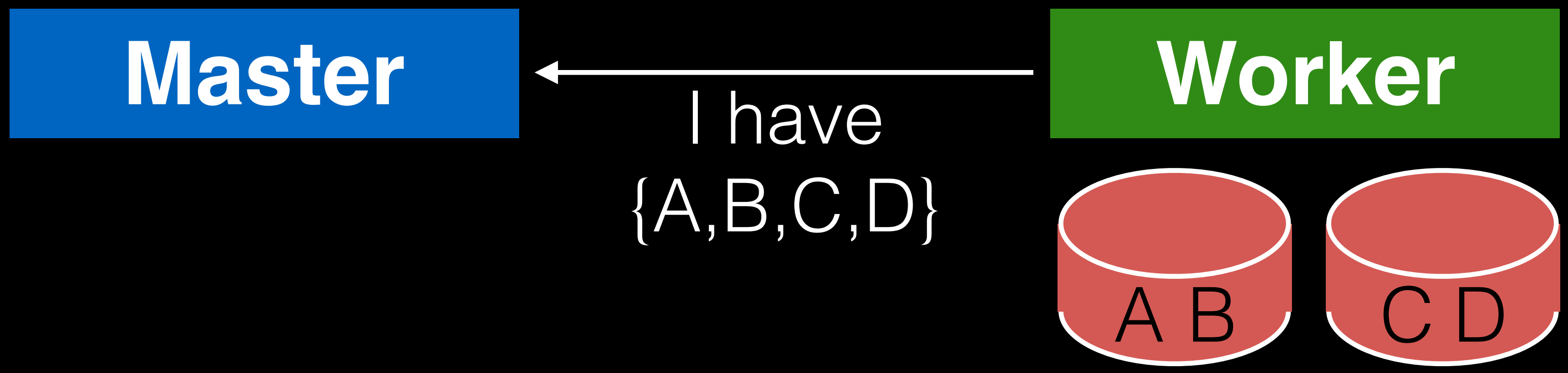
What if worker dies too? Doesn't matter, then that worker can serve chunks in the map anyway.



Chunk Map

Don't persist on master. Just ask workers which chunks they have.

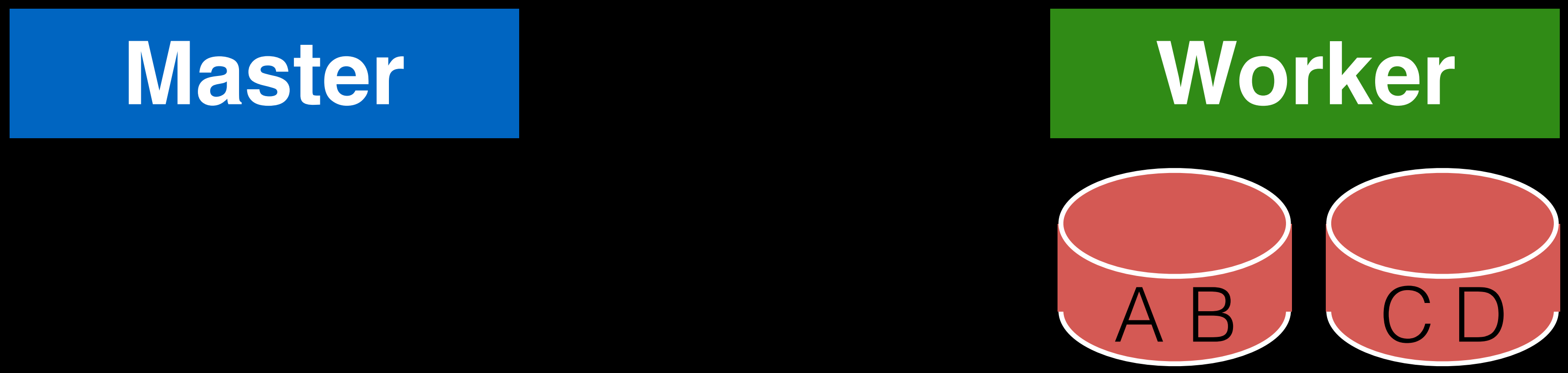
What if worker dies too? Doesn't matter, then that worker can serve chunks in the map anyway.



Chunk Map

Don’t persist on master. Just ask workers which chunks they have.

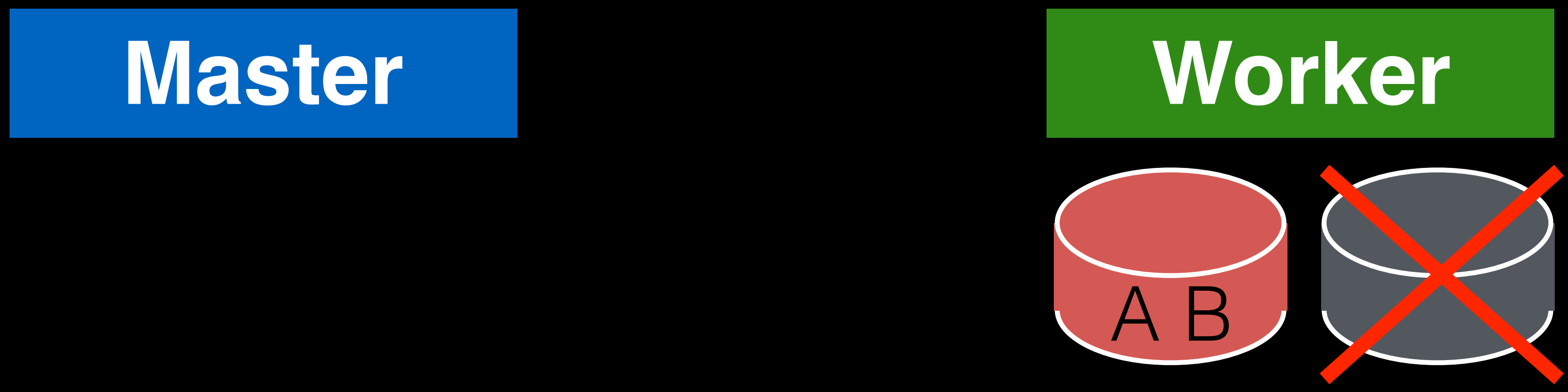
What if worker dies too? Doesn’t matter, then that worker can serve chunks in the map anyway.



