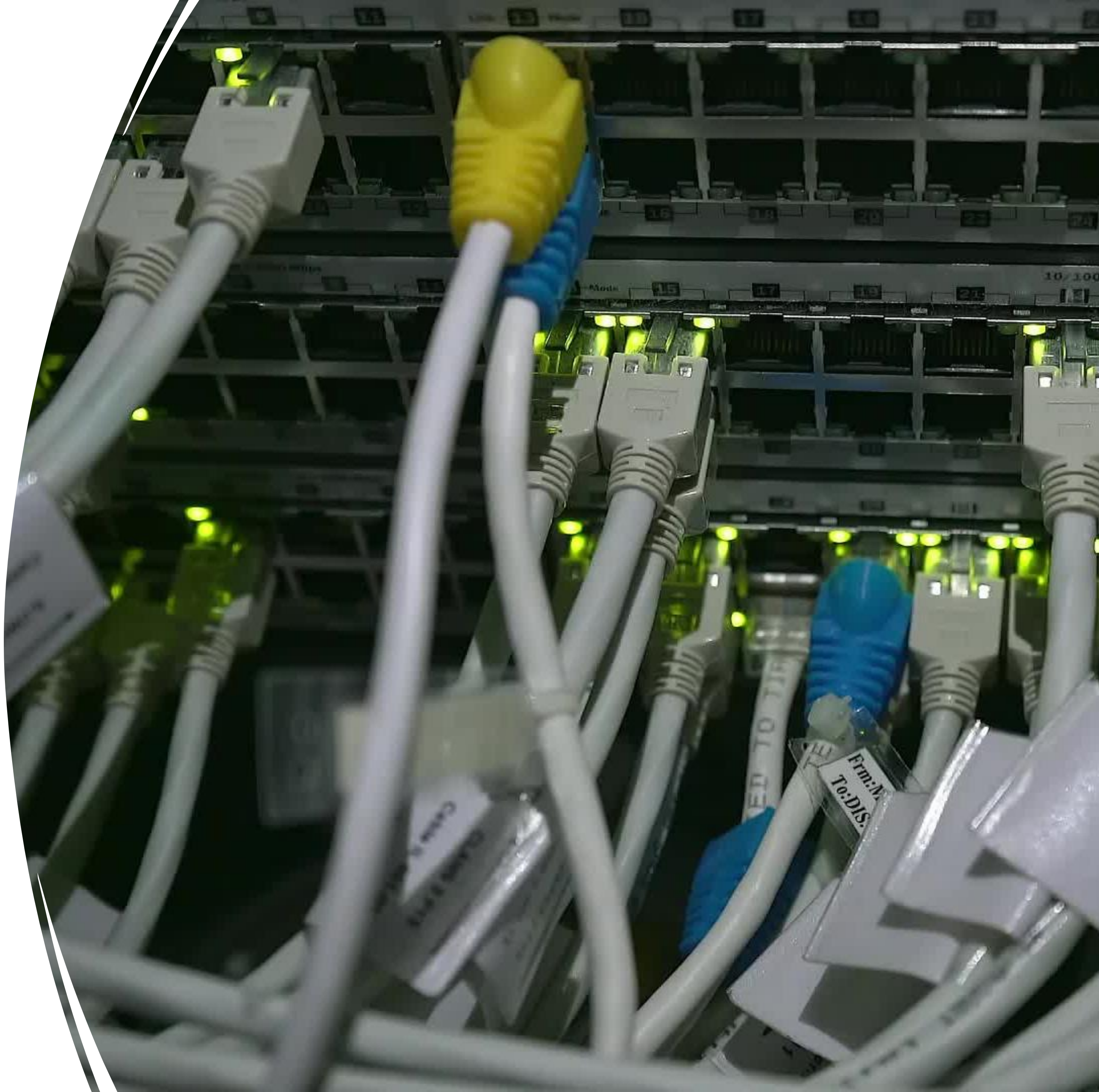
Data-Intensive Distributed Computing CS431/451/651

Module 2 - MapReduce



This Module’s Agenda

Computer Clusters

Distributed Computation (MapReduce)

Distributed Storage

Algorithm Design

Hello, World?

- Something basic to do with a text file:
- How many times does the word “Waterloo” appear?
- We usually did this as the last tutorial in CS116!
- Read lines, Split lines, count “Waterloo”

Word Count in Python

```
counts = Counter()
with open(“file.txt”, “rt”) as file:
  for line in file:
    counts.update(tokenize(line))
```

Believe it or not – this is basically as fast as your HDD

Word Count at Scale

Assume HDD: 100MB/s sustained sequential reads

File Size	Load Time
10MB	0.1 seconds
1GB	10 seconds
10GB	1.67 minutes
100GB	16 minutes
10TB	28 hours

28 hours???

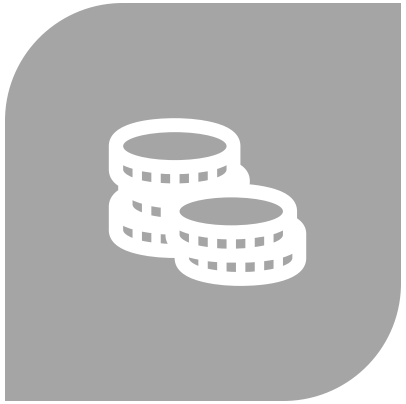
- How can we improve that time?
- NVMe Gen4.0 – 7000 MB/s sequential read
- Only 23 minutes now!
- Price / TB = $150 vs $15 for HDD

Not fast enough?

- You can make a RAID of NVMe drives
- You need an enterprise server to have the PCIe lanes for that

Horizontal vs Vertical

SUPER BEEFY SERVER - $200,000



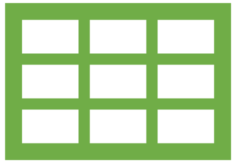
COMMODITY SERVER - $2000

CHEAPER IS BETTER?

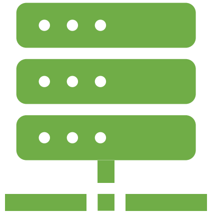
HORIZONTAL SCALING

- 100x the servers, 100x the speed?

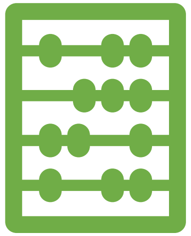
Hello World x100



Each server loads 1/100th of the file



Each server counts "Waterloo"



Add the 100 totals together

Split by lines?

Sorry, can’t split it by line numbers.

Why?

How do you find line number 1723193 in a 10TB file?

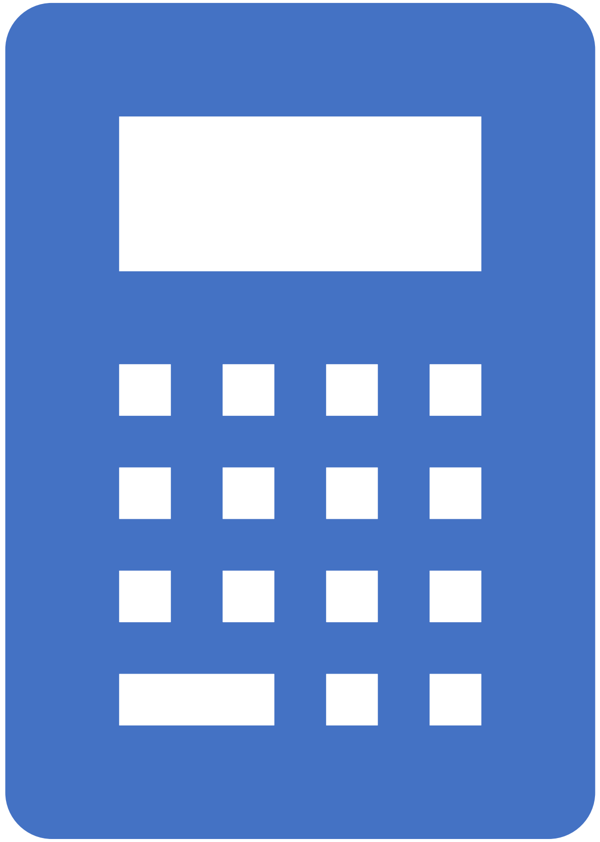
Split by Byes?

Works, but has an issue. Let’s take some text:

Baby Shark, do do do do

Baby Shark, do do do do

What if we split it at this byte?



MapReduce

- Two Functions
- Map: Like* Python’s / Racket’s Map
- Reduce Like* Python’s Reduce

* KINDA

Key-Value Pairs

MapReduce is based around Key-Value Pairs

This is a common way to break things down!

If the input is a text file:

Key – Position of a line (BYTE # not LINE #)

Value – Text of a line.

MapReduce

Programmer defines two functions:

$$\texttt{map: } (k_1, v_1) \longrightarrow \texttt{List}[(k_2, v_2)]$$
$$\texttt{reduce: } (k_2, \texttt{List}[v_2]) \longrightarrow \texttt{List}[(k_3, v_3)]$$

(Those aren’t the actual types – it doesn’t “technically” return things in the programmer sense, but DOES in the mathematician sense).

Map

Input:
- key : k_1
- value : v_1

Output:
- List[(k_2, v_2)]

Note: The output key can be different than the input key!

And usually will be

Map – Counting Waterloo

```
(0 : 'Waterloo is a city in the
Canadian province of Ontario. It
is one of three cities in the
Regional Municipality of Waterloo
(formerly Waterloo County).
Waterloo is situated about 94 km
(58 mi) southwest of Toronto. Due
to the close proximity of the
city of Kitchener to Waterloo,
the two together are often
referred to as "Kitchener-
Waterloo" or the "Twin Cities".')
```

```
(366 : 'While several
unsuccessful attempts to combine
the municipalities of Kitchener
and Waterloo have been made,
following the 1973 establishment
of the Region of Waterloo, less
motivation to do so existed, and
as a result, Waterloo remains an
independent city. At the time of
the 2021 census, the population
of Waterloo was 121,436')
```

map →

```
(('waterloo': 5))

(('waterloo' : 4))
```

Reduce

Input:

- key: k_2
- **ALL** values associated with that key: List[v_2]

Output:

- List[(k_3, v_3)]

Again, the types need not be the same.

They "often" will be.

Reduce – Counting Waterloo

`(‘waterloo’,[4, 5])` → reduce → `(‘waterloo’ : 9)`

Square Peg, Round Hole?

MapReduce requires a key, even though we only need a single integer (the count)

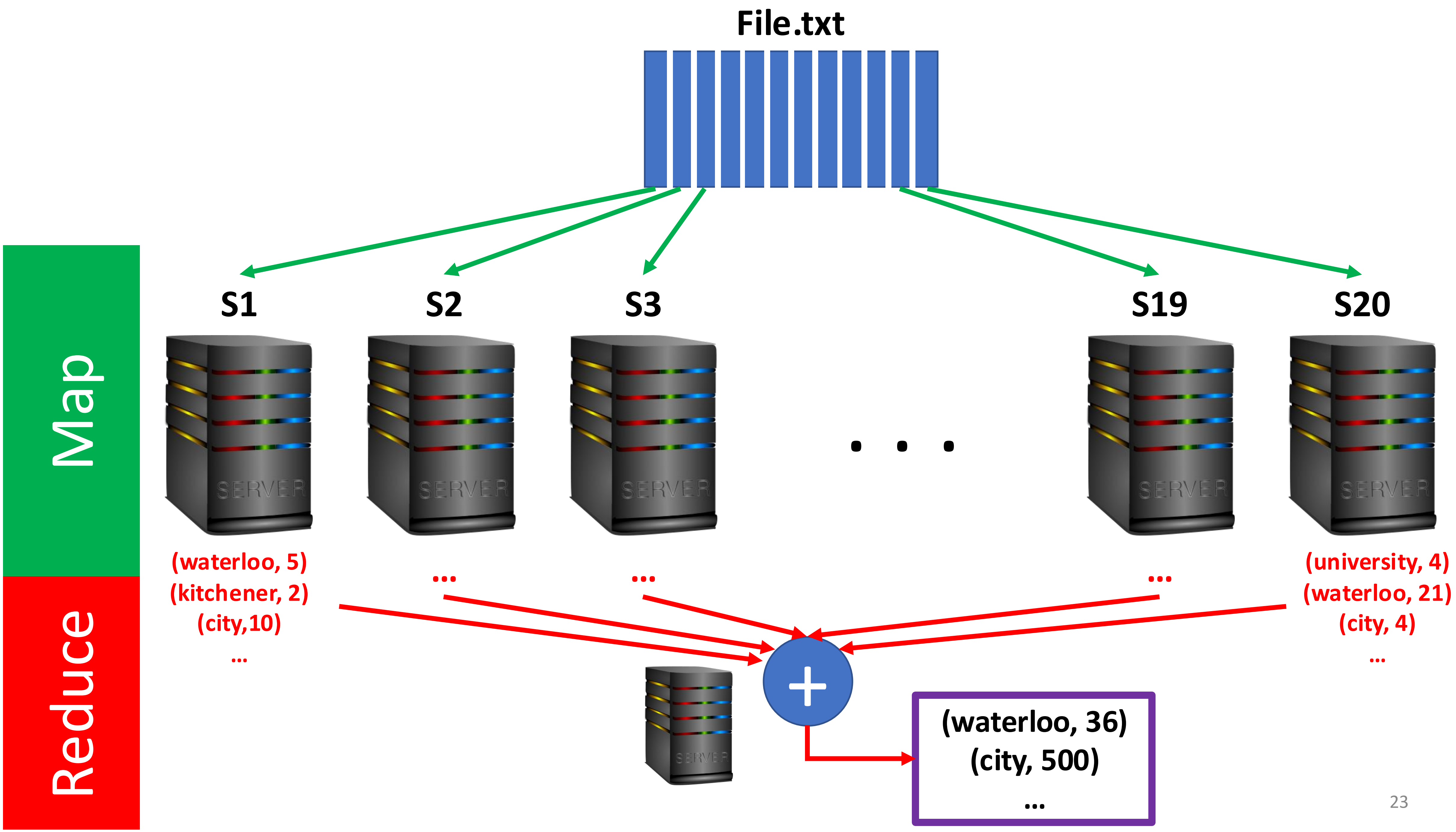
All Word Counts

- From Counter to Map
- Keys are Words, Values are Counts
- Aggregation is now non-trivial
  - (and having a key makes sense)

The expected output is …

- For each word in the input file, count how many times it appears in the file.

Word	Count
Waterloo	36
Kitchener	27
City	512
Is	12450
The	16700
University	123
…	


File.txt
Map
Reduce
S1
S2
S3
S19
S20
SERVER
(waterloo, 5)
(kitchener, 2)
(city,10)
…
…
…
…
(university, 4)
(waterloo, 21)
(city, 4)
…
+
(waterloo, 36)
(city, 500)
…

Memory?

