Search Engine Architecture

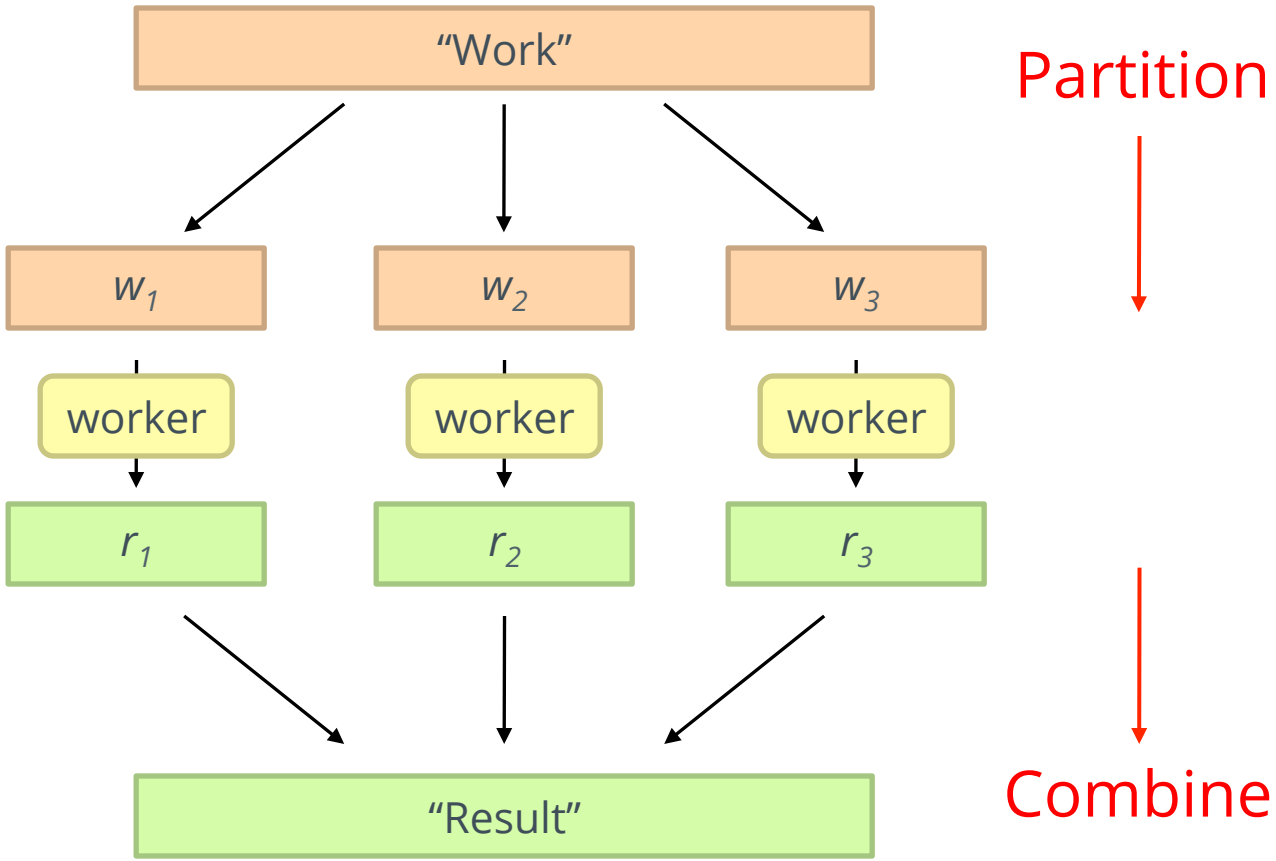
5. Big Data Processing

Noted slides adapted from: Lin et al.'s Big Data Infrastructure, UMD Spring 2015 with cosmetic changes.

Overview

- Parallel computation is hard
  - Because of synchronization
- MapReduce offers a solution
  - (for some applications)
  - Simple, highly-constrained API
    - Sometimes requires creativity to work with
  - No side effects
- Batch index construction can be a good fit

Divide and Conquer



Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Parallelization Challenges

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers die?

What's the common theme of all of these problems?

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Common Theme?

- Parallelization problems arise from:
  - Communication between workers (e.g., to exchange state)
  - Access to shared resources (e.g., data)
- Thus, we need a synchronization mechanism

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

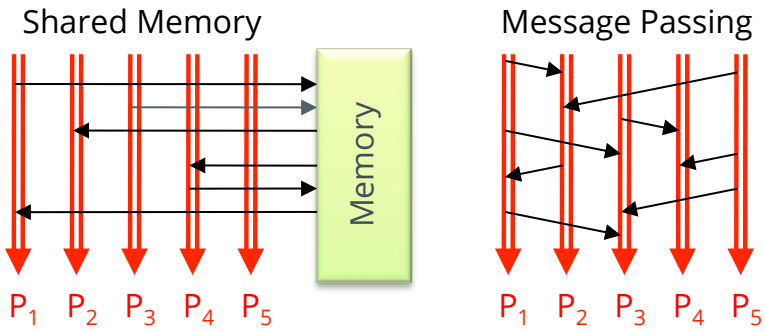
Managing Multiple Workers

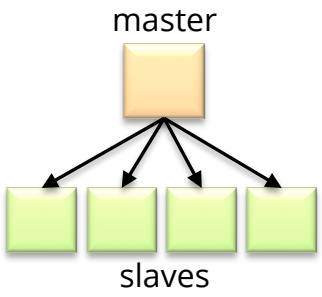
- Difficult because
  - We don't know the order in which workers run
  - We don't know when workers interrupt each other
  - We don't know when workers need to communicate partial results
  - We don't know the order in which workers access shared data
- Thus, we need:
  - Semaphores (lock, unlock)
  - Conditional variables (wait, notify, broadcast)
  - Barriers
- Still, lots of problems:
  - Deadlock, livelock, race conditions...
  - Dining philosophers, sleeping barbers, cigarette smokers...
- Moral of the story: be careful!

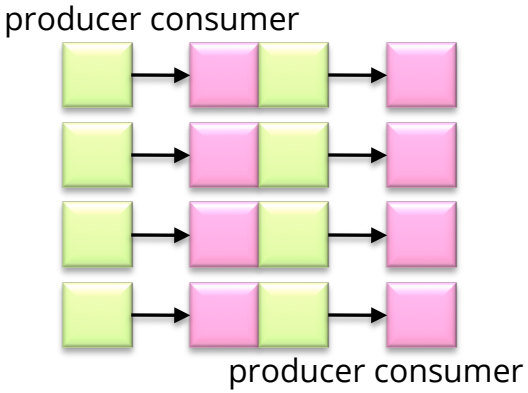
Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

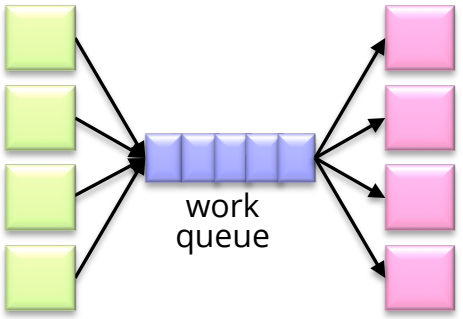
Traditional Tools

- Programming models
  - Shared memory (pthreads)
  - Message passing (MPI)
- Design Patterns
  - Master-slaves
  - Producer-consumer flows
  - Shared work queues