Chunk Map

Don't persist on master. Just ask workers which chunks they have.

What if worker dies too? Doesn't matter, then that worker can serve chunks in the map anyway.



Chunk Map

Don't persist on master. Just ask workers which chunks they have.

What if worker dies too? Doesn't matter, then that worker can serve chunks in the map anyway.

GFS Overview

Motivation

Architecture

Master metadata

Worker data

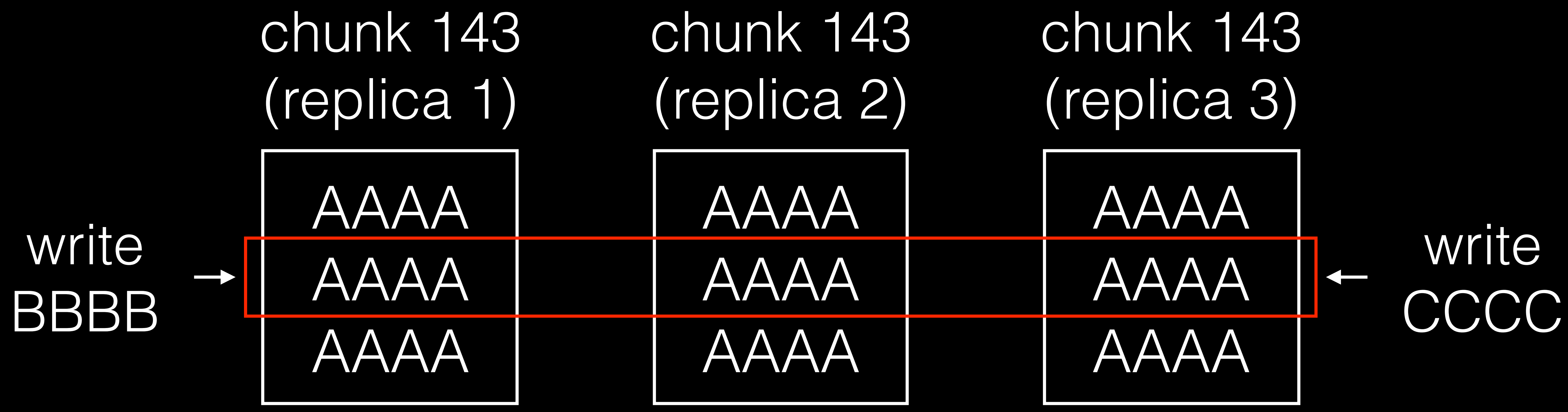
Worker Consistency

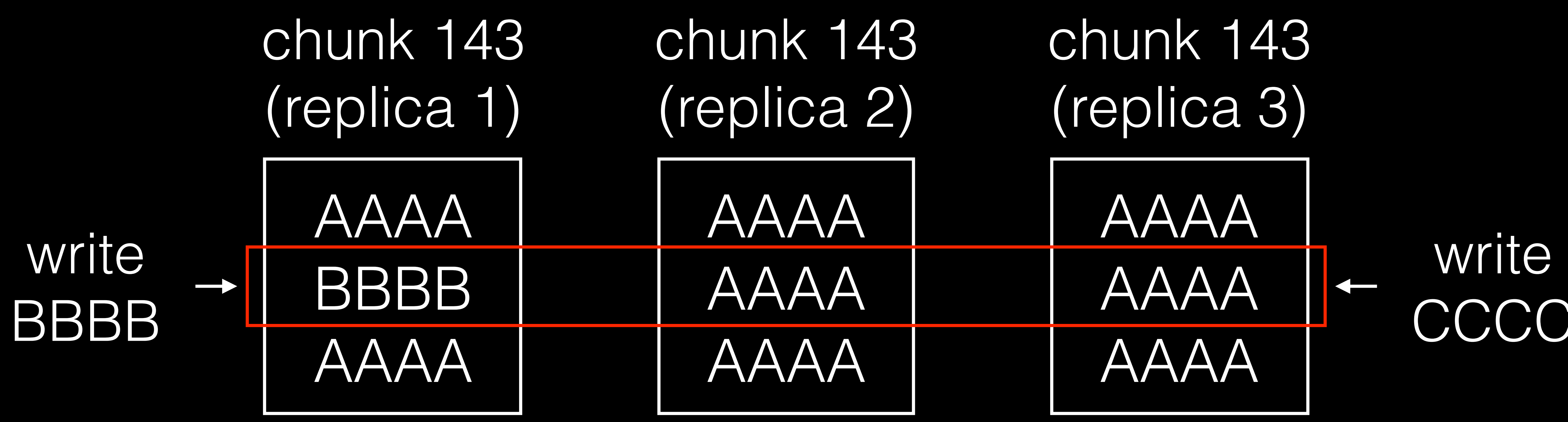
How do we make sure physical chunks are consistent with each other?

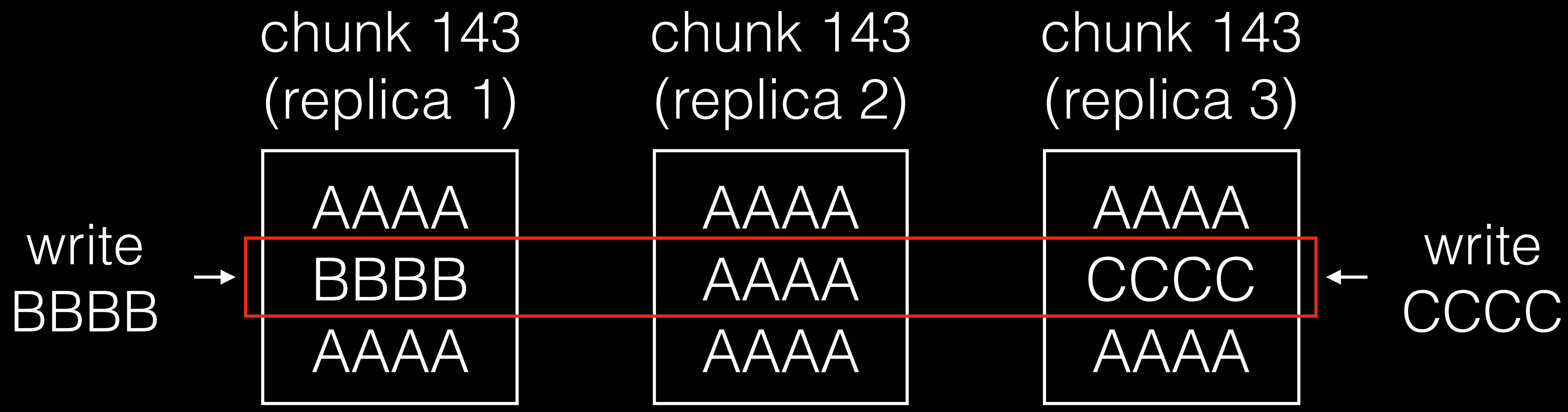
Corruption: delete chunks that violate checksum.