- The Counter used 8 bytes max
- How much does the Dictionary use?
- O(n) if there are n unique words.
- In 10TB of data…what's n?
    - Irrelevant, we're working with a line at a time.
- How many unique words per line?
    - Not many.*

Map – Counting Waterloo, Alternative

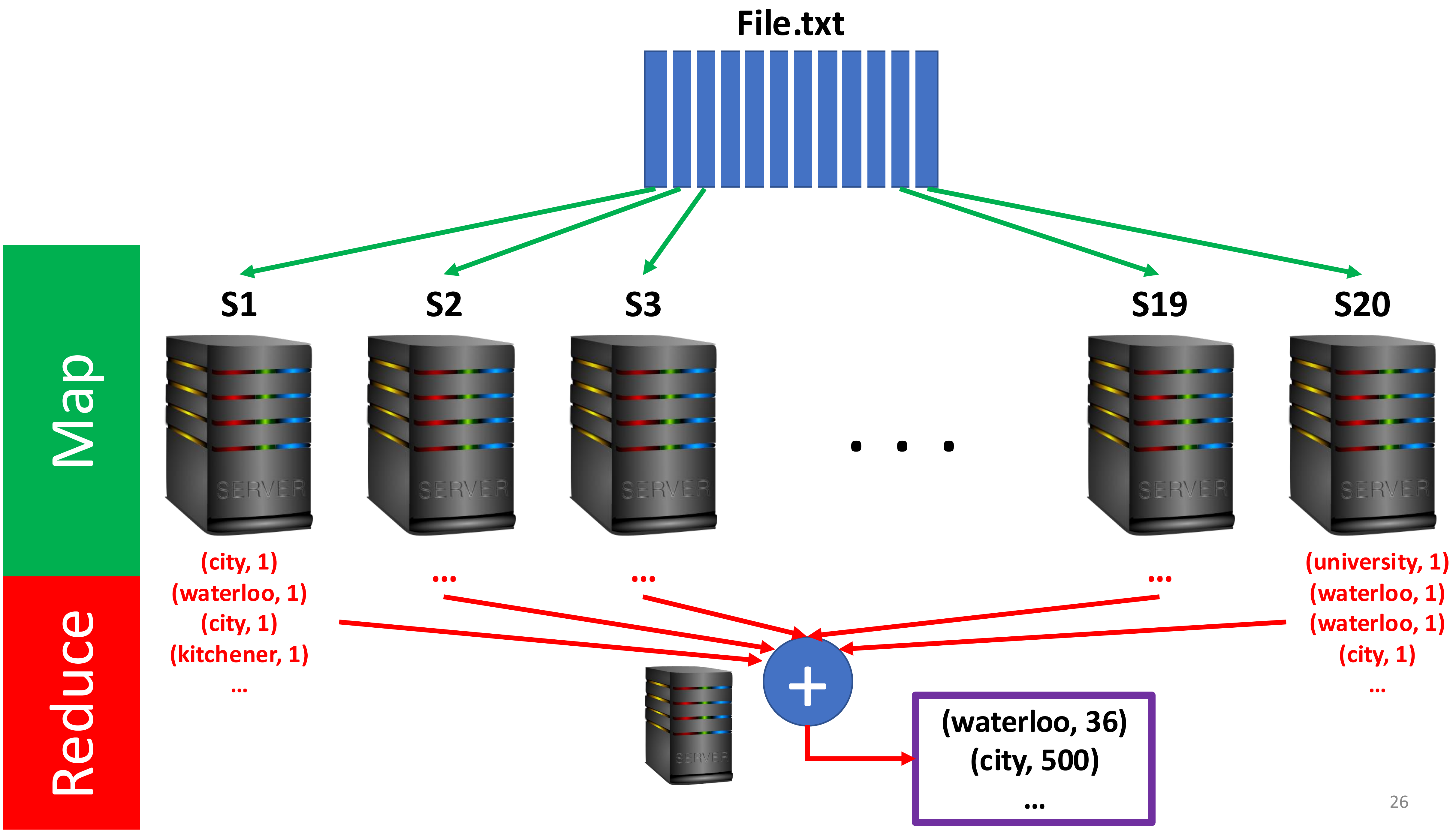
```
(0 : ‘Waterloo is a city in the
Canadian province of Ontario. It
is one of three cities in the
Regional Municipality of Waterloo
(formerly Waterloo County).
Waterloo is situated about 94 km
(58 mi) southwest of Toronto. Due
to the close proximity of the
city of Kitchener to Waterloo,
the two together are often
referred to as "Kitchener-
Waterloo" or the "Twin Cities".’)
```

```
(1 : ‘While several unsuccessful
attempts to combine the
municipalities of Kitchener and
Waterloo have been made,
following the 1973 establishment
of the Region of Waterloo, less
motivation to do so existed, and
as a result, Waterloo remains an
independent city. At the time of
the 2021 census, the population
of Waterloo was 121,436’)
```

map

```
((‘waterloo’: 1), (‘waterloo’: 1), (‘waterloo’: 1)
(‘waterloo’: 1), (‘waterloo’: 1))
```

```
((‘waterloo’ : 1), (‘waterloo’: 1), (‘waterloo’: 1),
(‘waterloo’: 1))
```


File.txt
Map
Reduce
S1
S2
S3
S19
S20
SERVER
. . .
(city, 1)
(waterloo, 1)
(city, 1)
(kitchener, 1)
...
...
...
...
(university, 1)
(waterloo, 1)
(waterloo, 1)
(city, 1)
...
+
(waterloo, 36)
(city, 500)
...

Word Count in MapReduce

```
def map(line):
  for word in line:
    emit(word, 1)

def reduce(key, values):
  sum = 0
  for v in values:
    sum += v
  emit(key, sum)
```

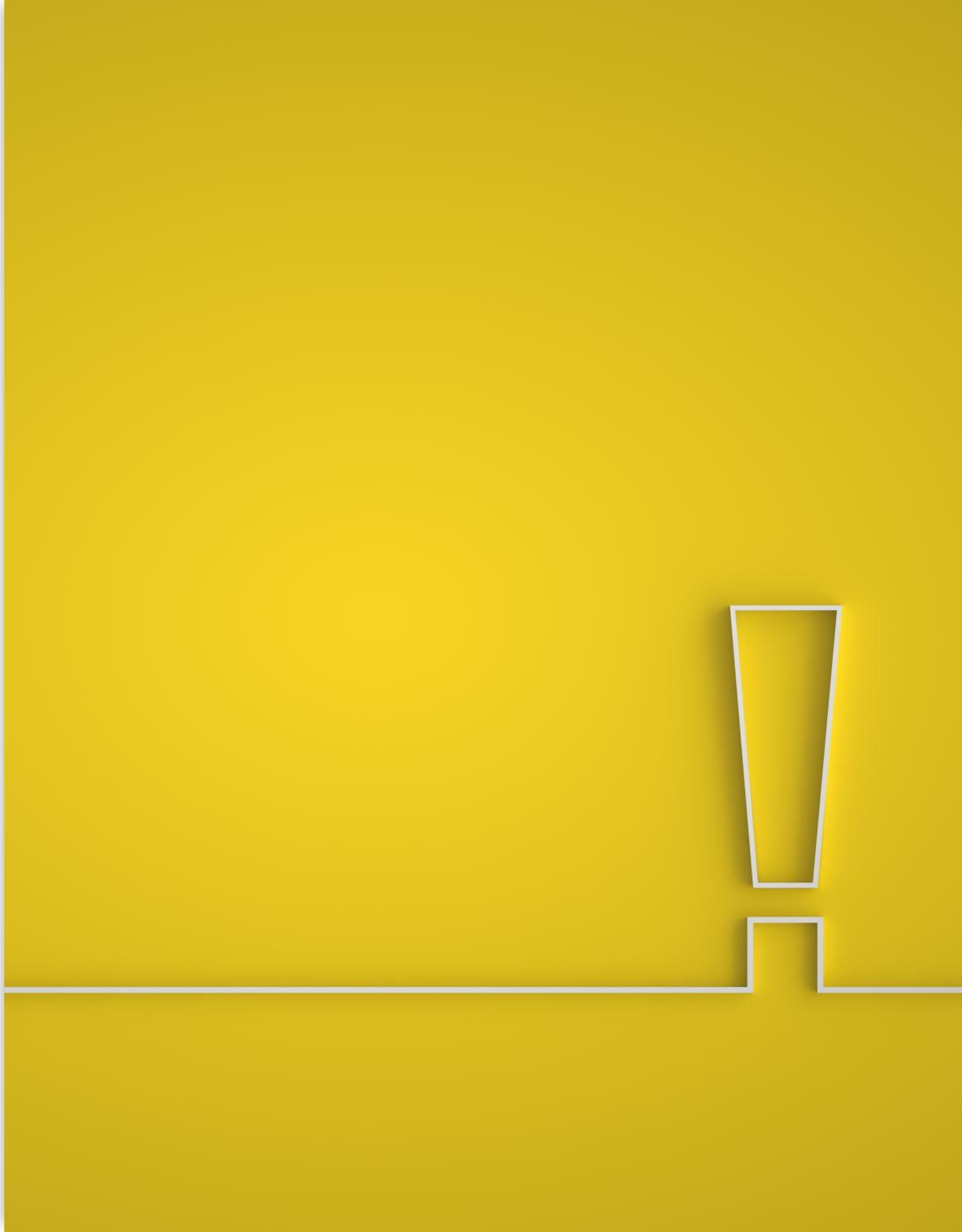
The textbook calls it emit so I’m doing the same. In MapReduce code it’s “context.write”

Problem

The Reduce server is getting too much data! If the file was 10TB, then more than 10TB will arrive!

Why? “some text” => (some,1) (text,1)

Slightly larger!



Distribution

What if you have multiple reducers?

Each reducer gets ALL pairs for a given Key

MapReduce

Programmer defines ~~two~~ three functions:

$$\texttt{map: } (k_1,\ v_1) \longrightarrow \texttt{List}[(k_2,\ v_2)]$$
$$\texttt{reduce: } (k_2,\ \texttt{List}[v_2]) \longrightarrow \texttt{List}[(k_3,\ v_3)]$$
$$\texttt{partition: } (k_2, v_2, n \in \mathbb{N}) \longrightarrow [0, n)$$

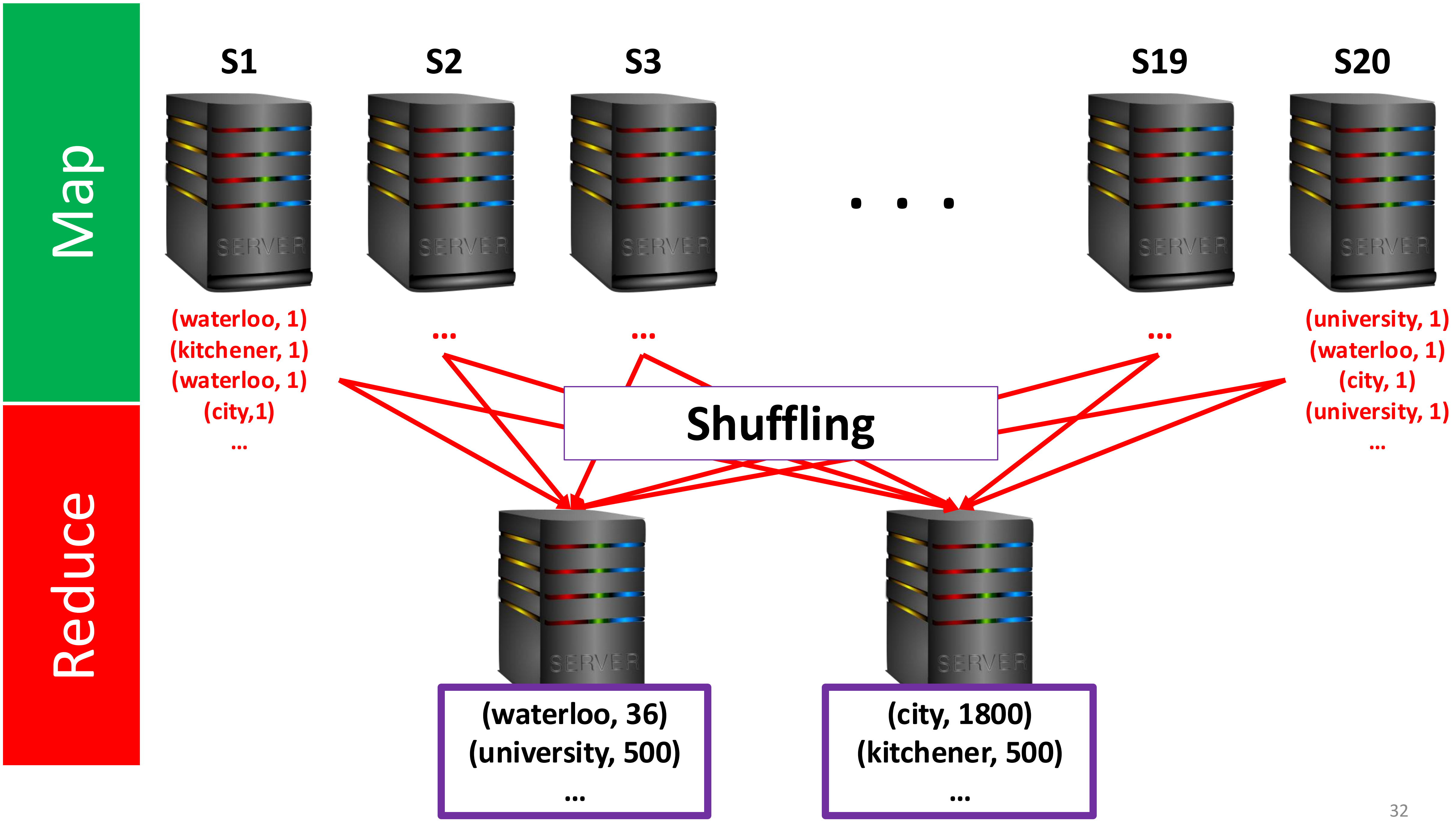
Partition will default to a hash function that hashes the key and ignores the value

Word Count in MapReduce, Less Pseudo, More Code

```
def map(pos : Long, text: String):
  for word in tokenize(text):
    emit(word, 1)

def reduce(key: String, values: Iterator[Int]):
  sum = 0
  for v in values:
    sum += v
  emit(key, sum)

def partition(key : String, value: Int, reducer_count: Nat):
  return hashcode(key) % reducer_count
```


Map
Reduce
S1
S2
S3
S19
S20
(waterloo, 1)
(kitchener, 1)
(waterloo, 1)
(city,1)
…
…
…
…
(university, 1)
(waterloo, 1)
(city, 1)
(university, 1)
…
Shuffling
(waterloo, 36)
(university, 500)
…
(city, 1800)
(kitchener, 500)
…

So, you want to drive the elephant!