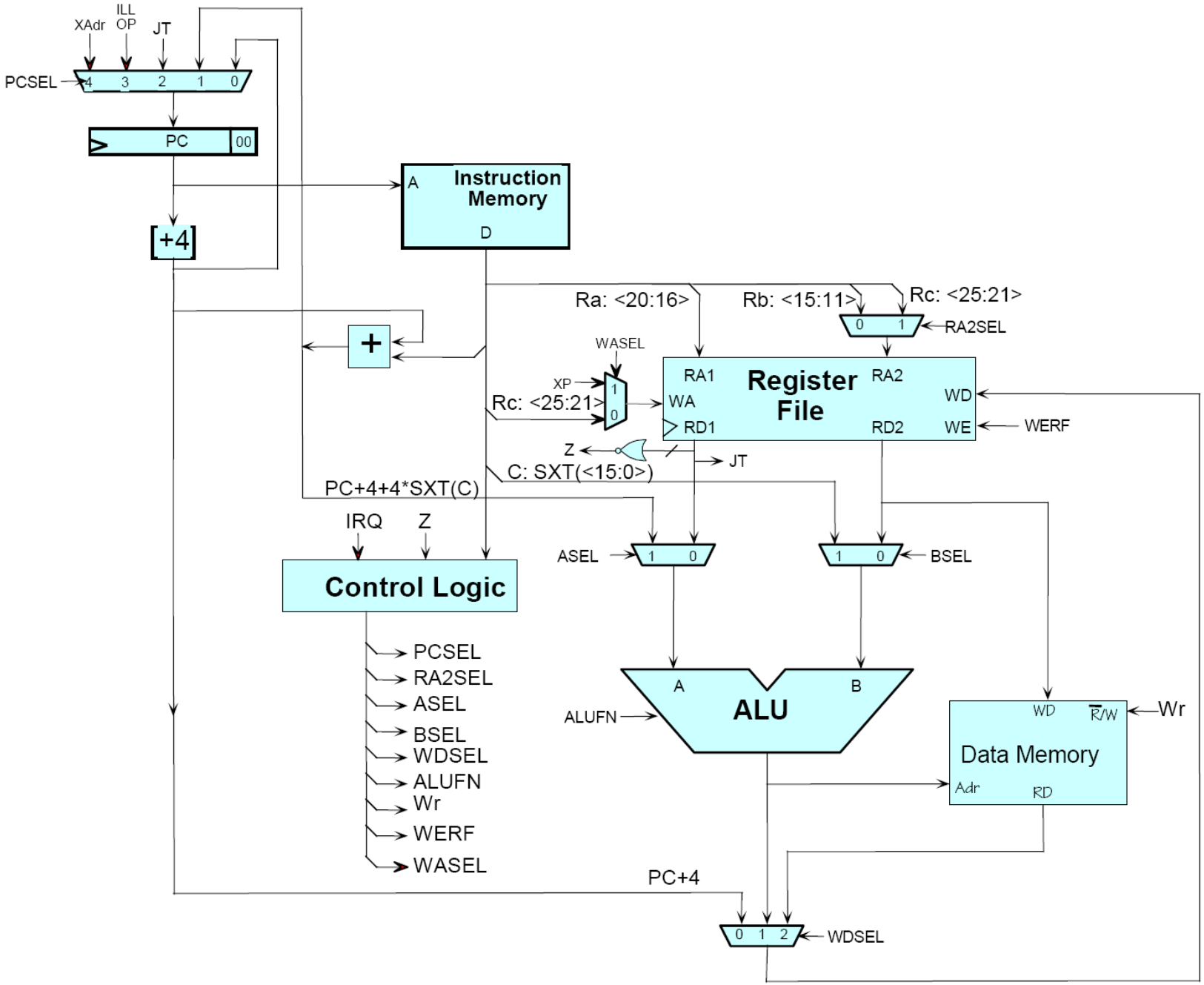


Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

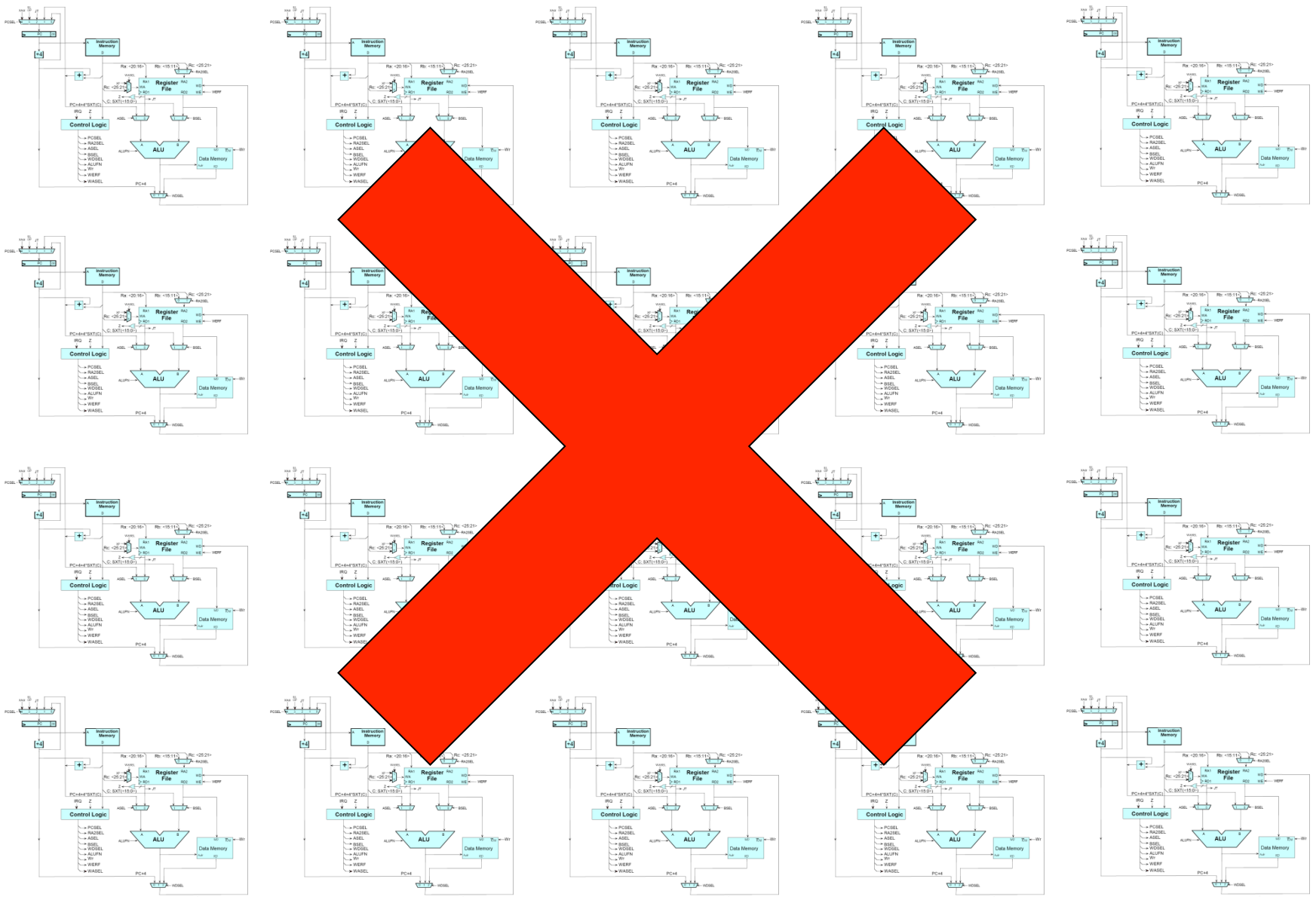
Where the rubber meets the road

- Concurrency is difficult to reason about
- Concurrency is even more difficult to reason about …
  - At the scale of datacenters and across datacenters
  - In the presence of failures
  - In terms of multiple interacting services
- Not to mention debugging
- The reality:
  - Lots of one-off solutions, custom code
  - Write you own dedicated library, then program with it
  - Burden on the programmer to explicitly manage everything

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.



Source: MIT Open Courseware



Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

What's the point?

- It's all about the right level of abstraction
  - Moving beyond the von Neumann architecture
  - We need better programming models
- Hide system-level details from the developers
  - No more race conditions, lock contention, etc.
- Separating the *what* from *how*
  - Developer specifies the computation that needs to be performed