What about concurrent writes?

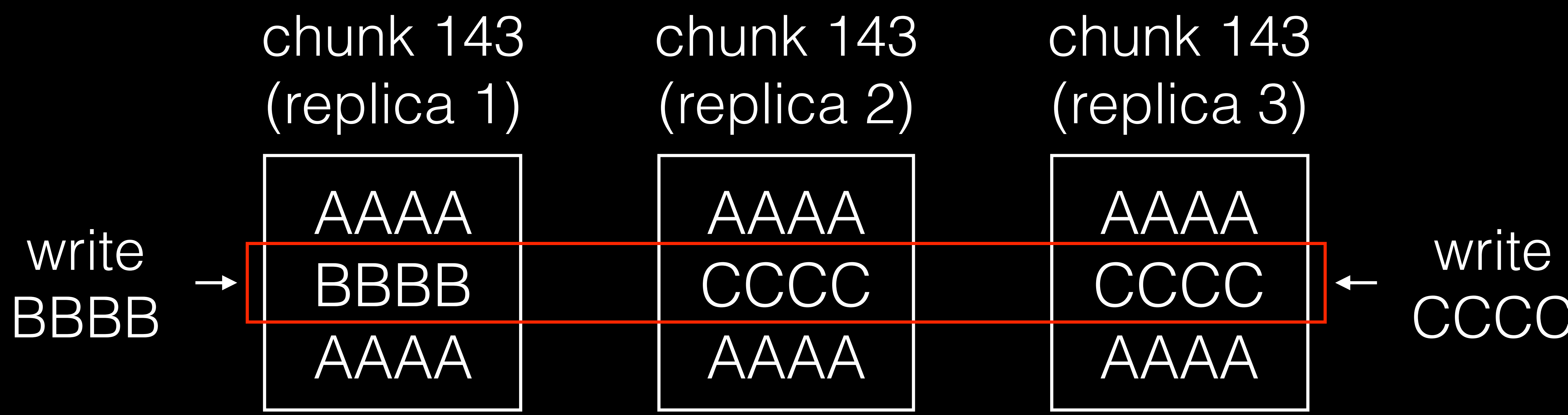
chunk 143 (replica 1)	chunk 143 (replica 2)	chunk 143 (replica 3)
AAAA AAAA AAAA	AAAA AAAA AAAA	AAAA AAAA AAAA

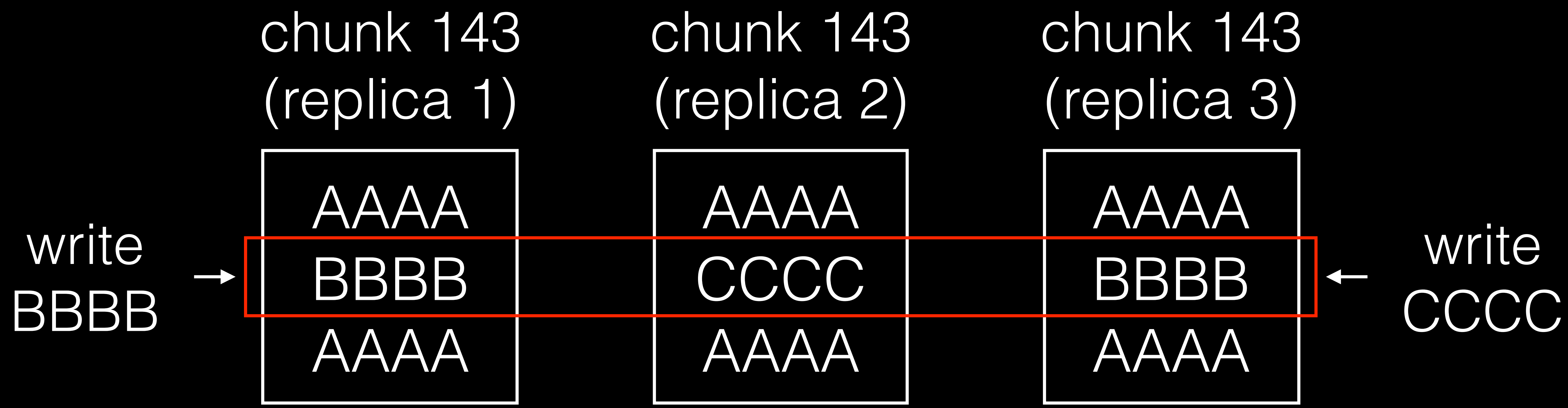

chunk 143
(replica 1)
chunk 143
(replica 2)
chunk 143
(replica 3)
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
write
BBBB
write
CCCC


chunk 143
(replica 1)
chunk 143
(replica 2)
chunk 143
(replica 3)
AAAA
BBBB
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
write
BBBB
write
CCCC