MapReduce Implementations

Google has a proprietary implementation in C++

Bindings in Java, Python

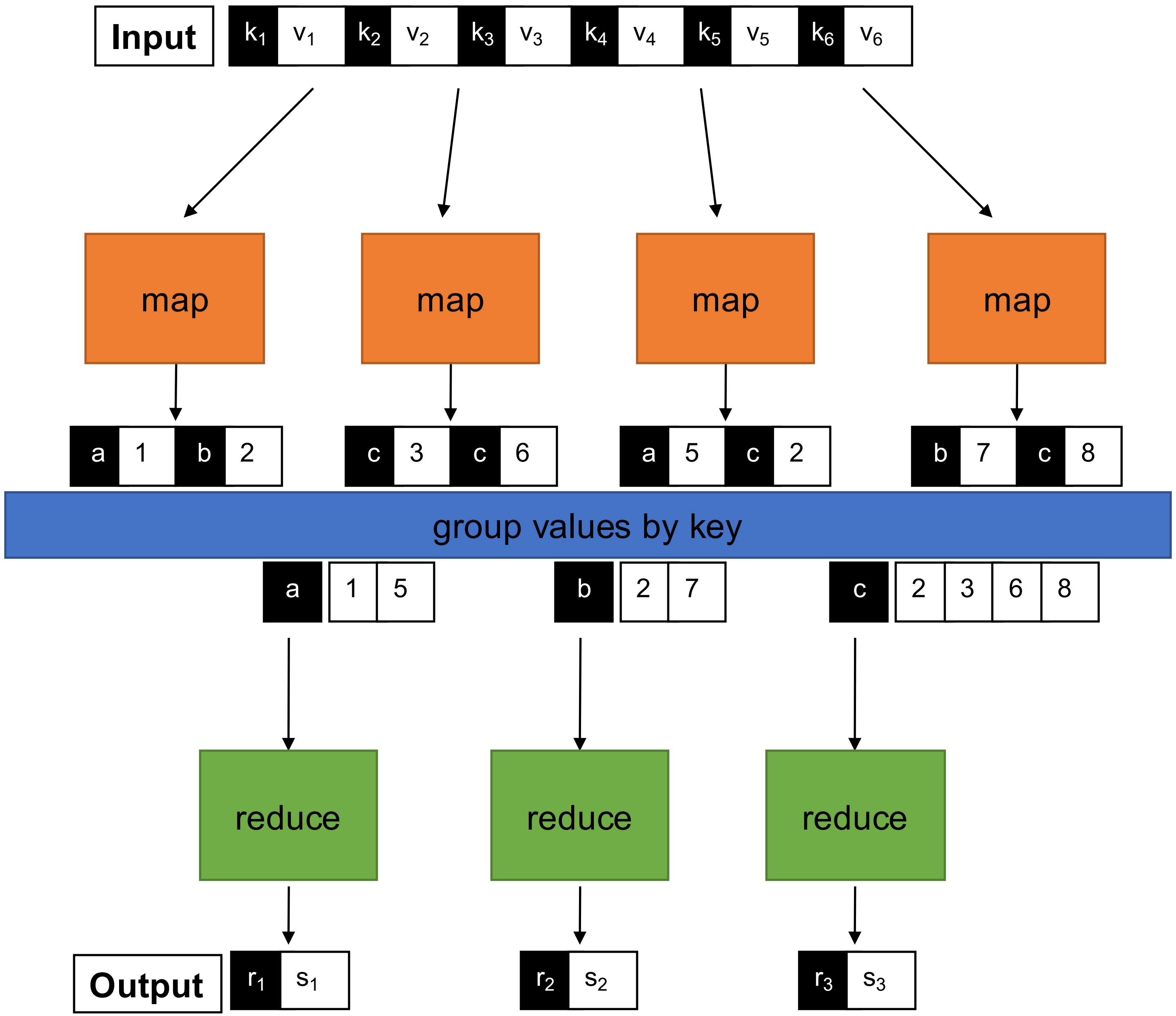
Hadoop provides an open-source implementation in Java

Development begun by Yahoo, later an Apache project
Used in production at Facebook, Twitter, LinkedIn, Netflix, …
Large and expanding software ecosystem
Potential point of confusion: Hadoop is more than MapReduce today

Lots of custom research implementations

Framework

- Assigns workers to map and reduce tasks
- Divides data between map workers*
- Groups intermediate values
  - Sorting pairs by key, determining which pairs go to which reduce worker
- Handles errors
  - What if a worker fails / crashes?


Input
k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
group values by key
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
Output
r1 s1
r2 s2
r3 s3

Faster???

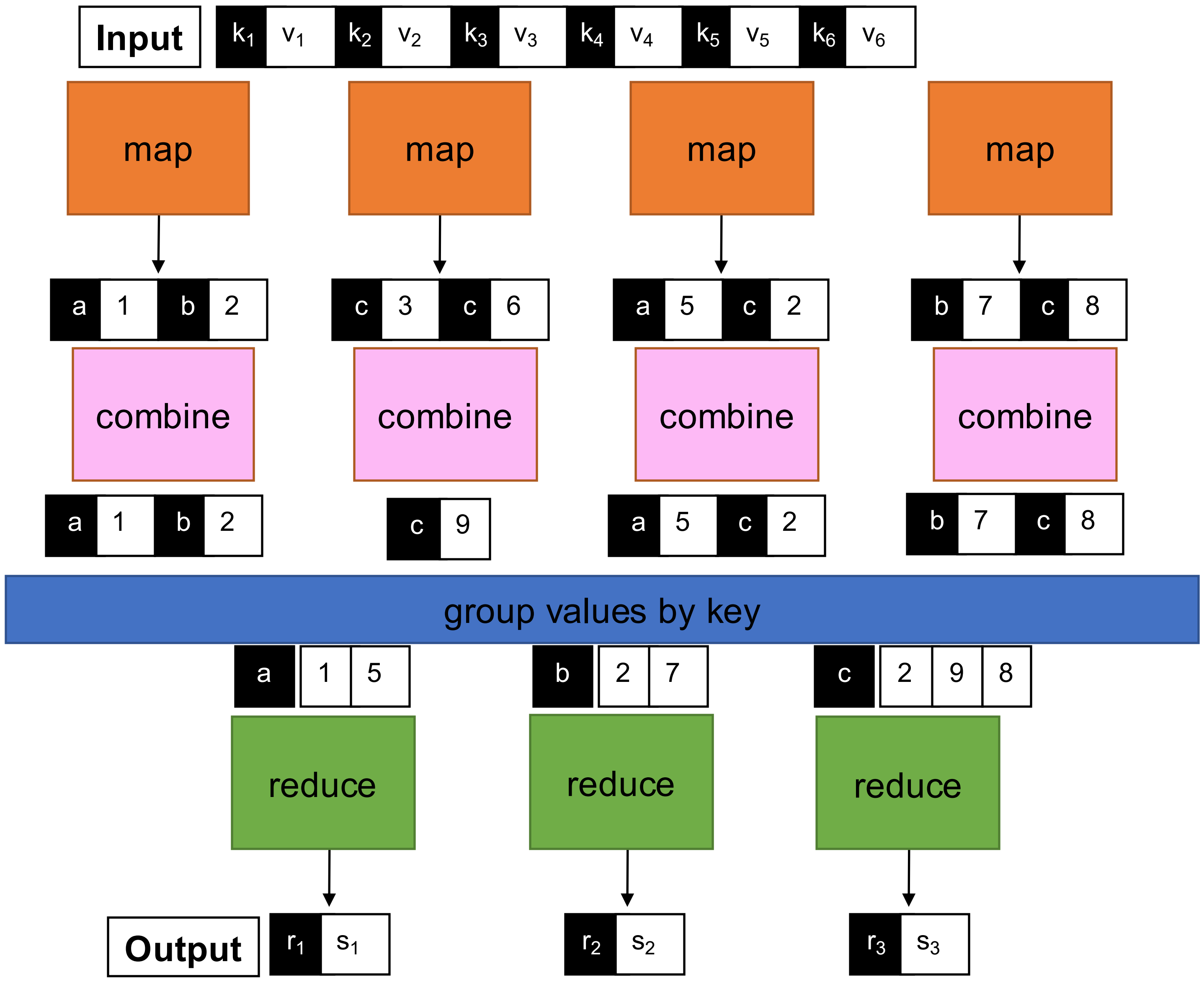
- How about only one value per key per mapper?

```
def combine(key, values):
  sum = 0
  for v in values:
    sum += v
  emit(key, sum)
```

MapReduce

Programmer defines ~~three~~ four functions:

```
map: (k₁, v₁) ⟶ List[(k₂, v₂)]
combine: (k₂,List[v₂]) -> List[(k₂, v₂)]
reduce: (k₂, List[v₂]) ⟶ List[(k₃, v₃)]
partition: (k₂,ℕ) ⟶ ℕ
```


Input
k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combine
combine
combine
combine
a 1 b 2
c 9
a 5 c 2
b 7 c 8
group values by key
a 1 5
b 2 7
c 2 9 8
reduce
reduce
reduce
Output
r1 s1
r2 s2
r3 s3

Combine

- Combine MIGHT be the same as reduce
    - **if** $k_2 = k_3$, $v_2 = v_3$ then it would be legal to do
- It also might not
    - Even if legal, it might be inappropriate! Meaning, it runs but gives the wrong answer

Averages

- Combine can’t be the same as Reduce
- Why?
  - Mean(2, 3, 4) => 3
  - Mean((Mean(2, 3), 4) => 3.25

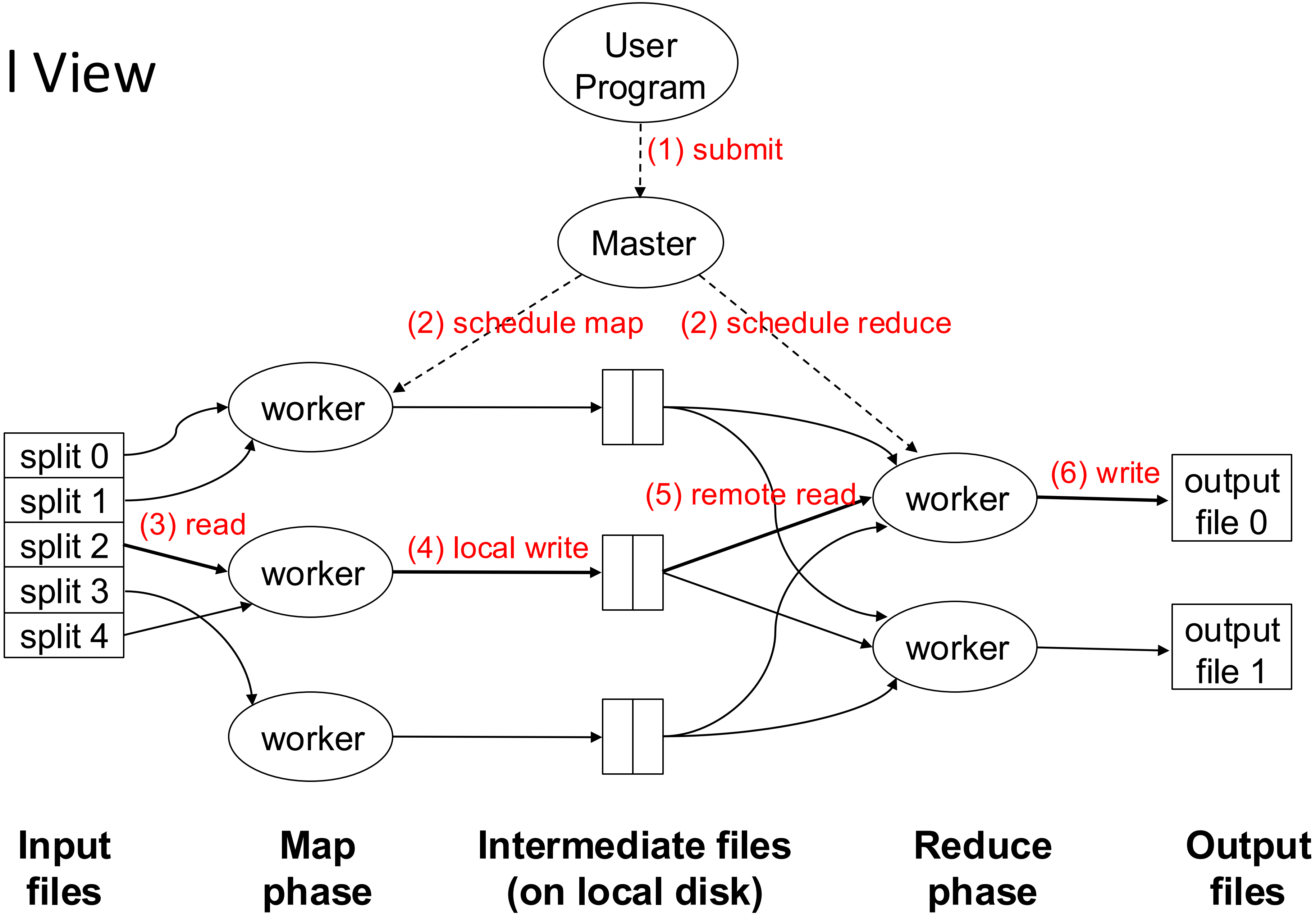
Physical View

Break time! Enough word count, let's talk systems engineering!

OR

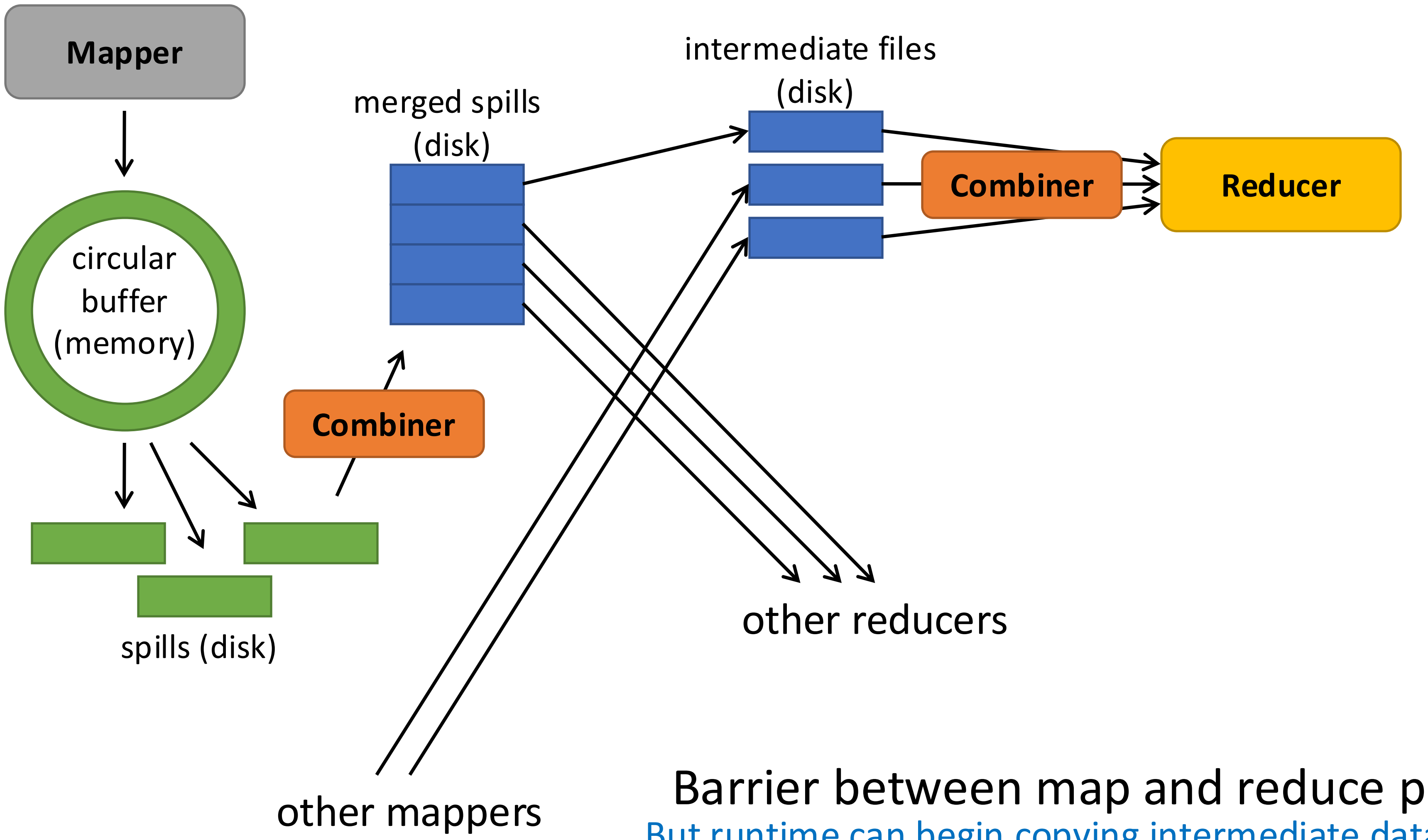
"What's Hadoop doing behind the scenes?"