  - Execution framework ("runtime") handles actual execution

The datacenter *is* the computer!

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

“Big Ideas”

- Scale “out”, not “up”
  - Limits of SMP and large shared-memory machines
- Move processing to the data
  - Cluster has limited bandwidth
- Process data sequentially, avoid random access
  - Seeks are expensive, disk throughput is reasonable
- Seamless scalability
  - From the mythical man-month to the tradable machine-hour

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Source: Lin et al. Big Data Infrastructure, UMD Course 2[illegible]

Typical Big Data Problem

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

Map

Reduce

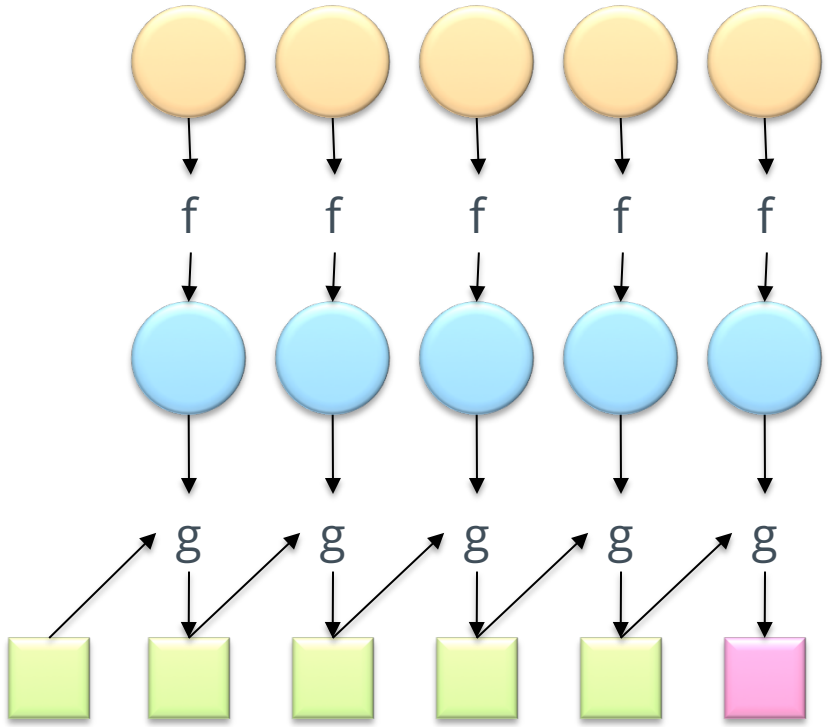
Key idea: provide a functional abstraction for these two operations

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Roots in Functional Programming

Map

Fold



Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

MapReduce

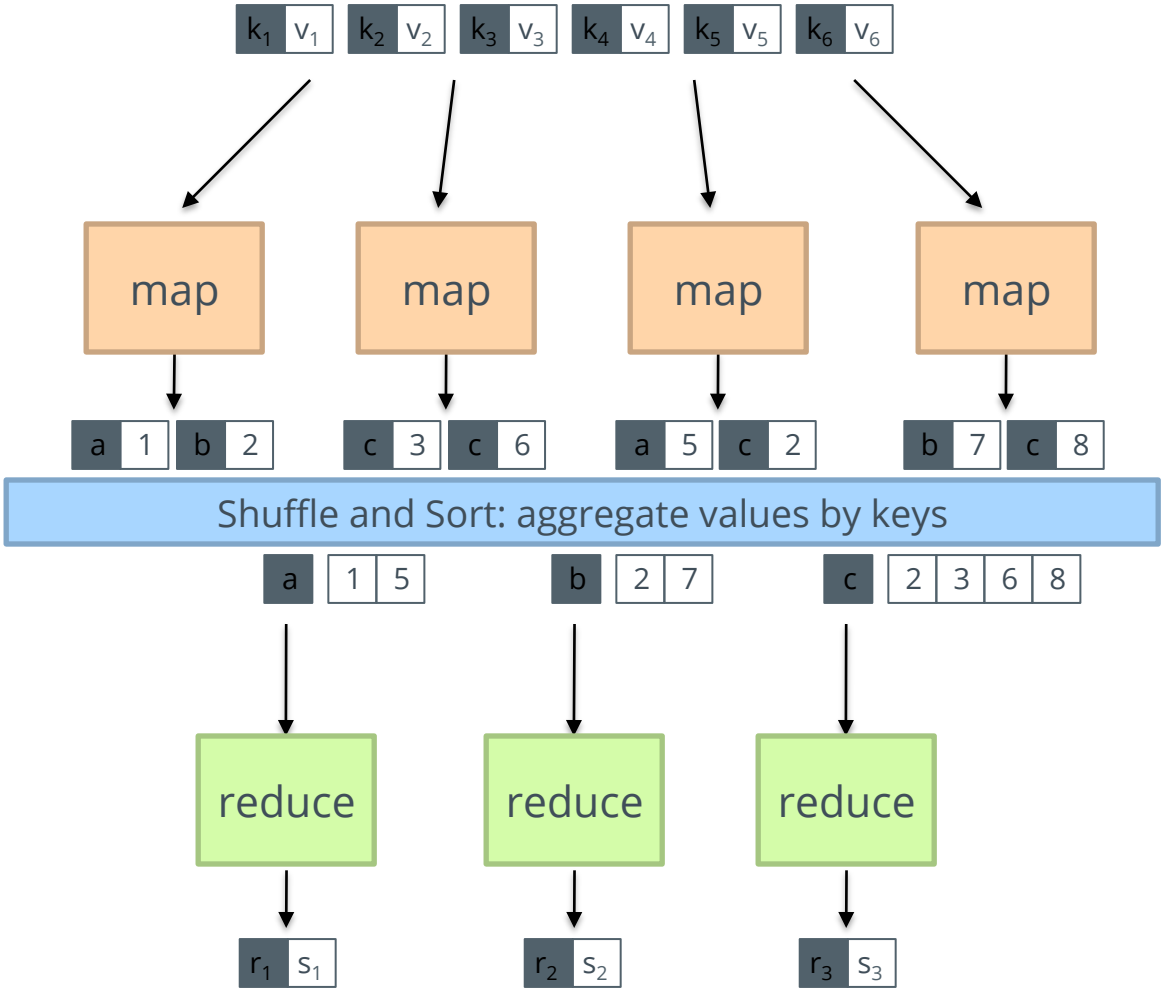
- Programmers specify two functions:

  map $(k_1, v_1) \rightarrow [<k_2, v_2>]$

  reduce $(k_2, [v_2]) \rightarrow [<k_3, v_3>]$

  - All values with the same key are sent to the same reducer
- The execution framework handles everything else

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.



Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

MapReduce “Runtime”

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles “data distribution”
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of distributed FS

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

MapReduce

- Programmers specify two functions:

  map (k, v) → <k′, v′>*

  reduce (k′, v′) → <k′, v′>*

  - All values with the same key are reduced together
- The execution framework handles everything else…
- (Not quite.) Usually, programmers also specify:

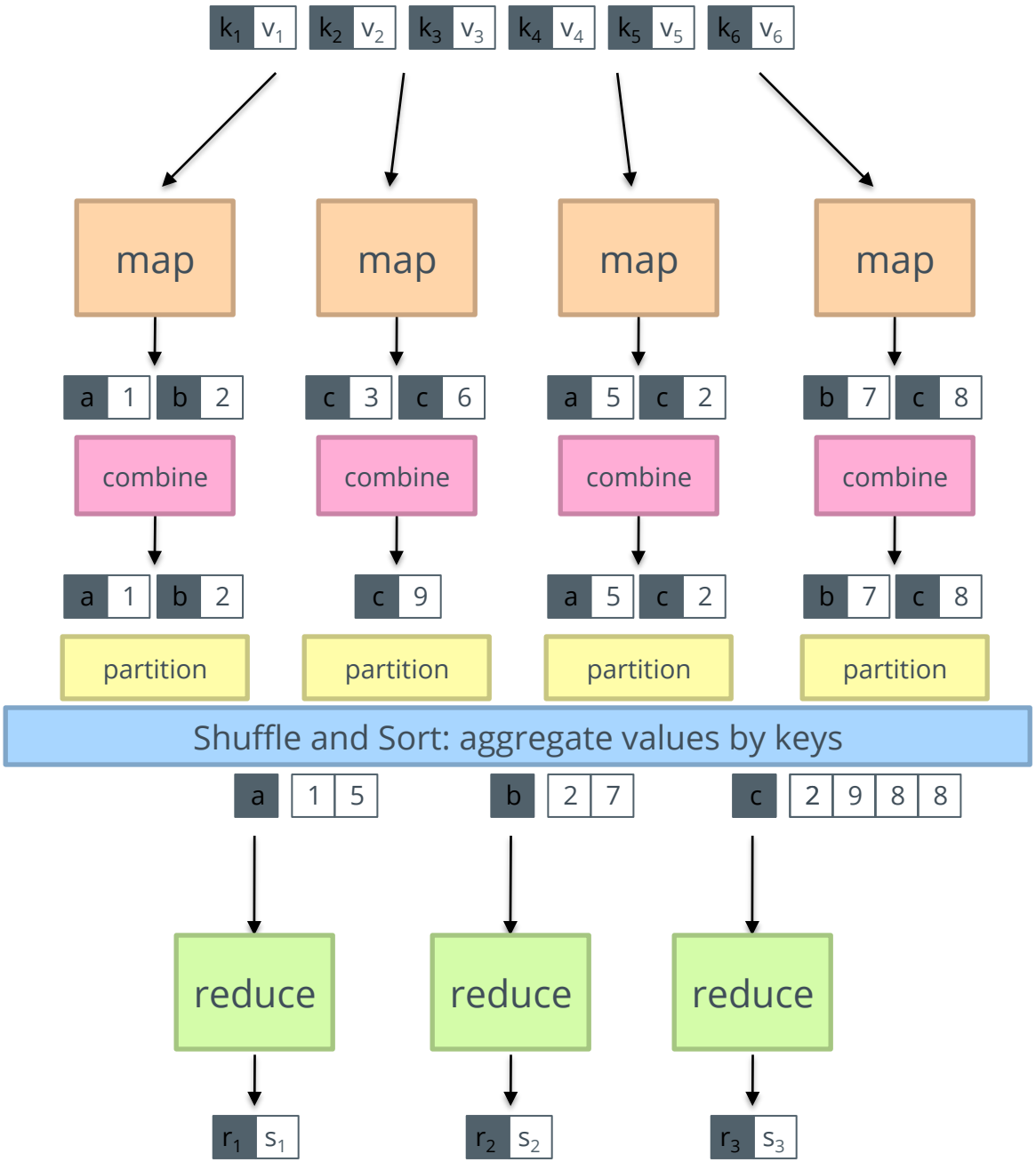
  partition (k′, number of partitions) → partition for k′

  - Often a simple hash of the key, e.g., hash(k′) mod n
  - Divides up key space for parallel reduce operations

  combine (k′, v′) → <k′, v′>*

  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.



Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Two more details…

- Barrier between map and reduce phases
  - But we can begin copying intermediate data earlier
- Keys arrive at each reducer in sorted order
  - No enforced ordering *across* reducers

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

“Hello World”: Word Count

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)