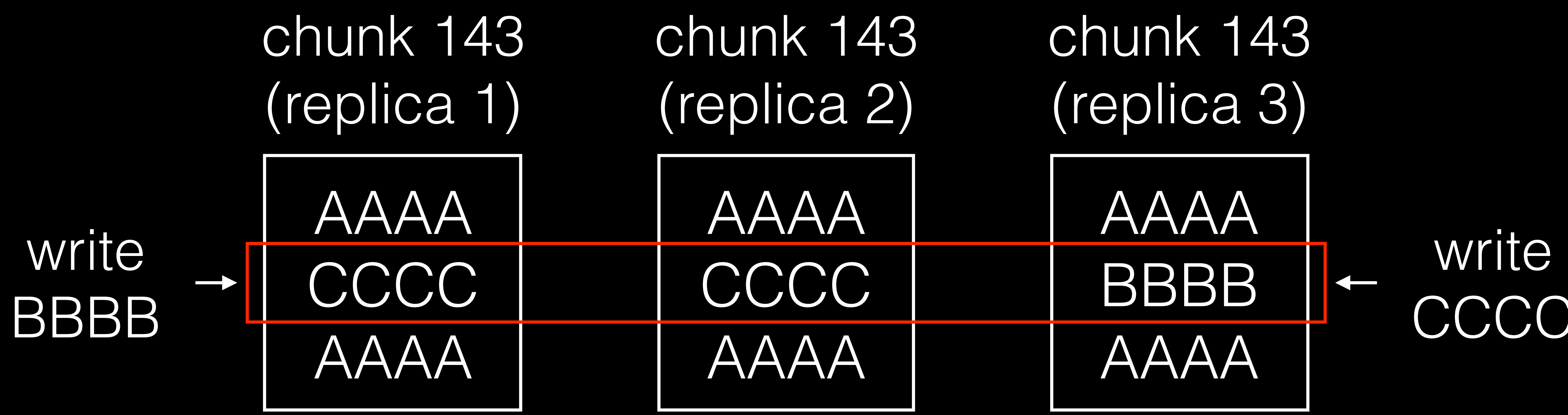

chunk 143
(replica 1)
chunk 143
(replica 2)
chunk 143
(replica 3)
write
BBBB
AAAA
BBBB
AAAA
AAAA
AAAA
AAAA
AAAA
CCCC
AAAA
write
CCCC


chunk 143
(replica 1)
chunk 143
(replica 2)
chunk 143
(replica 3)
AAAA
BBBB
AAAA
AAAA
BBBB
AAAA
AAAA
CCCC
AAAA
write
BBBB
write
CCCC


chunk 143
(replica 1)
chunk 143
(replica 2)
chunk 143
(replica 3)
AAAA
BBBB
AAAA
AAAA
CCCC
AAAA
AAAA
CCCC
AAAA
write
BBBB
write
CCCC



chunk 143
(replica 1)
chunk 143
(replica 2)
chunk 143
(replica 3)
AAAA
BBBB
AAAA
AAAA
CCCC
AAAA
AAAA
BBBB
AAAA
write
BBBB
write
CCCC


chunk 143
(replica 1)
chunk 143
(replica 2)
chunk 143
(replica 3)
write
BBBB
AAAA
CCCC
AAAA
AAAA
CCCC
AAAA
AAAA
BBBB
AAAA
write
CCCC

chunk 143 (replica 1)	chunk 143 (replica 2)	chunk 143 (replica 3)
AAAA	AAAA	AAAA
CCCC	CCCC	BBBB
AAAA	AAAA	AAAA

chunk 143 (replica 1)

AAAA
CCCC
AAAA

chunk 143 (replica 2)

AAAA
CCCC
AAAA

chunk 143 (replica 3)

AAAA
BBBB
AAAA

Chunks disagree, but all checksums are correct!

Worker Consistency: Strategy

We want to “serialize” writes.

That is, we want to decide an order of writes, and make all workers use the same order.

Who to decide order?

Worker Consistency: Strategy

We want to “serialize” writes.

That is, we want to decide an order of writes, and make all workers use the same order.

Who to decide order?
- don’t want to overload master
- let one replica be the primary and decide

Primary Replica

Master decide primary for each logical chunk.

What if primary dies?

Give primary leases that expire after 1 minute.

If master wants to reassign primary, and it can't reach old primary, just wait 1 minute.

GFS Summary

Fight failure with replication.

Metadata consistency is hard, centralize to make it easier.

Data consistency is easier, distribute it for scalability.

MapReduce

Problem

Datasets are too big to process single threaded.

Good concurrent programmers are rare.

Want a concurrent programming framework that is:

- easy to use (no locks, CVs, race conditions)
- general (works for many problems)

MapReduce

Strategy: break data into buckets, do computation over each bucket.

Google published details in 2004.

Open source implementation: Hadoop

Example: Revenue per State

State	Sale	ClientID
WI	100	9292
CA	10	9523
WI	15	9331
CA	45	9523
TX	9	8810
WI	20	9292

How to quickly sum sales in every state without any one machine iterating over all results?

Strategy

One set of processes groups data into logical buckets.

Each bucket has a single process that computes over it.

Strategy

One set of processes groups data into logical buckets. (mappers)

Each bucket has a single process that computes over it. (reducers)

Strategy

One set of processes groups data into logical buckets. (mappers)

Each bucket has a single process that computes over it. (reducers)

Claim: if no bucket has too much data, no single process can do too much work.

MapReduce Overview

Motivation

MapReduce Programming

Implementation

Example: Revenue per State

State	Sale
WI	100
CA	10
WI	15
CA	45
TX	9
WI	20

How to quickly sum sales in every state without any one machine iterating over all results?