Physical View



Adapted from (Dean and Ghemawat, OSDI 2004)

Distributed Group By in MapReduce



Barrier between map and reduce phases

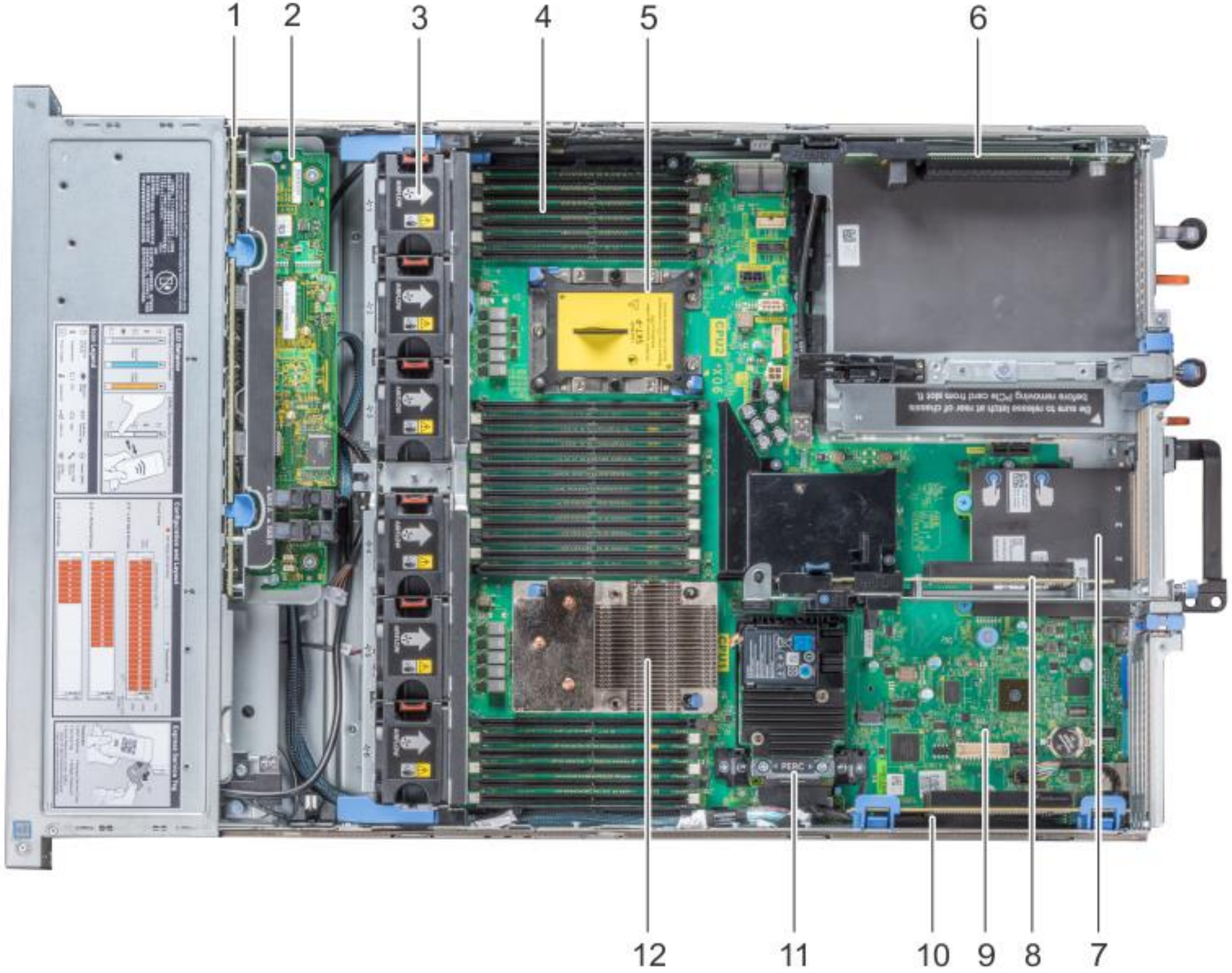
But runtime can begin copying intermediate data earlier

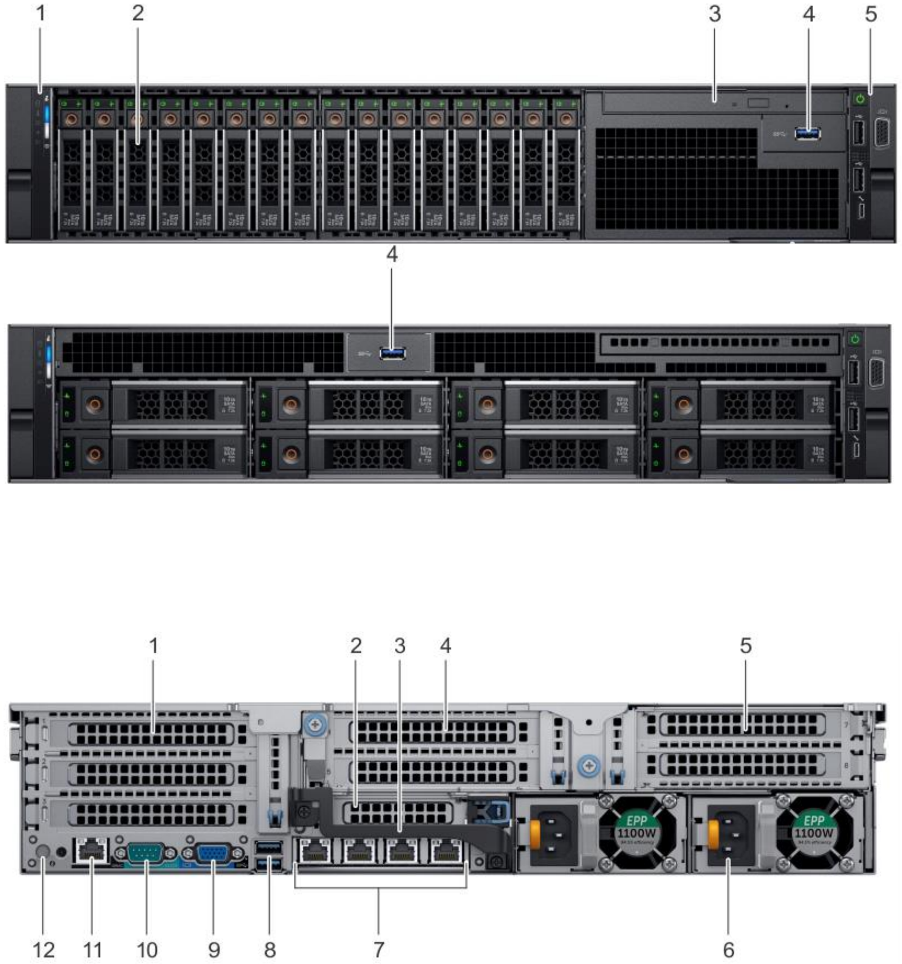
Let’s Get (More) Physical

What does a data center really look like?

Really.