1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count s)
```

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

MapReduce can refer to…

- The programming model
- The execution framework (aka “runtime”)
- The specific implementation

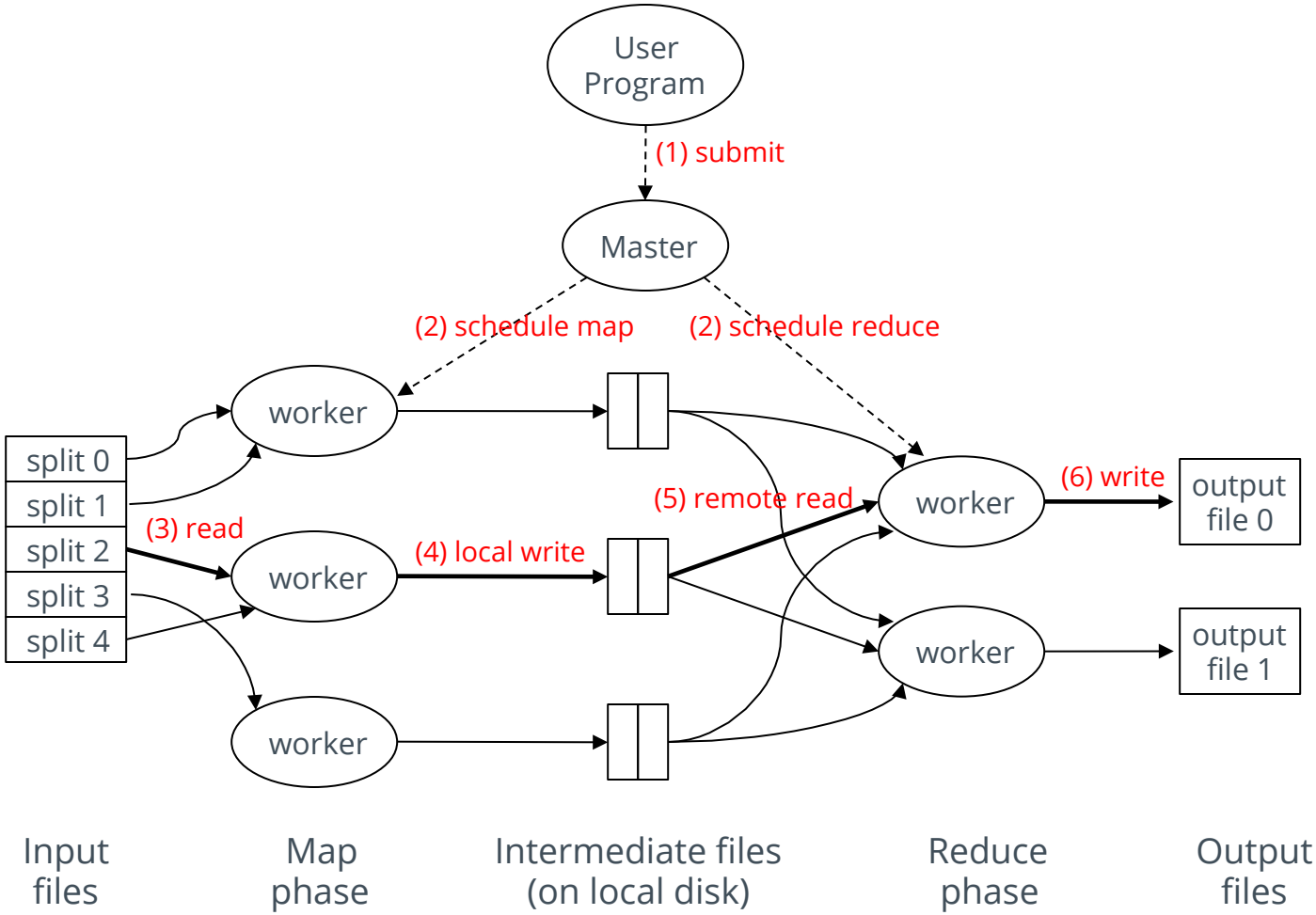
Usage is usually clear from context!

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

MapReduce Implementations

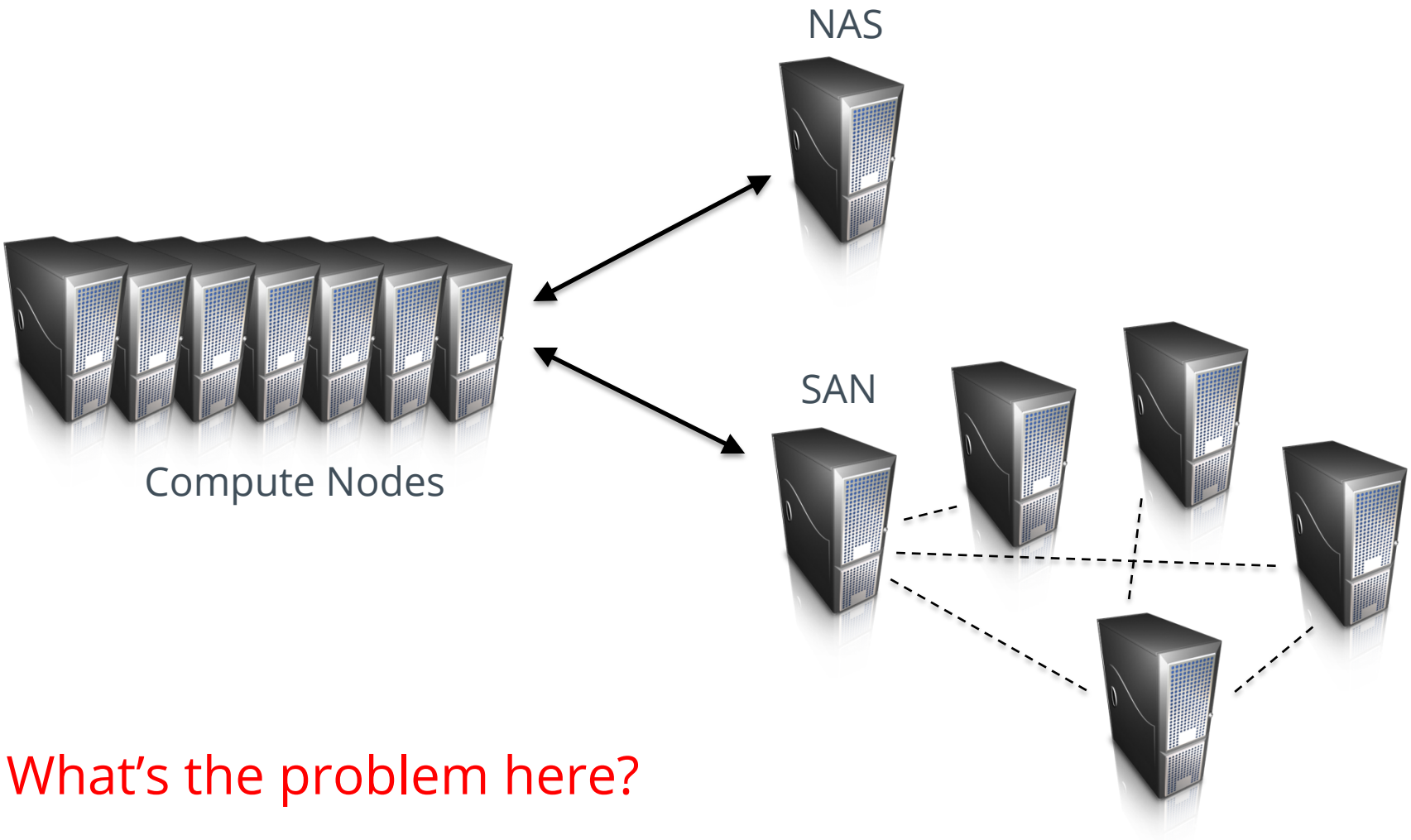
- Google has a proprietary implementation in C++
  - Bindings in Java, Python
- Hadoop is an open-source implementation in Java
  - Development led by Yahoo, now an Apache project
  - Used in production at Yahoo, Facebook, Twitter, LinkedIn, Netflix, …
  - The *de facto* big data processing platform
  - Rapidly expanding software ecosystem
- Lots of custom research implementations
  - For GPUs, cell processors, etc.

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.



Adapted from (Dean and Ghemawat, OSDI 2004) via Lin et al. Big Data Infrastructure, UMD Spring 2015.

How do we get data to the workers?



What's the problem here?

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Distributed File System

- Don’t move data to workers… move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- Why?
  - Not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable
- A distributed file system is the answer
  - GFS (Google File System) for Google’s MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

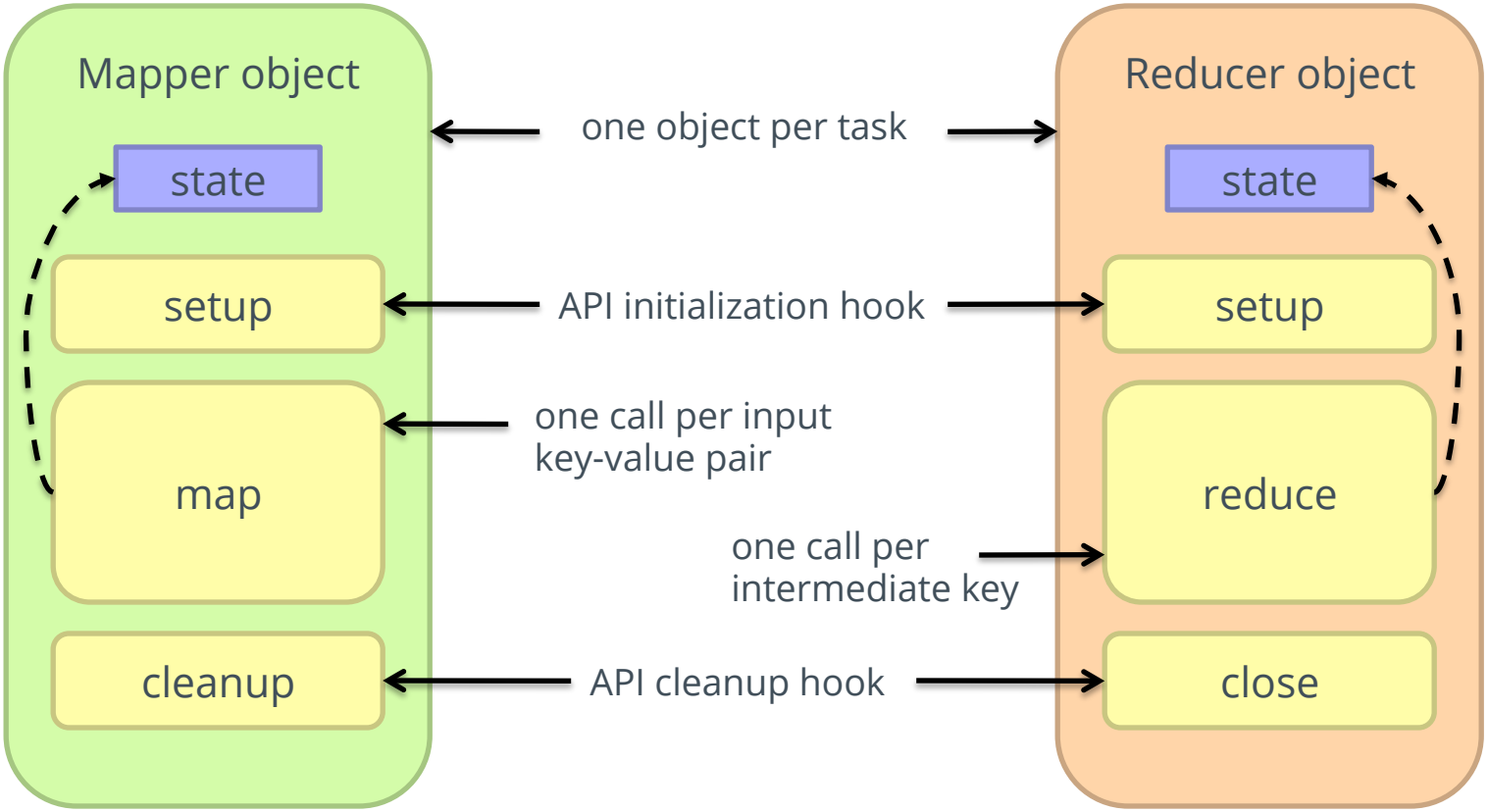
Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Tools for Synchronization

- Cleverly-constructed data structures
  - Bring partial results together
- Sort order of intermediate keys
  - Control order in which reducers process keys
- Partitioner
  - Control which reducer processes which keys
- Preserving state in mappers and reducers
  - Capture dependencies across multiple keys and values

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Preserving State



Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Importance of Local Aggregation

- Ideal scaling characteristics:
  - Twice the data, twice the running time
  - Twice the resources, half the running time
- Why can't we achieve this?
  - Synchronization requires communication
  - Communication kills performance
- Thus… avoid communication!
  - Reduce intermediate data via local aggregation
  - Combiners can help

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Word Count: Baseline

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)

1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count s)
```