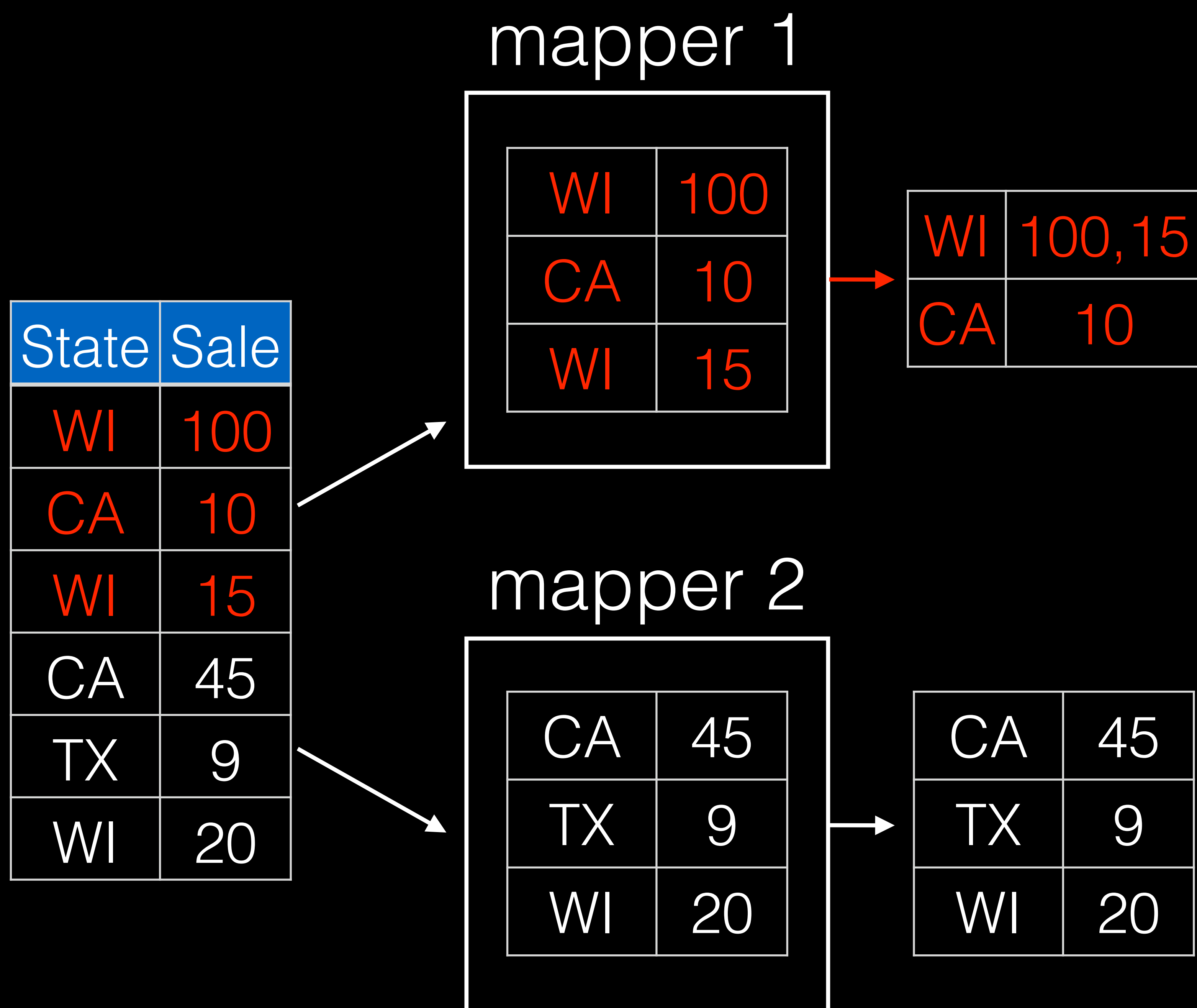
State	Sale
WI	100
CA	10
WI	15
CA	45
TX	9
WI	20