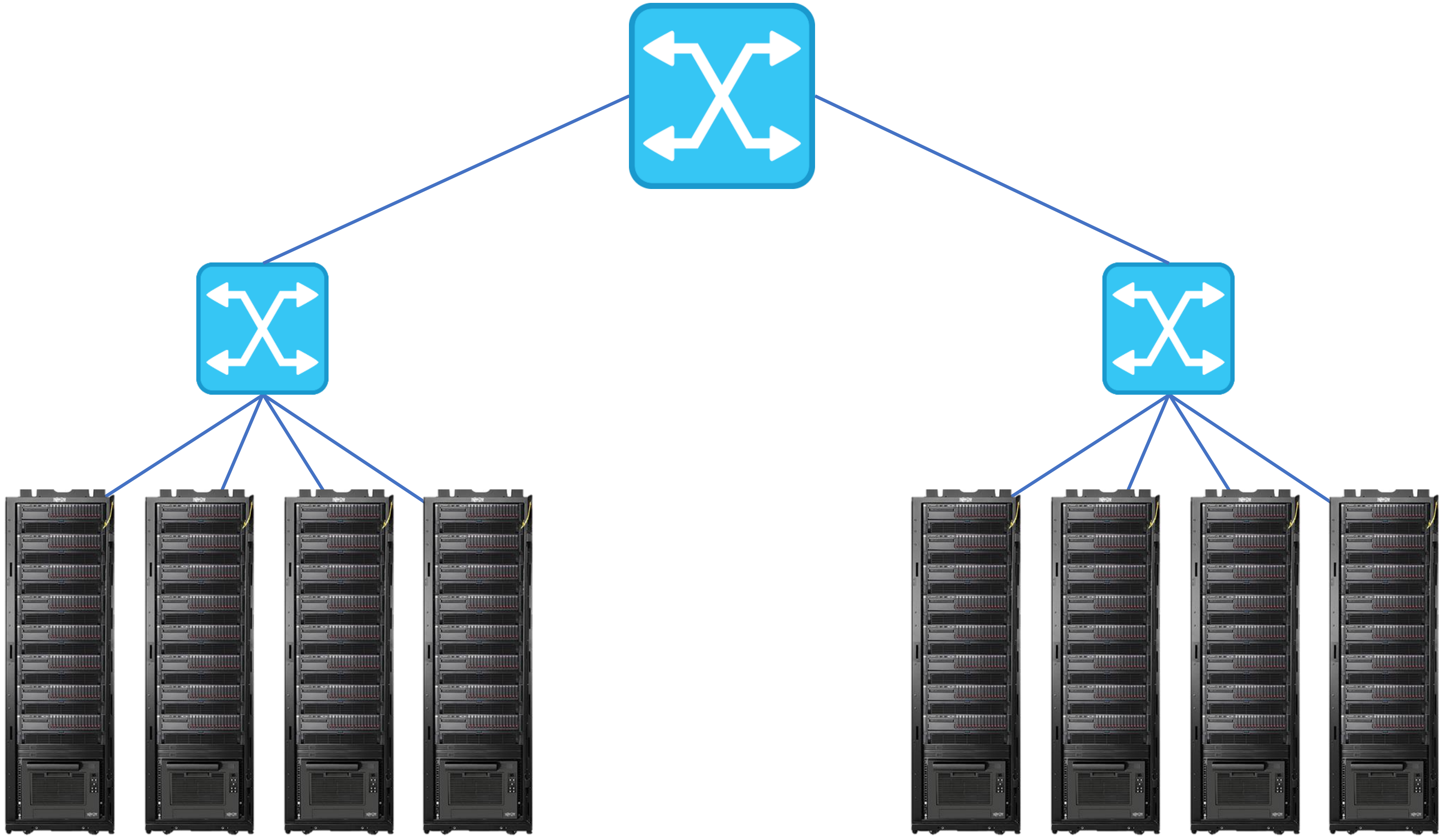
The anatomy of a server

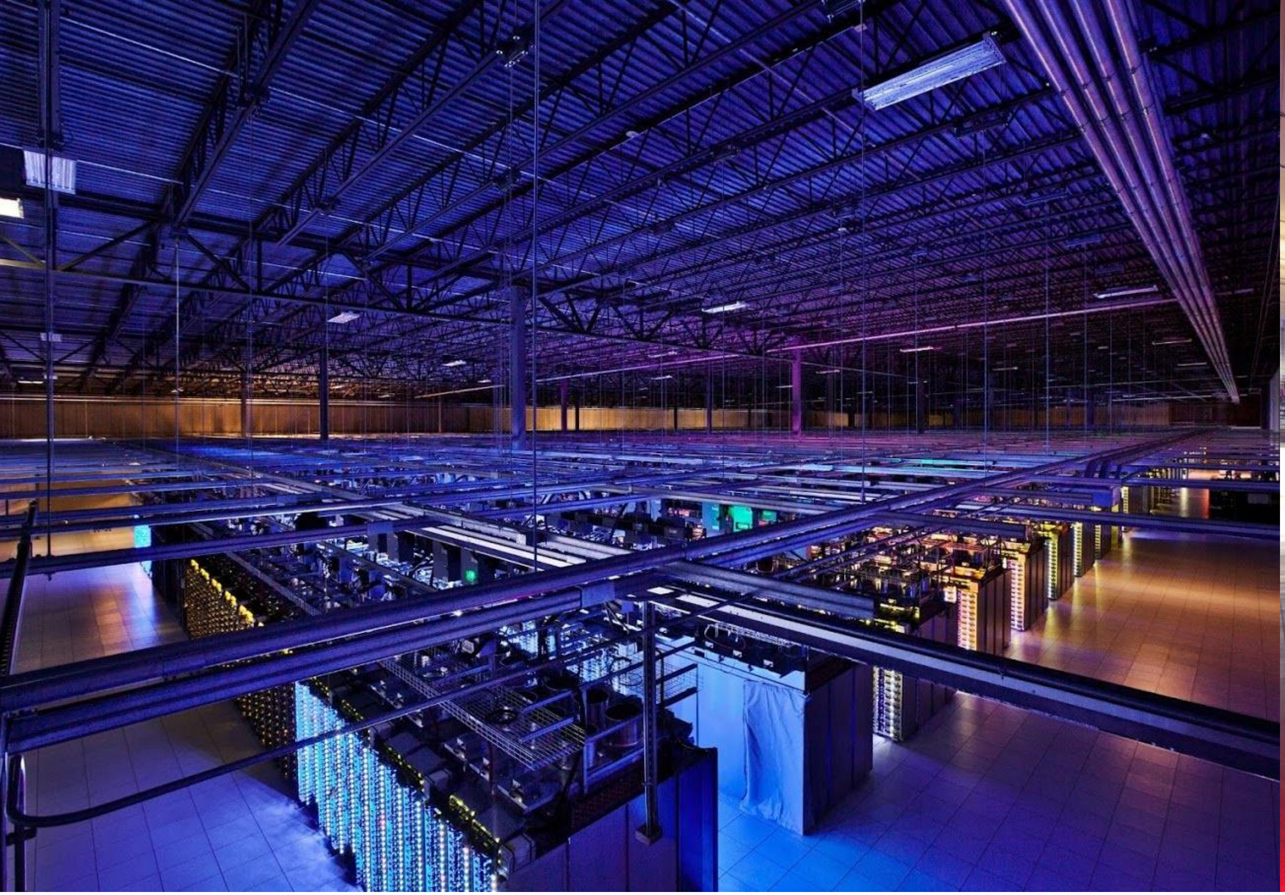




The anatomy of a server rack

The anatomy of a data center

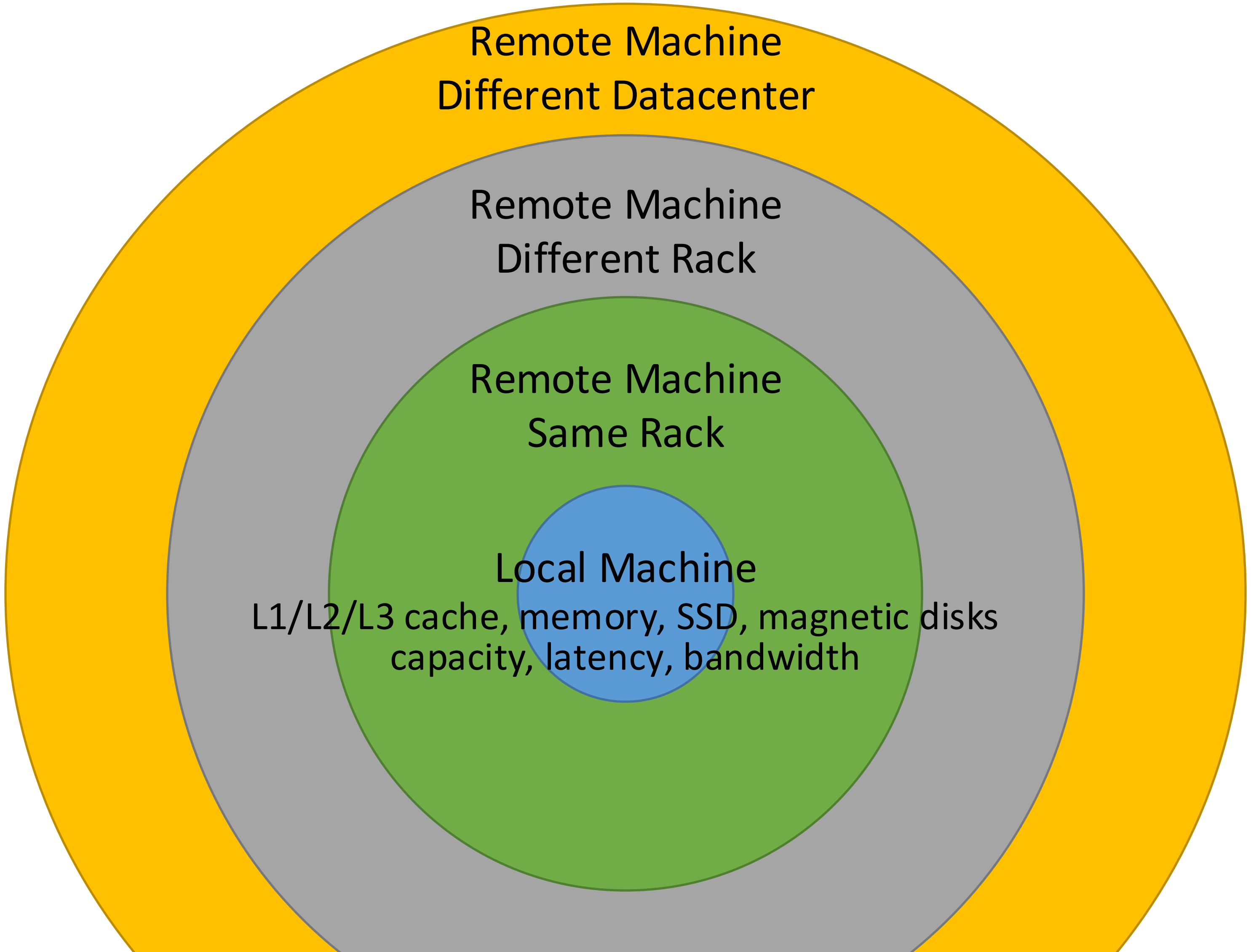


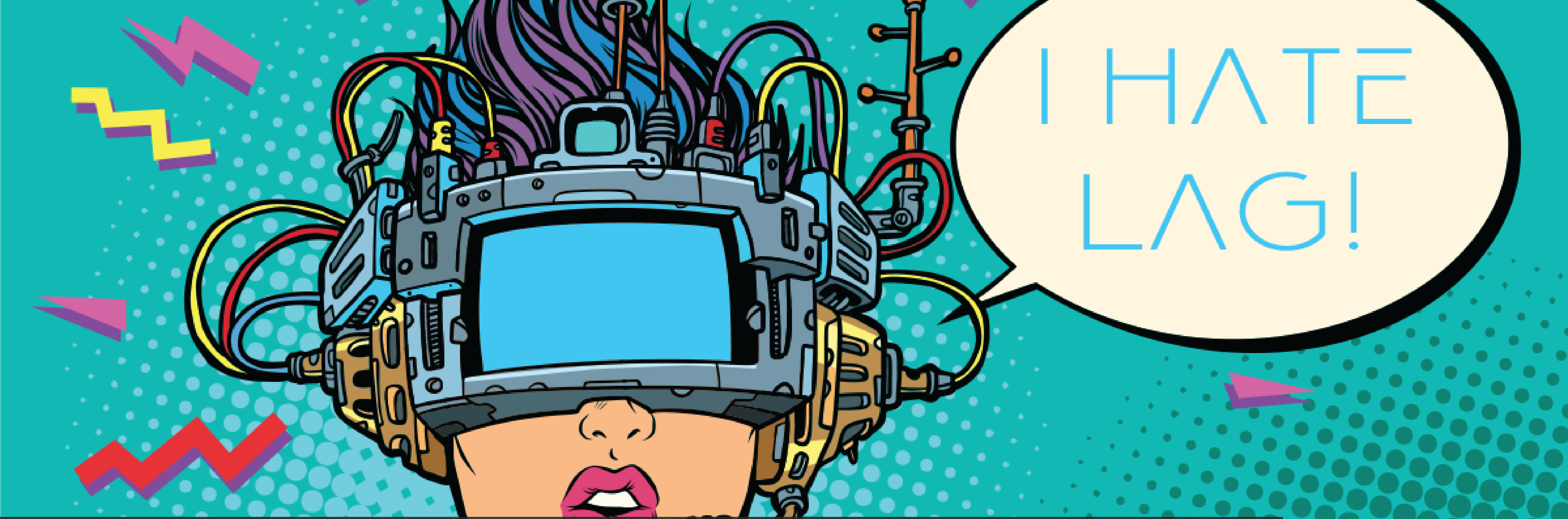


The anatomy of a data center

Google's data center video

Storage Hierarchy





Latency numbers every programmer should know

Demo

Distributed File System

How can we store a large file on a distributed system?

File.txt

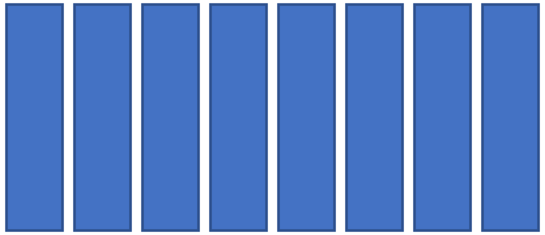
200 TB

How do you store this file?

S1 S2 S3 . . . S19 S20

100 TB 100 TB 100 TB 100 TB 100 TB

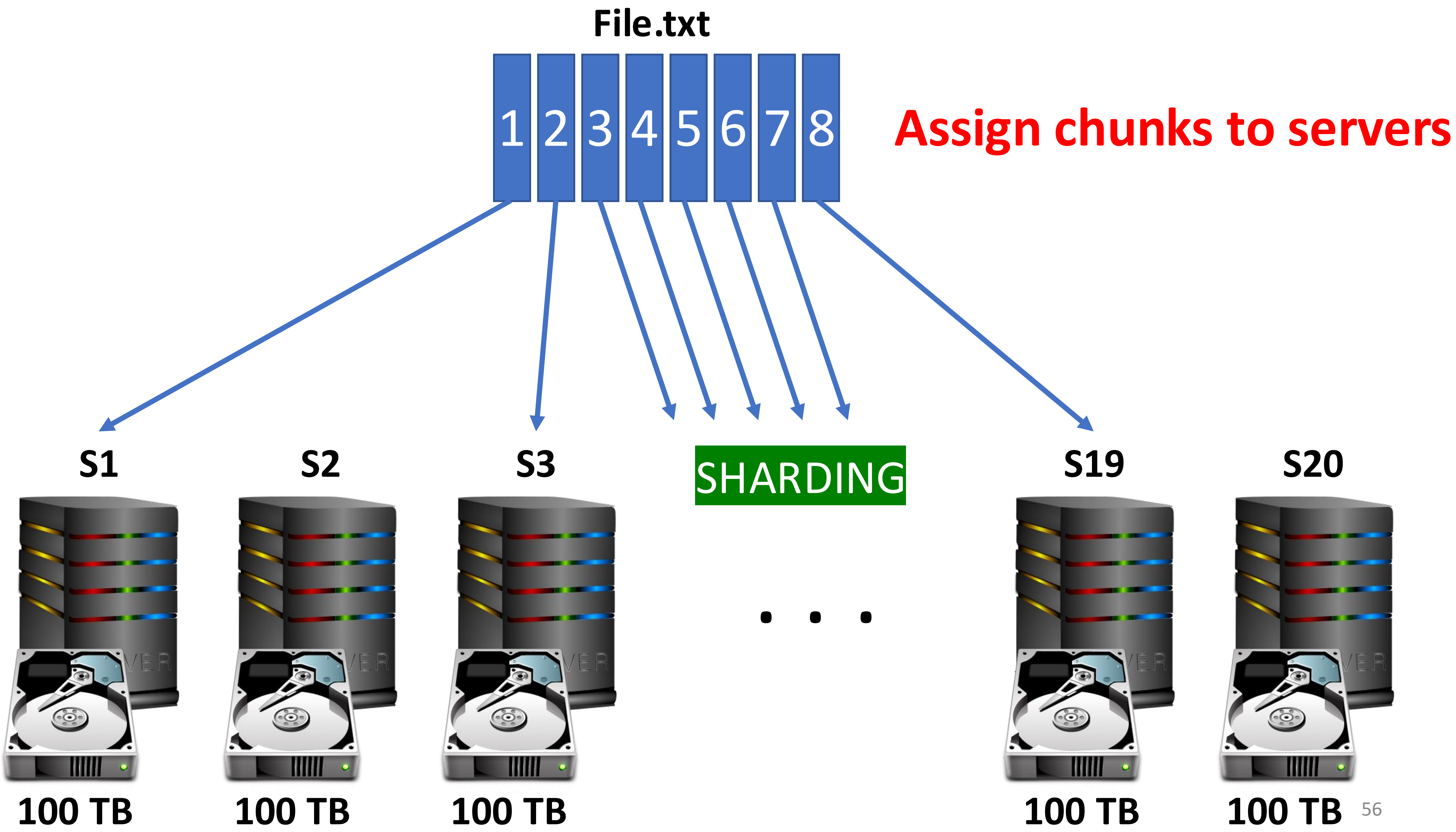
File.txt



Divide into smaller chunks

S1 S2 S3 . . . S19 S20

100 TB 100 TB 100 TB 100 TB 100 TB


File.txt
1
2
3
4
5
6
7
8
Assign chunks to servers
S1
S2
S3
SHARDING
S19
S20
. . .
100 TB
100 TB
100 TB
100 TB
100 TB