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Word Count: Version 1

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1                ▷ Tally counts for entire document
6:         for all term t ∈ H do
7:             Emit(term t, count H{t})
```

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Word Count: Version 2

```
1: class Mapper
2:     method Initialize
3:         H ← new AssociativeArray
4:     method Map(docid a, doc d)
5:         for all term t ∈ doc d do
6:             H{t} ← H{t} + 1                ▷ Tally counts across documents
7:     method Close
8:         for all term t ∈ H do
9:             Emit(term t, count H{t})
```

Key idea: preserve state across input key-value pairs!

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Design Pattern for Local Aggregation

- "In-mapper combining"
  - Fold the functionality of the combiner into the mapper by preserving state across multiple map calls
- Advantages
  - Speed
  - Why is this faster than actual combiners?
- Disadvantages
  - Explicit memory management required
  - Potential for order-dependent bugs

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Algorithm Design: Example

- Term co-occurrence matrix for a text collection
  - M = N x N matrix (N = vocabulary size)
  - M_{ij}: number of times *i* and *j* co-occur in some context (for concreteness, let's say context = sentence)
- Why?
  - Distributional profiles as a way of measuring semantic distance
  - Semantic distance useful for many language processing tasks

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

MapReduce: Large Counting Problems

- Term co-occurrence matrix for a text collection
  = specific instance of a large counting problem
  - A large event space (number of terms)
  - A large number of observations (the collection itself)
  - Goal: keep track of interesting statistics about the events
- Basic approach
  - Mappers generate partial counts
  - Reducers aggregate partial counts

How do we aggregate partial counts efficiently?

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

First Try: “Pairs”

- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For all pairs, emit (a, b) → count
- Reducers sum up counts associated with these pairs
- Use combiners!

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Pairs: Pseudo-Code

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             for all term u ∈ Neighbors(w) do
5:                 Emit(pair (w, u), count 1)        ▷ Emit count for each co-occurrence

1: class Reducer
2:     method Reduce(pair p, counts [c1, c2, ...])
3:         s ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             s ← s + c                             ▷ Sum co-occurrence counts
6:         Emit(pair p, count s)
```