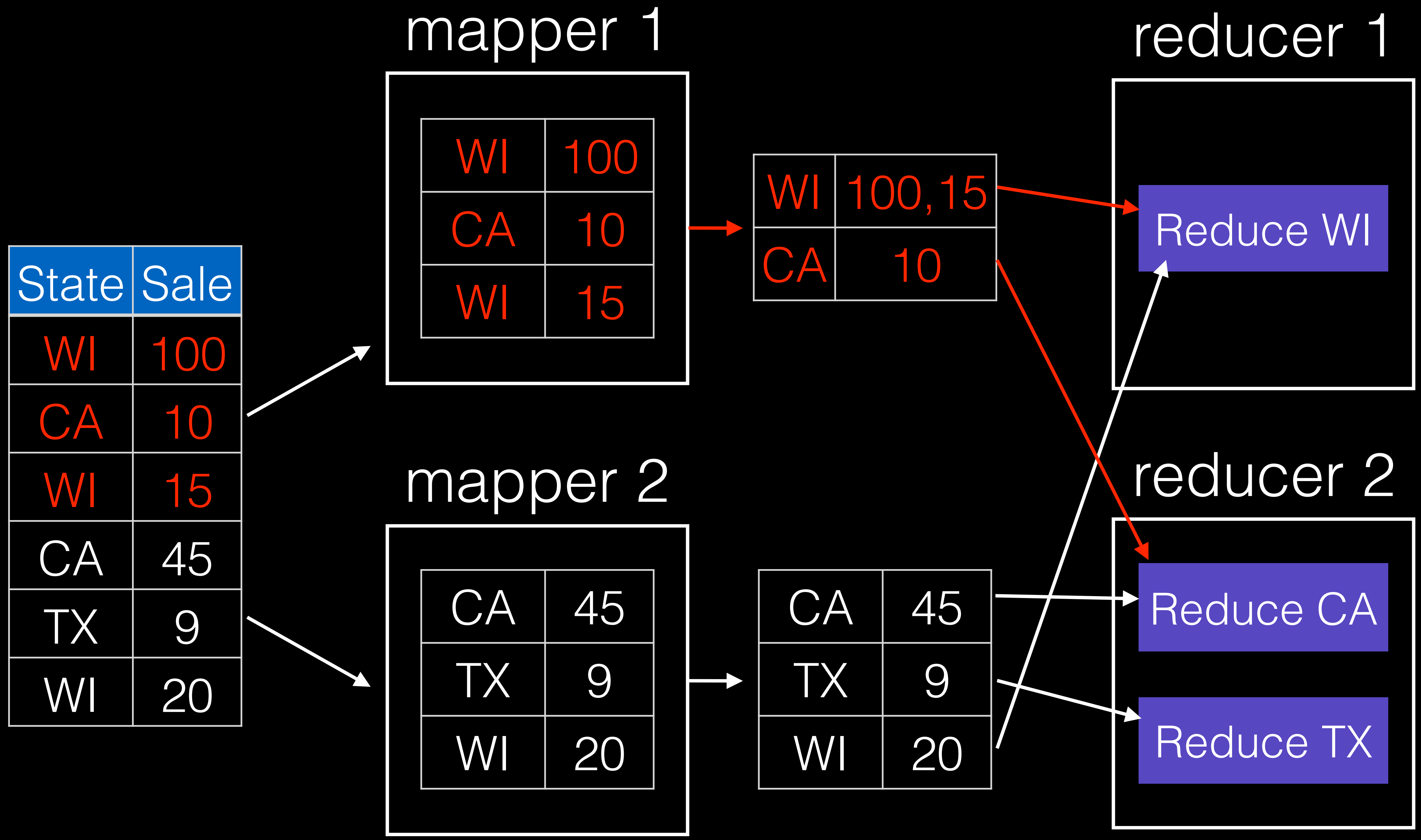
mapper 1

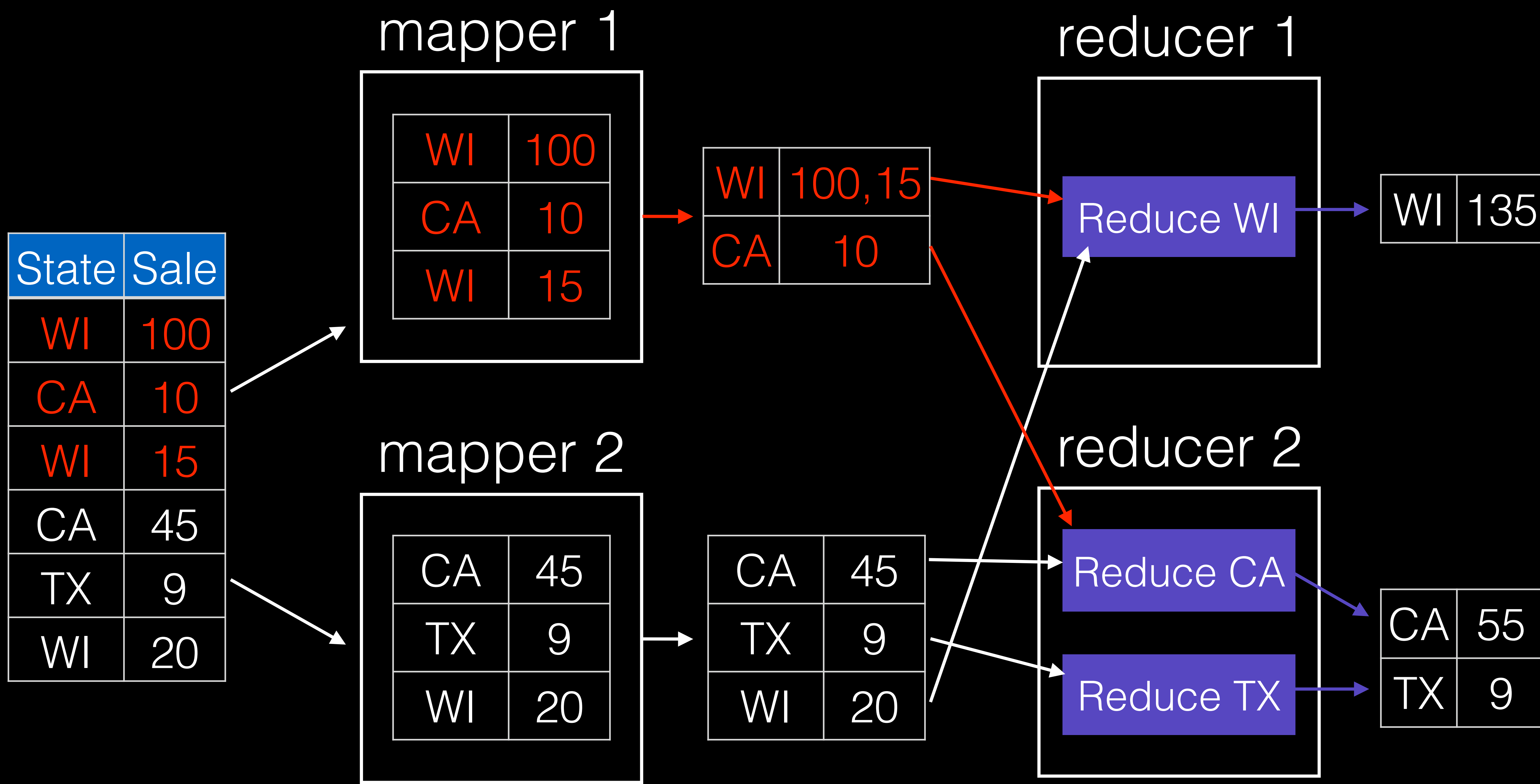
WI	100
CA	10
WI	15

mapper 2

CA	45
TX	9
WI	20


State
Sale
WI 100
CA 10
WI 15
CA 45
TX 9
WI 20
mapper 1
WI 100
CA 10
WI 15
WI 100,15
CA 10
mapper 2
CA 45
TX 9
WI 20
CA 45
TX 9
WI 20


State
Sale
WI
100
CA
10
WI
15
CA
45
TX
9
WI
20
mapper 1
WI
100
CA
10
WI
15
mapper 2
CA
45
TX
9
WI
20
WI
100,15
CA
10
CA
45
TX
9
WI
20
reducer 1
Reduce WI
reducer 2
Reduce CA
Reduce TX


State | Sale
WI | 100
CA | 10
WI | 15
CA | 45
TX | 9
WI | 20
mapper 1
WI | 100
CA | 10
WI | 15
mapper 2
CA | 45
TX | 9
WI | 20
WI | 100,15
CA | 10
CA | 45
TX | 9
WI | 20
reducer 1
Reduce WI
reducer 2
Reduce CA
Reduce TX
WI | 135
CA | 55
TX | 9

Revenue per State

State	Sale	ClientID
WI	100	9292
CA	10	9523
WI	15	9331
CA	45	9523
TX	9	8810
WI	20	9292

Mappers could have grouped by any field desired (e.g., by ClientID).