File.txt

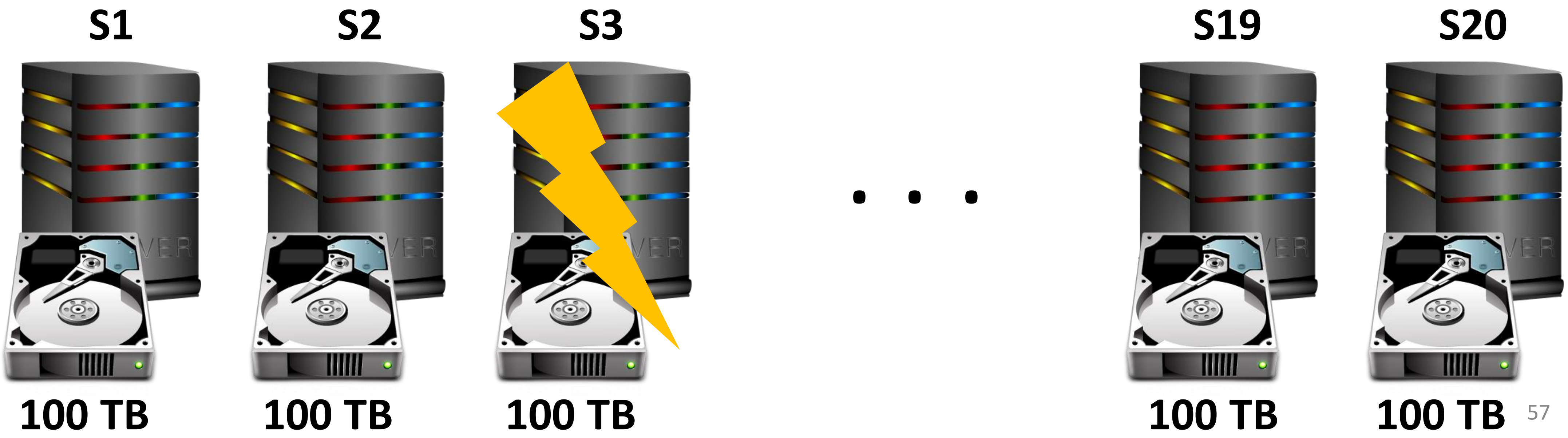
1 → S1

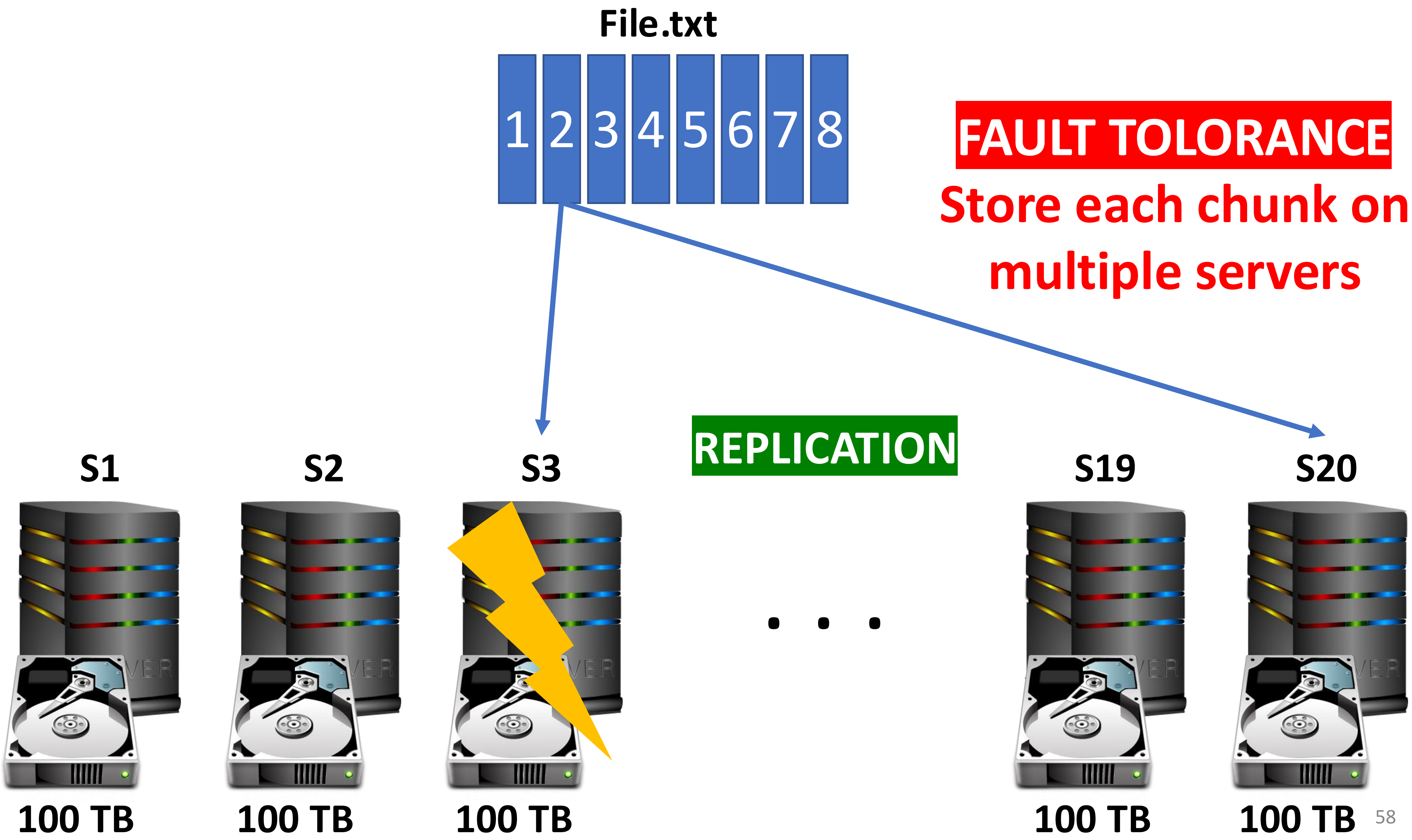
~~2 → S3~~

…

8 → S19

What happens when a server fails?!




File.txt
1 2 3 4 5 6 7 8
FAULT TOLORANCE
Store each chunk on multiple servers
REPLICATION
S1
S2
S3
S19
S20
. . .
100 TB
100 TB
100 TB
100 TB
100 TB

Hadoop Distributed File System (HDFS)

Adapted from Erik Jonsson (UT Dallas)

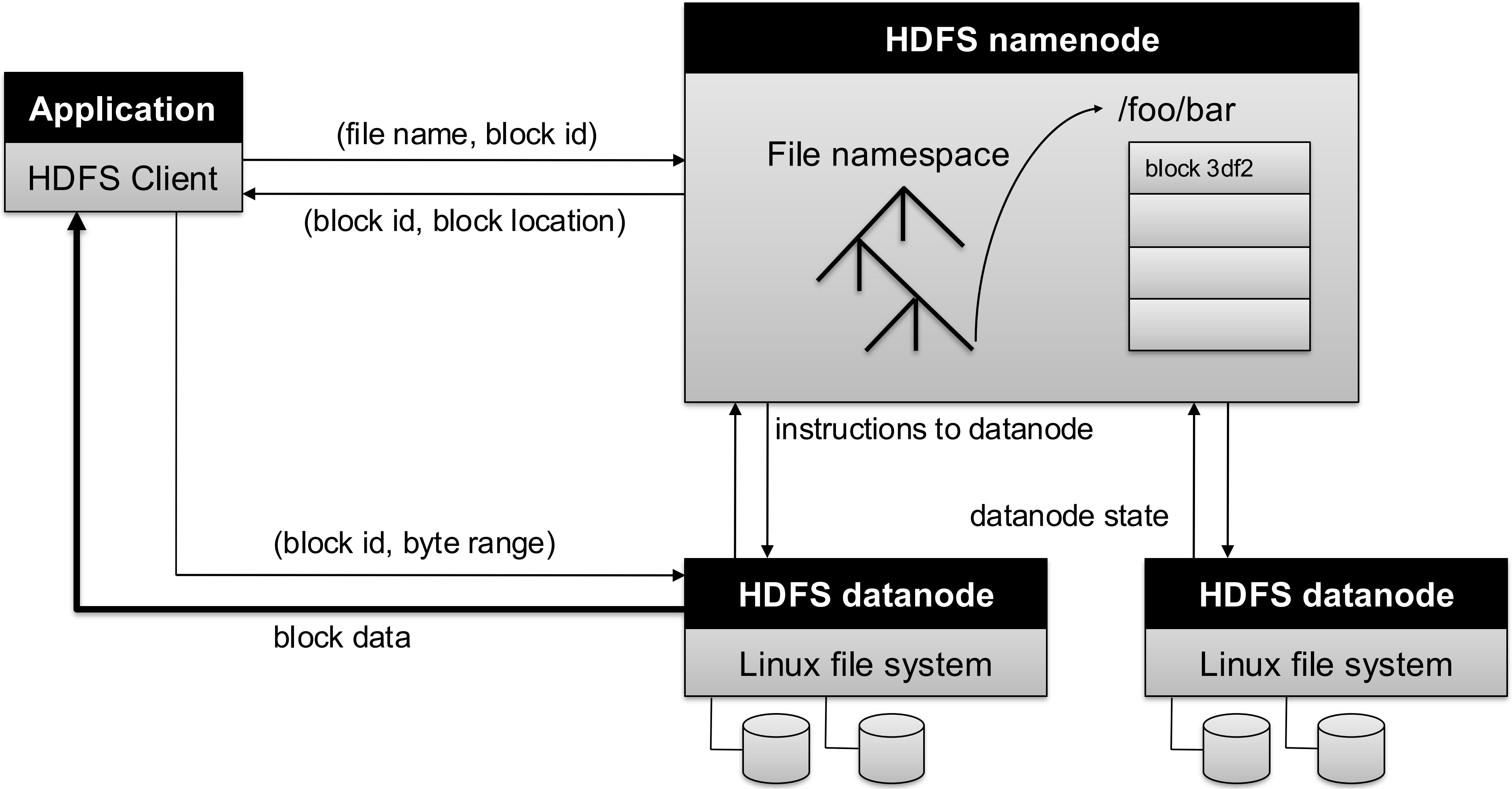
Goals of HDFS

- Very Large Distributed File System
  - 10K nodes, 100 million files, 10PB
- Assumes Commodity Hardware
  - Files are replicated to handle hardware failure
  - Detect failures and recover from them
- Optimized for Batch Processing
  - Provides very high aggregate bandwidth

Distributed File System

- Data Coherency
  - Write-once-read-many access model
  - Client can only append to existing files
- Files are broken up into blocks
  - Typically 64MB block size
  - Each block replicated on multiple DataNodes
- Intelligent Client
  - Client can find location of blocks
  - Client accesses data directly from DataNode

HDFS Architecture



Adapted from (Ghemawat et al., SOSP 2003)

Functions of a NameNode

- Manages File System Namespace
  - Maps a file name to a set of blocks
  - Maps a block to the DataNodes where it resides
- Cluster Configuration Management
- Replication Engine for Blocks

NameNode Metadata

- Metadata in Memory
  - The entire metadata is in main memory
  - No demand paging of metadata
- Types of metadata
  - List of files
  - List of Blocks for each file
  - List of DataNodes for each block
  - File attributes, e.g. creation time, replication factor
- A Transaction Log
  - Records file creations, file deletions etc

DataNode

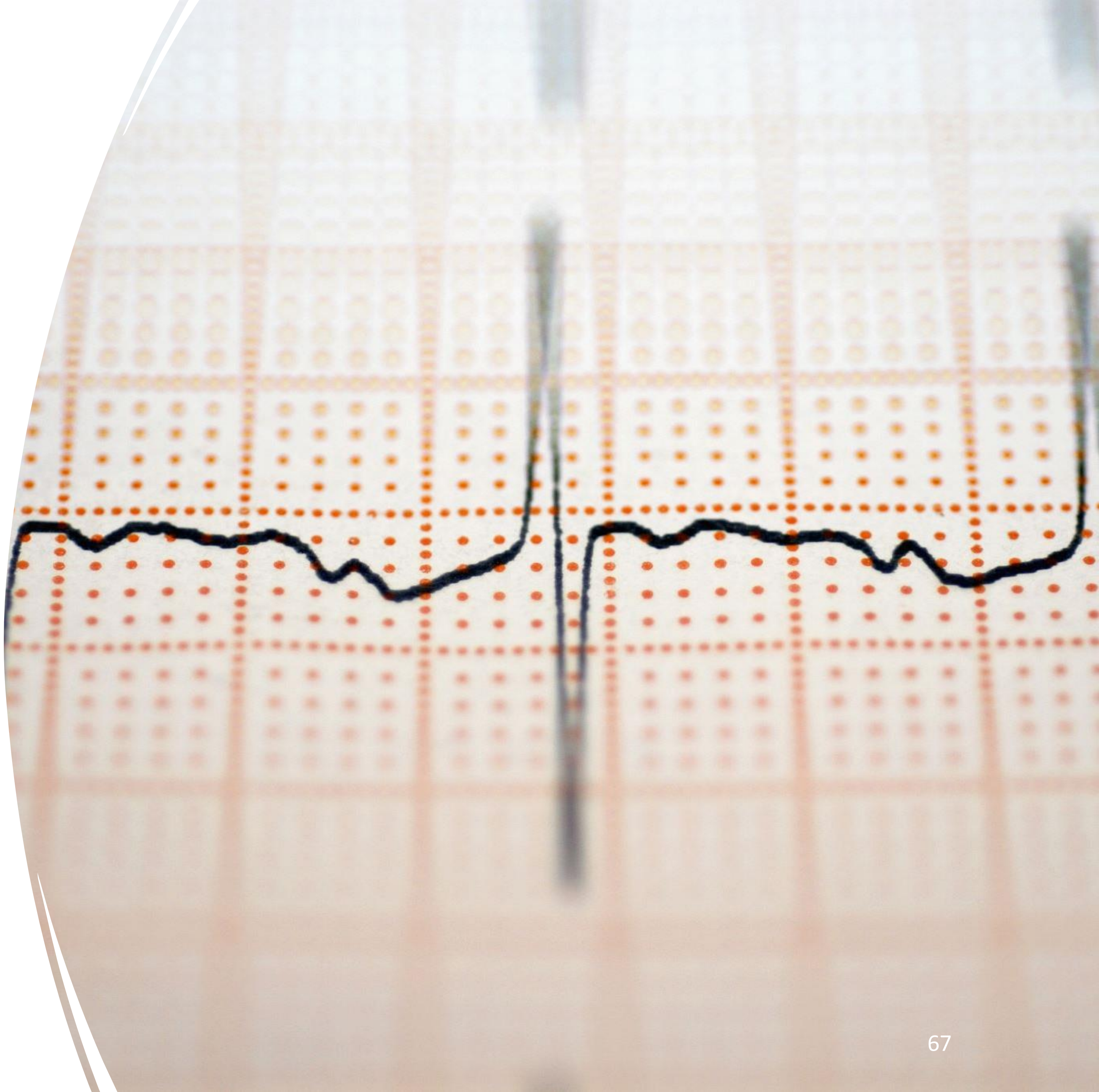
- A Block Server
  - Stores data in the local file system (e.g. ext3)
  - Stores metadata of a block (e.g. CRC)
  - Serves data and metadata to Clients
- Block Report
  - Periodically sends a report of all existing blocks to the NameNode
- Facilitates Pipelining of Data
  - Forwards data to other specified DataNodes

Block Placement Policy

- Current Policy: 3 replicas will be stored on at least 2 racks
  - One replica on local node
  - Second replica on a remote rack
  - Third replica on same remote rack
    - Rebalance **might** later move this to a third rack
- Clients read from nearest replicas

Heartbeats

- DataNodes send heartbeat to the NameNode
  - Once every 3 seconds
- NameNode uses heartbeats to detect DataNode failure



Replication Engine

- NameNode detects DataNode failures
  - Chooses new DataNodes for new replicas
  - Balances disk usage
  - Balances communication traffic to DataNodes

HDFS Demo

- Dan – open PuTTY and show them how to do some stuff?