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

“Pairs” Analysis

- Advantages
  - Easy to implement, easy to understand
- Disadvantages
  - Lots of pairs to sort and shuffle around (upper bound?)
  - Not many opportunities for combiners to work

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Another Try: "Stripes"

- Idea: group together pairs into an associative array

(a, b) → 1
(a, c) → 2
(a, d) → 5
(a, e) → 3
(a, f) → 2

a → { b: 1, c: 2, d: 5, e: 3, f: 2 }

- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For each term, emit a → { b: $count_b$, c: $count_c$, d: $count_d$ … }
- Reducers perform element-wise sum of associative arrays

```
    a → { b: 1,       d: 5, e: 3 }
+   a → { b: 1, c: 2, d: 2,      f: 2 }
    ------------------------------------
    a → { b: 2, c: 2, d: 7, e: 3, f: 2 }
```

Key idea: cleverly-constructed data structure brings together partial results

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Stripes: Pseudo-Code

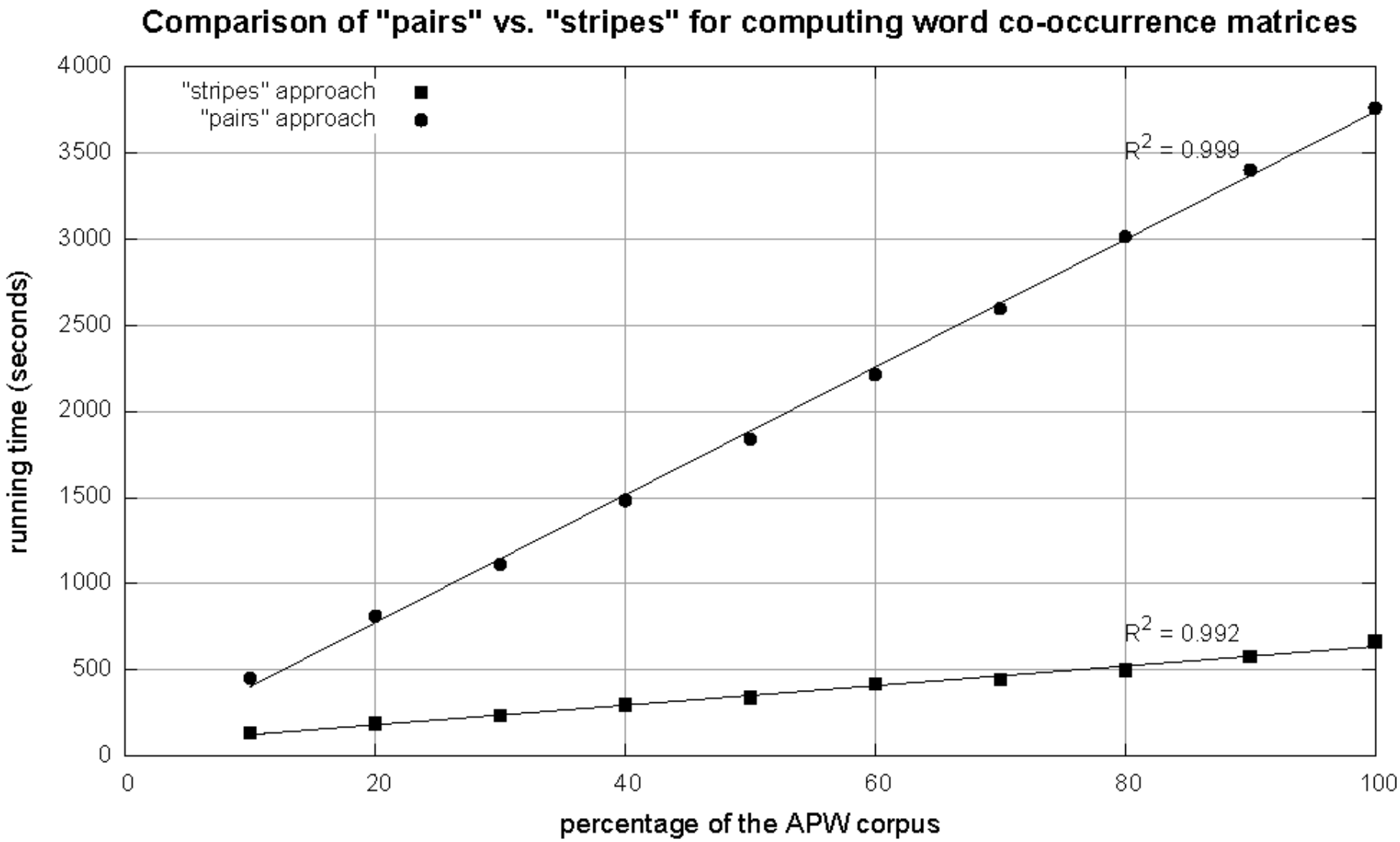
```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             H ← new AssociativeArray
5:             for all term u ∈ Neighbors(w) do
6:                 H{u} ← H{u} + 1                ▷ Tally words co-occurring with w
7:             Emit(Term w, Stripe H)

1: class Reducer
2:     method Reduce(term w, stripes [H_1, H_2, H_3, ...])
3:         H_f ← new AssociativeArray
4:         for all stripe H ∈ stripes [H_1, H_2, H_3, ...] do
5:             Sum(H_f, H)                        ▷ Element-wise sum
6:         Emit(term w, stripe H_f)
```