SQL Equivalents

```sql
SELECT sum(sale)
FROM tbl_sales
GROUP BY state;
```

SQL Equivalents

```sql
SELECT sum(sale)
FROM tbl_sales
GROUP BY clientID;
```

SQL Equivalents

```
SELECT max(sale)
FROM tbl_sales
GROUP BY clientID;
```

SQL Equivalents

reduce

```
SELECT max(sale)
FROM tbl_sales
GROUP BY clientID;
```

map

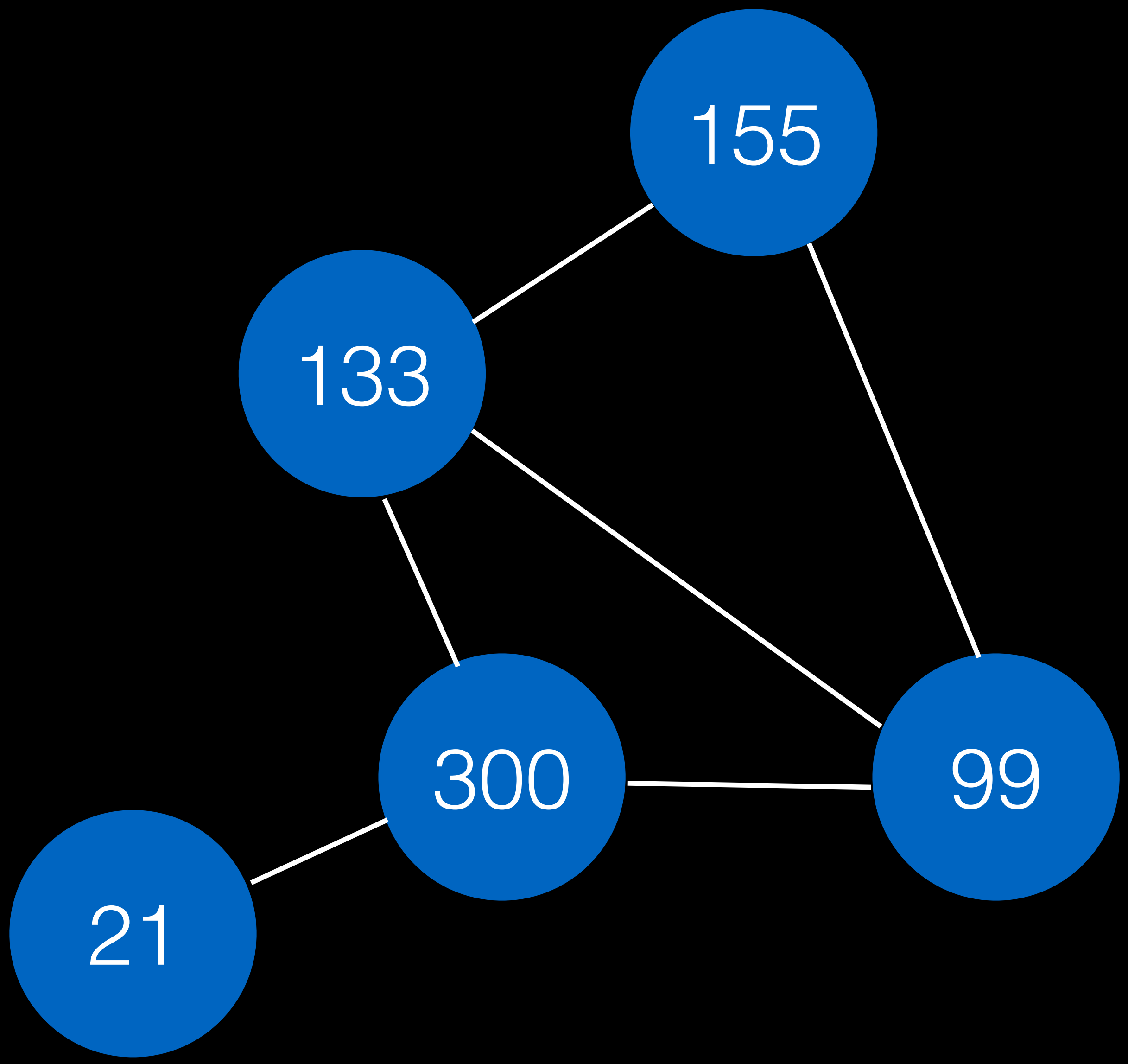
Mapper Output

Sometimes mappers simply classify records (state revenue example).

Sometimes mappers produce multiple intermediate records per input (e.g., friend counts).

Example: Counting Friends

friend1	friend2
133	155
133	99
133	300
300	99
300	21
99	155



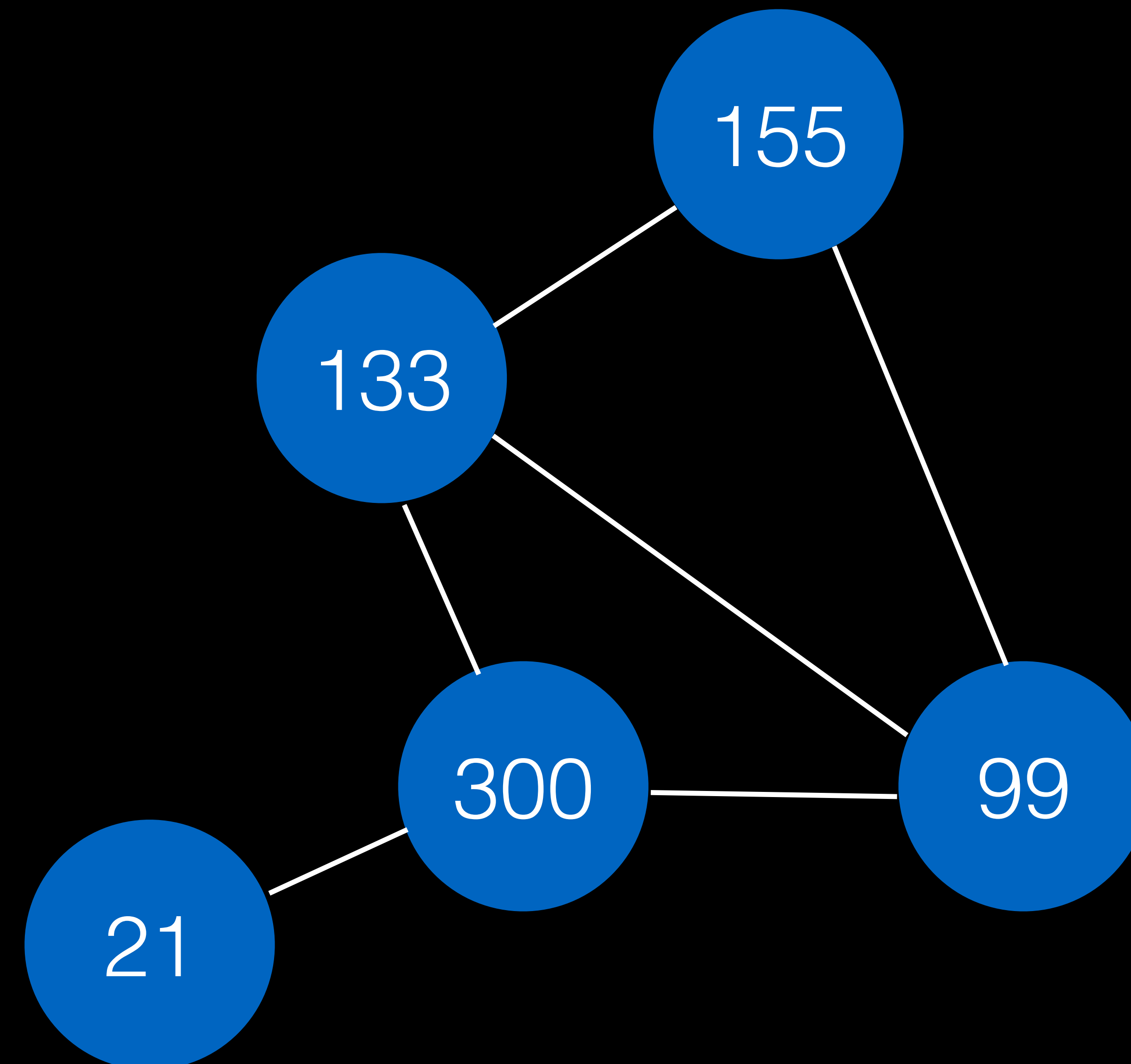
friend1	friend2
133	155
133	99
133	300
300	99
300	21
99	155