- Students viewing this on the webpage –
  - Ummm, google “HDFS Demo”, the first one on Google is good I think

Google File System (GFS)

Terminology differences:

GFS master = Hadoop namenode

GFS chunkservers = Hadoop datanodes

Implementation differences:

Different consistency model for file appends

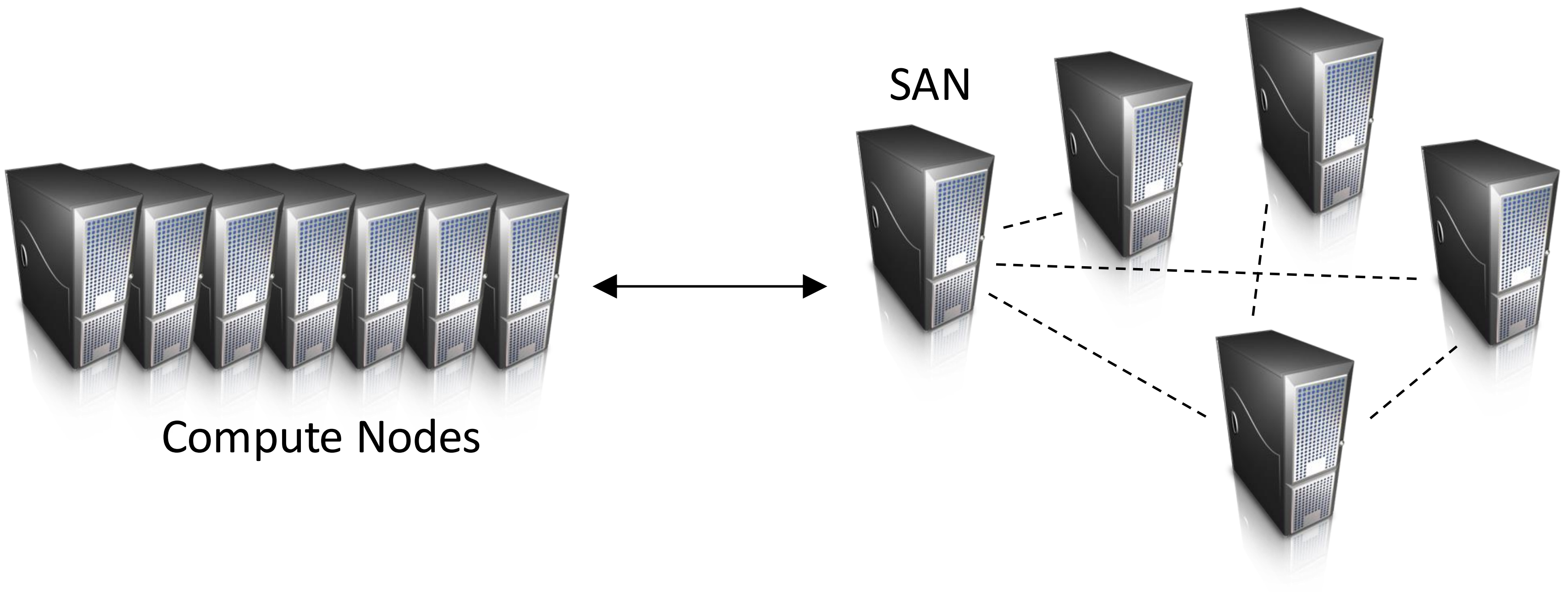
Implementation language

Performance

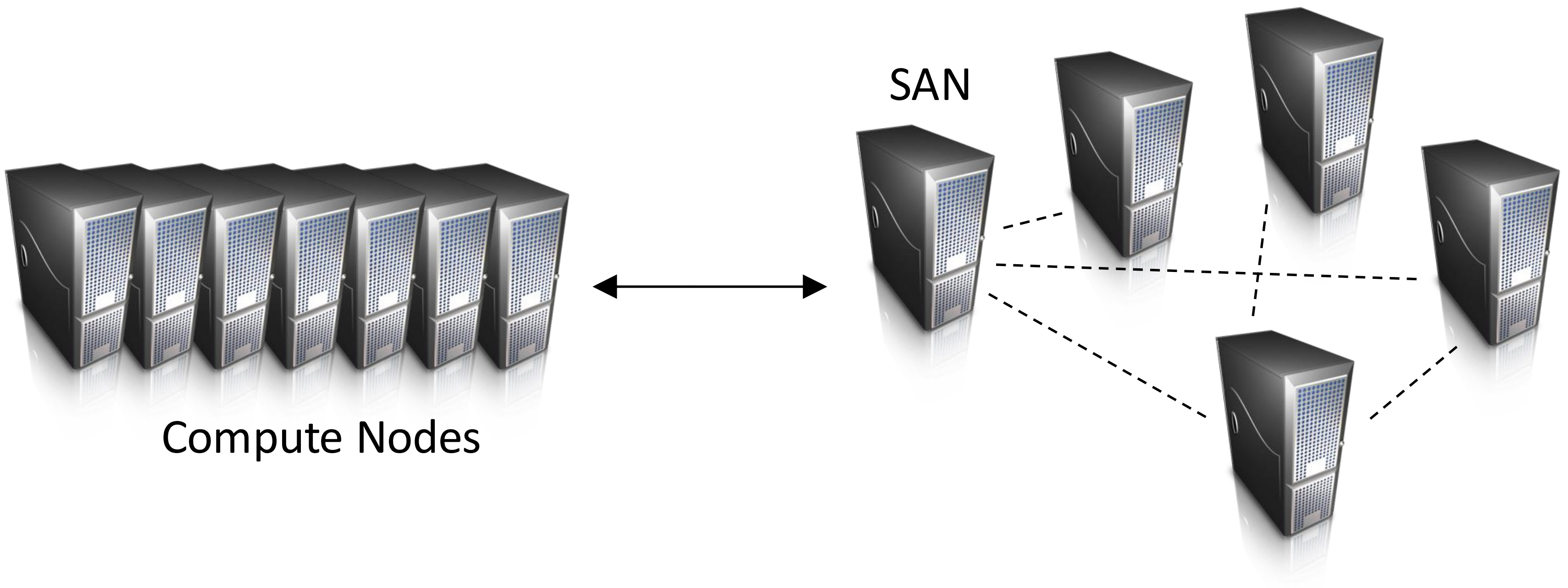
Hadoop Cluster Architecture

How do we get data to the workers?

Let's consider a typical supercomputer…



Compute-Intensive vs. Data-Intensive



Why does this make sense for compute-intensive tasks?

What's the issue for data-intensive tasks?

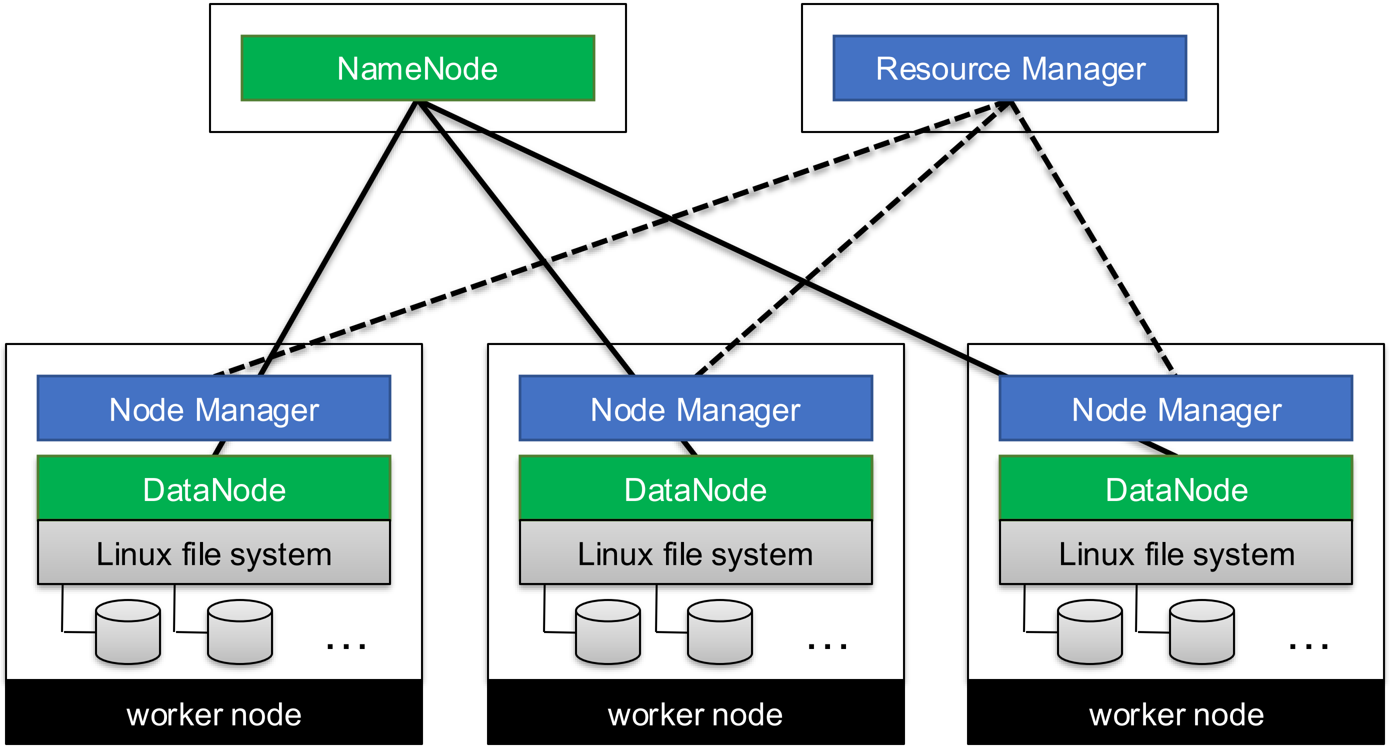
What's the solution?

Don't move data to workers… move workers to the data!

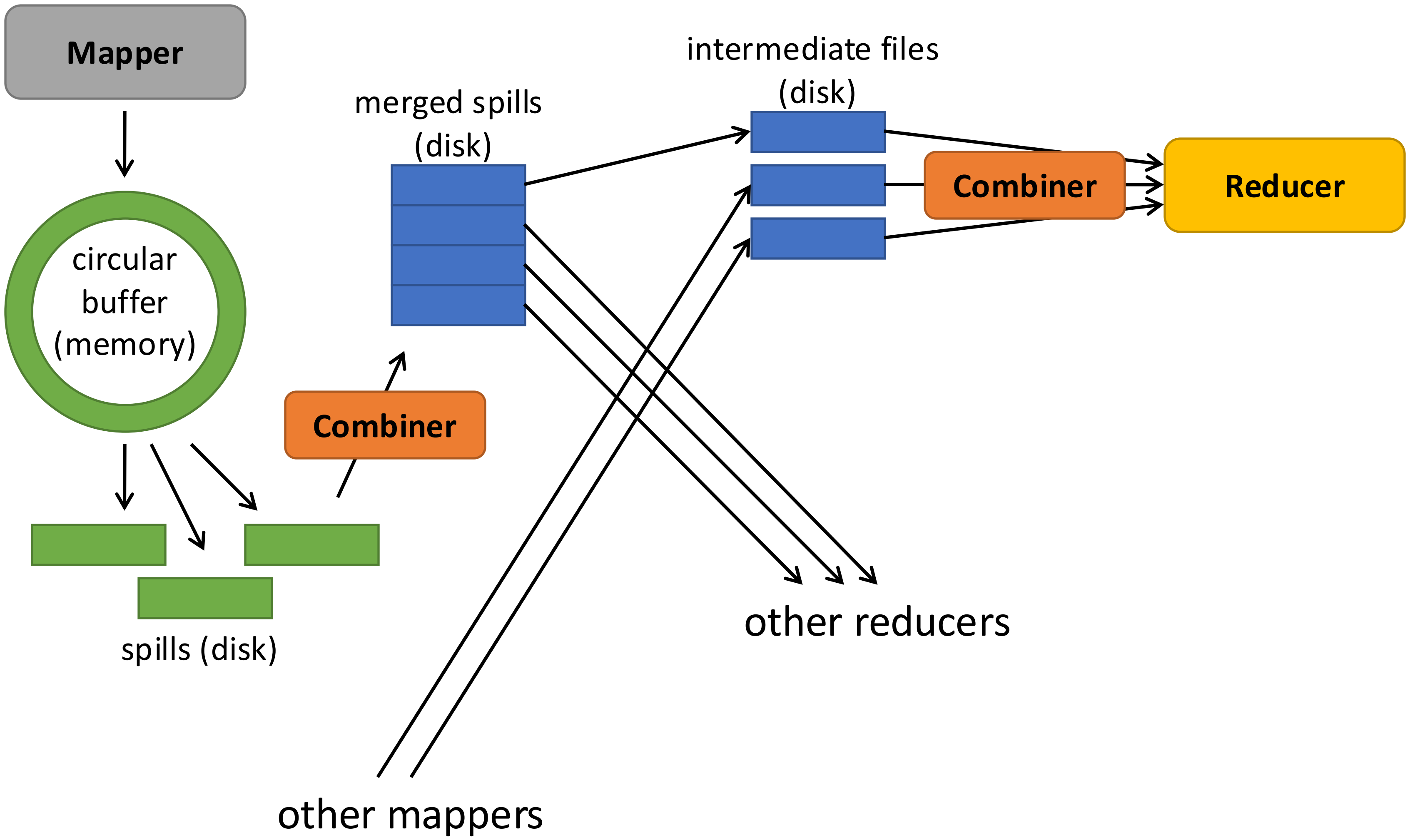
Key idea: co-locate storage and compute

Start up worker on nodes that hold the data

Putting everything together…

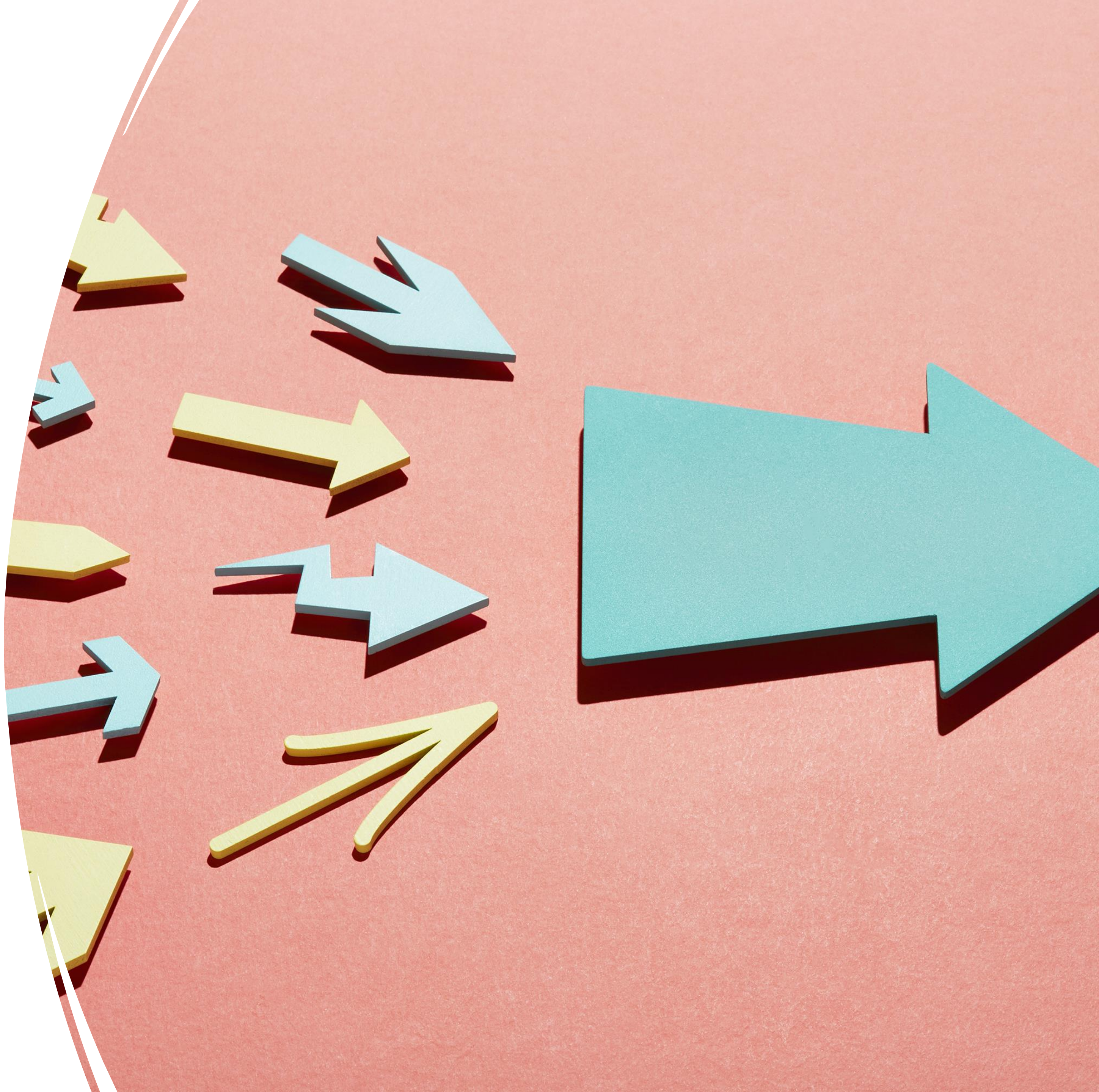


Back to Combiners in MapReduce



Combiner Design

- Combiners are like Reducers – they have the same signature
  - A reducer can have different key types
- Combiners are **optional**
  - May not be run
  - May run once
  - May run many times



Computing the mean

```
def map(key : String, value: Int):
  emit(key, value)

def reduce(key: String, values: List[Int]):
  sum = 0
  count = 0
for value in values:
   sum += value
   count += 1
   emit(key, sum / count)
```

(a, 7)
(a,18)
(c, 4)
(b,1)
(c, 10)
(a, 3)
…

Computing the mean (v2)

```
def map(key : String, value: Int):
  emit(key, value)

def combine(key: String, values: List[Int]):
  for value in values:
    sum += value
    count += 1
   emit(key, (sum, count))

def reduce(key: String, values: List[(Int, Int)]):
  for (v, c) in values:
    sum += v
    count += c
  emit(key, sum / count)
```

INVALID

(a, 7)
(a,18)
(c, 4)
(b,1)
(c, 10)
(a, 3)
…

Computing the mean (v3)

```
def map(key : String, value: Int):
  emit(key, (value, 1))

def combine(key: String, values: List[(Int, Int)]):
  for (v, c) in values:
    sum += v
    count += c
   emit(key, (sum, count))

def reduce(key: String, values: List[(Int, Int)]):
  for (v, c) in values:
    sum += v
    count += c
  emit(key, sum / count)
```

(a, 7)
(a,18)
(c, 4)
(b,1)
(c, 10)
(a, 3)
…

Performance

Input size: 200m integers, 3 unique keys

V1 (baseline)   ~120 seconds

V3 (combiner) ~90 seconds

I wanna go fast

Combiners improve performance by reducing network traffic

Combiners work during file merges.