mapper 1

133	155
133	99
133	300

mapper 2

300	99
300	21
99	155



friend1	friend2
133	155
133	99
133	300
300	99
300	21
99	155

mapper 1

133	155
133	99
133	300

133	155,99,300
155	133
99	133
300	133

mapper 2

300	99
300	21
99	155

300	99,21
99	300,155
21	300
155	99

Example: Counting Links

url	html
http://	<html><body>…<a href="…
…	…

Many Other Workloads

Distributed grep (over text files)

URL access frequency (over web request logs)

Distributed sort (over strings)

PageRank (over all web pages)

…

Map/Reduce Function Types

```
map(k1,v1) -> list(k2,v2)
reduce(k2,list(v2)) -> list(k3,v3)
```

Hadoop API

```
public void map(LongWritable key, Text value) {
   // WRITE CODE HERE
}

public void reduce(Text key,
                   Iterator<IntWritable> values) {
   // WRITE CODE HERE
}
```

```java
public void map(LongWritable key, Text value) {
   String line = value.toString();
   StringToke st = new StringToke(line);
   while (st.hasMoreTokens())
      output.collect(st.nextToken(), 1);
}

public void reduce(Text key,
                   Iterator<IntWritable> values) {
  int sum = 0;
  while (values.hasNext())
    sum += values.next().get();
  output.collect(key, sum);
}
```

what does
this do?

MapReduce Overview

Motivation

MapReduce Programming

Implementation

MapReduce over GFS

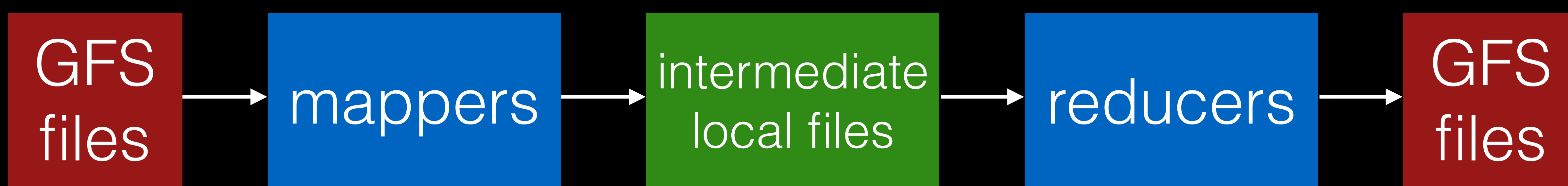
MapReduce writes/reads data to/from GFS.

MapReduce workers run on same machines as GFS workers.

MapReduce over GFS

MapReduce writes/reads data to/from GFS.

MapReduce workers run on same machines as GFS workers.



Why not store intermediate files in GFS?

MapReduce over GFS

MapReduce writes/reads data to/from GFS.

MapReduce workers run on same machines as GFS workers.

Which edges involve network I/O?

MapReduce over GFS

MapReduce writes/reads data to/from GFS.

MapReduce workers run on same machines as GFS workers.

Which edges involve network I/O? Edges 3+4. Maybe 1.

MapReduce over GFS

MapReduce writes/reads data to/from GFS.

MapReduce workers run on same machines as GFS workers.

How to avoid I/O for 1?

Exposing Location

GFS exposes which servers store which files (not transparent, but very useful!)

Hadoop example:

```
BlockLocation[]
getFileBlockLocations(Path p, long start, long len);
```

Spec: return an array containing hostnames, offset and size of portions of the given file.

MapReduce Policy

MapReduce needs to decide which machines to use for map and reduce tasks. Potential factors:

- try to put mappers near one of the three replicas
- for reducers, store one output replica locally
- try to use underloaded machines
- consider network topology

Failed Tasks

A MapReduce master server tracks status of all map and reduce tasks.

If any don't respond to pings, they are simply restarted on different machines.

This is possible because tasks are deterministic, and we still have the inputs.

Slow Tasks

Sometimes a machine gets overloaded or a network link is slow.

With 1000's of tasks, this will always happen.

Spawning duplicate tasks when there are only a few stragglers left reduces some job times by 30%.

MapReduce Summary

MapReduce makes concurrency easy!

Limited programming environment, but works for a fairly wide variety of applications.

Machine failures are easily handled.

Announcements

p5a due Friday.

Office hours today, after class, in lab.

Email sent about final exam topics.