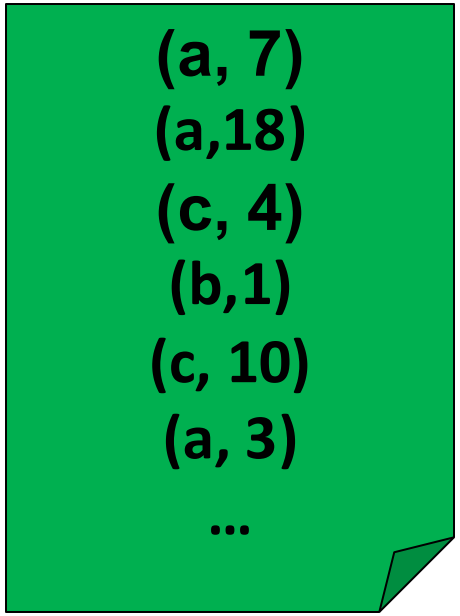
- Local filesystem is faster than network access

But memory is faster than the filesystem

Computing the mean (v4)

```
class mapper:
    def setup(self):
        self.sums = Map()
        self.counts = Map()
    def map(self, key, value):
        self.sums[key] += value
        self.counts[key] += 1
    def cleanup(self):
        for (key, count) in counts:
        emit(key, (sums[key], count))
```

Didn't you say not to do this???



In-Mapper Combine

Preserve state across calls to map

Advantage: Speed

Disadvantage: Requires memory management

Performance

Input size: 200m integers, 3 unique keys

V1 (baseline)	~120 seconds
V3 (combiner)	~90 seconds
V4 (IMC)	~60 seconds

Discussion: Can we do this for word frequency?

```
class mapper:
  def setup(self):
    counts = HashMap()
  def map(self, key: Long, value: String):
    for word in tokenize(value):
      counts[word] += 1
  def map_cleanup():
    for (key, count) in counts:
      emit(key, count)
```

New Problem: Term Co-Occurrence

M_{ij}: number of times word i and word j coöccur in some context

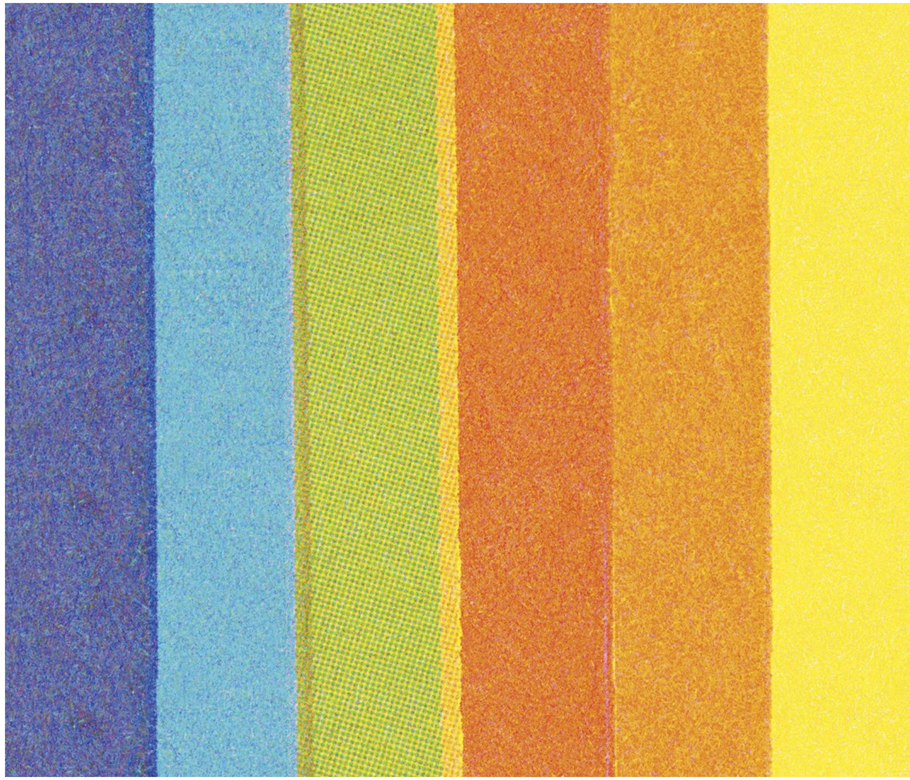
E.g. how many times is i followed immediately b j in a sentence

M is N x N, where N is the vocabulary

Two Approaches

Pairs

Stripes



Pairs

Mapper

Input: Sentence

Output: ((a, b), 1), for all pairs of words a, b in the sentence.

Reducer

Input: pair of words, list of counts

Output: Pair of words, count

Pairs, In Pseudocode

```
def map(key : Long, value: String):
  for u in tokenize(value):
    for each v that coöccurs with u in value:
      emit((u, v), 1)

def reduce(key: (String, String), values: List[Int]):
   for value in values:
     sum += value
   emit(key, sum)
```

Pairs Analysis

- Easy to implement
- Easy to understand
- That’s a lot of pairs!
- Combiner won’t do much. Why?

Stripes

Mapper

Input: Sentence

Output: $(a, \{b_1:c_1, b_2:c_2, \ldots, b_m:c_m\})$, where:

a is a word from the input

$b_1 \ldots b_m$ are all words that coöccur with a

c_i is the number of times (a, b_i) coöccur

{} means a map (aka a dictionary, associative array, etc)

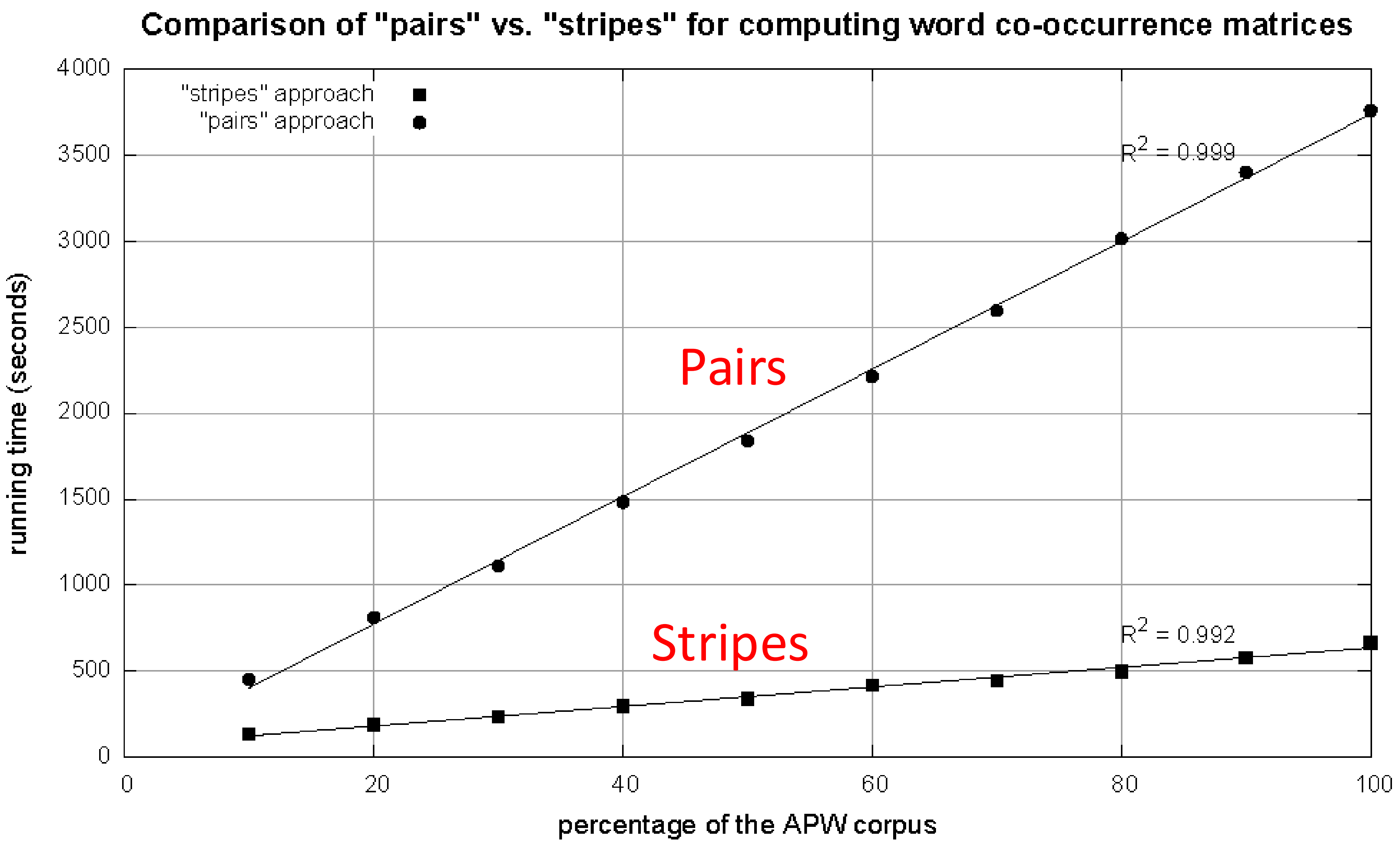
Stripes, Pseudocode

```
def map(key: Long, value: String):
   for u in tokenize(value)
     counts = {}
     for each v that coöcurs with u in value:
       counts(v) += 1
     emit(u, counts)

def reduce(key: Long, values: List[Map[String->Int]]):
  for value in values:
    sum += value
  emit(key, sum)
```

Stripes Analysis

- Fewer key-value pairs to send
- Combiners will do more work
- Map is a heavier object than a single Int
- More computationally intensive
- Will the map fit in memory???



Cluster size: 38 cores
Data Source: Associated Press Worldstream (APW) of the English Gigaword Corpus (v3), which contains 2.27 million documents (1.8 GB compressed, 5.7 GB uncompressed)

So Always Use Stripes?

No. There's a tradeoff.

"Easier to understand and implement" is NOT bad.

You'll see after A1, mwhahaha. (For CS431 this only hits you on A2, don't get complacent)

For English words and normal sentence lengths, the stripe fits in memory easily. It won't always work out that way.

Another Problem, Relative Frequencies

$$f(B|A) = \frac{N(A,B)}{N(A,*)}$$

Where N(A, B) is number of coöccurrences of A and B, and N(A,*) is the sum of N(A,x) over all x

Why do we want to do this?

How do we make it fit into MapReduce?

Stripes

A -> {B_1:C_1, B_2,C_2, ….}

Easy-Peasy. If N(A, B) = N(B, A) then N(A,*) is just C_1+C_2+…

The stripe gives us all the information we need!

Pairs?

```
def reduce(key:Pair[String], values: List[Int]):
   let (a, b) = key
   for v in values:
     sum += v
   emit((b, a), sum / freq(a))
```

Hmmm, what’s freq(a)? We don’t know that until we’ve processed all keys of the form (a, *)

f(B|A): "Pairs"

(a, *) → 32

Reducer holds this value in memory

(a, b_1) → 3
(a, b_2) → 12
(a, b_3) → 7
(a, b_4) → 1
…

(a, b_1) → 3 / 32
(a, b_2) → 12 / 32
(a, b_3) → 7 / 32
(a, b_4) → 1 / 32
…

For this to work:

- Emit extra (a, *) for every b_n in mapper
- Make sure all a's get sent to same reducer (use partitioner)
- Make sure (a, *) comes first (define sort order)
- Hold state in reducer across different key-value pairs

Pairs, Mapper and Partitioner

```
def map(key: Long, value: String):
   for u in tokenize(value):
     for v in cooccurrence(u):
        emit((u, v), 1)
        emit((u, “*”), 1)

def partition(key: Pair, value: Int, N: Int):
   return hash(key.left) % N
```

Pairs, Mapper and Partitioner (improved)

```
def map(key: Long, value: String):
   for u in tokenize(value):
      for v in cooccurrence(u):
         emit((u, v), 1)
      emit((u, “*”), len(cooccurrence(u))

def partition(key: Pair, value: Int, N: Int):
   return hash(key.left) % N
```

Pairs, Reducer

Stats term

```
marginal = 0

def reduce(key: Pair, values: List[Int]):
   let (a, b) = key
   for (v in values):
     sum += v
   if (b == “*”):
      marginal = sum
   else:
     emit((b, a), sum / marginal)
```

Pairwise Mutual Information (PMI)

On the assignment, you're doing something SIMILAR

$$PMI(x,y) = \log \frac{p(x,y)}{p(x)p(y)}$$

This requires* two passes

On the assignment "cooccur" means "both occur on the same line"

* It doesn't PER SE but it's way more trouble than it's worth

PMI, Yeah, What's It Good For?

~~Absolutely Nothing!~~

PMI is useful for establishing "semantic distance" between tokens

Tokens with similar lists of cooccurrences sorted by PMI likely have similar meaning.

Sweet, Delicious Hints

A1 suggests multiple passes as something you might want to consider.

CONSIDER IT STRONGLY

(In other words, it’s possible to do with a single pass but there’s no gain to doing this. This is not a challenge)