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

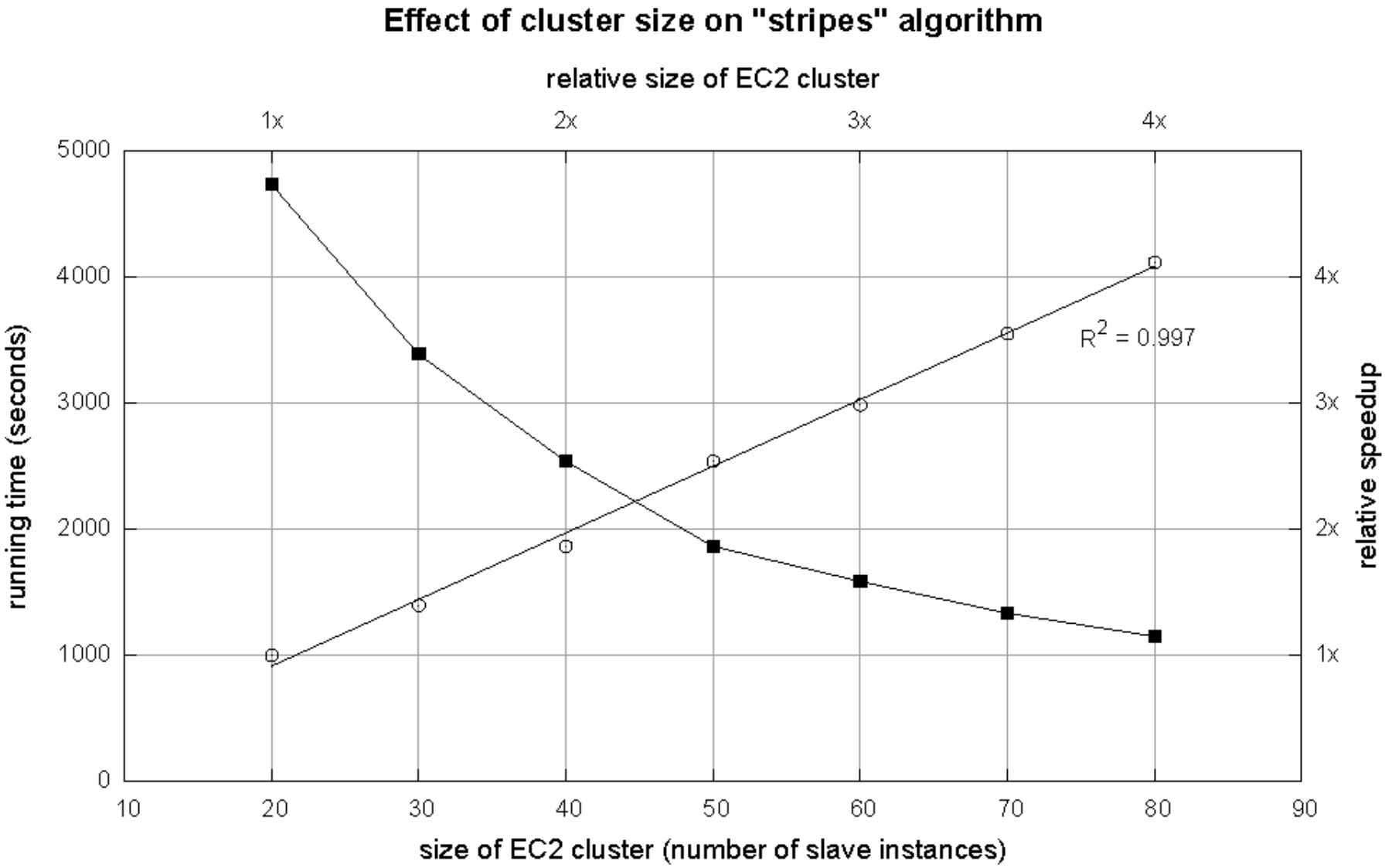
“Stripes” Analysis

- Advantages
  - Far less sorting and shuffling of key-value pairs
  - Can make better use of combiners
- Disadvantages
  - More difficult to implement
  - Underlying object more heavyweight
  - Fundamental limitation in terms of size of event space

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.



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)
Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.



Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Debugging at Scale

- Works on small datasets, won't scale… why?
  - Memory management issues (buffering and object creation)
  - Too much intermediate data
  - Mangled input records
- Real-world data is messy!
  - There's no such thing as "consistent data"
  - Watch out for corner cases
  - Isolate unexpected behavior, bring local

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Demo

Index Construction with MapReduce

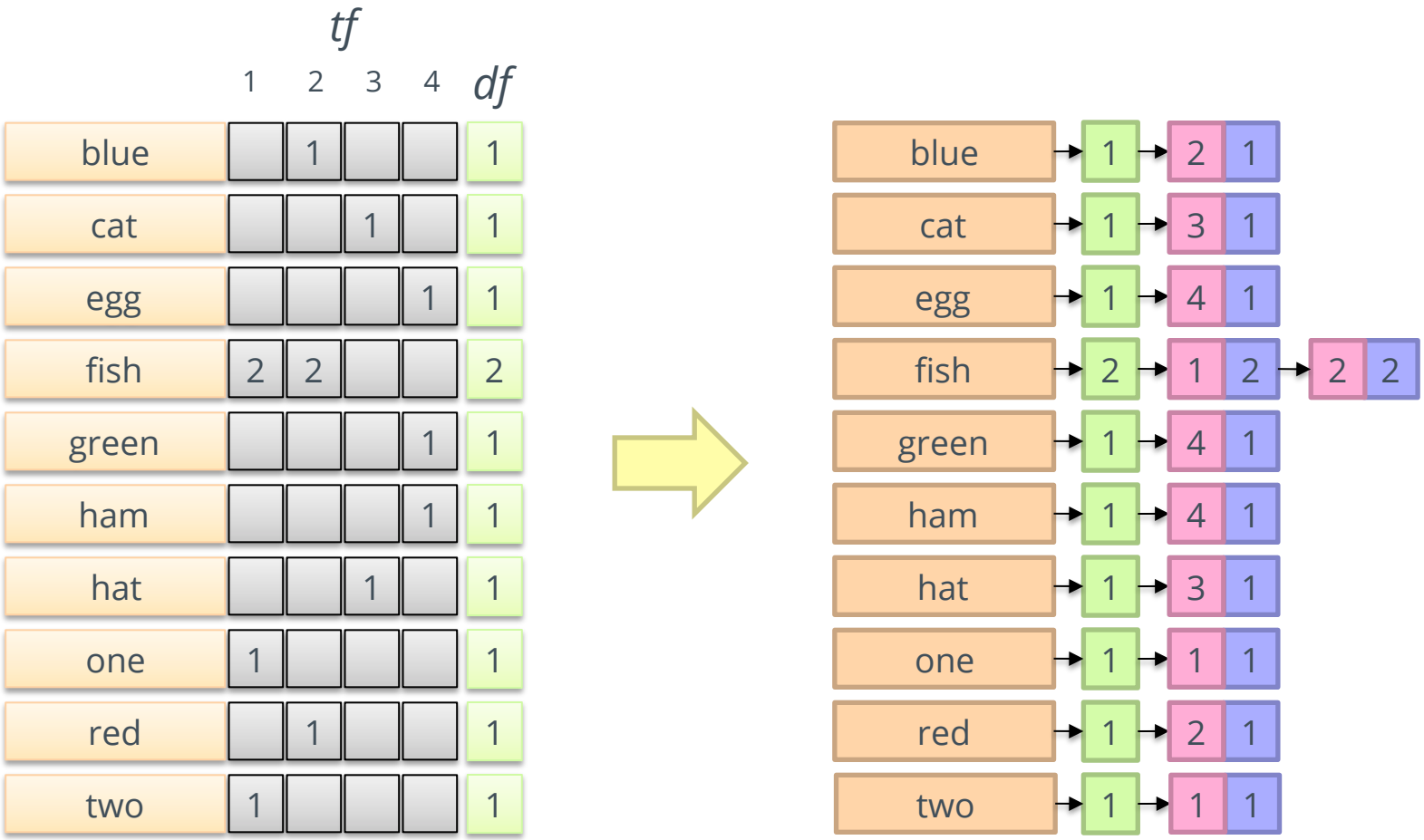
Inverted Index: TF.IDF

Doc 1 one fish, two fish

Doc 2 red fish, blue fish

Doc 3 cat in the hat

Doc 4 green eggs and ham



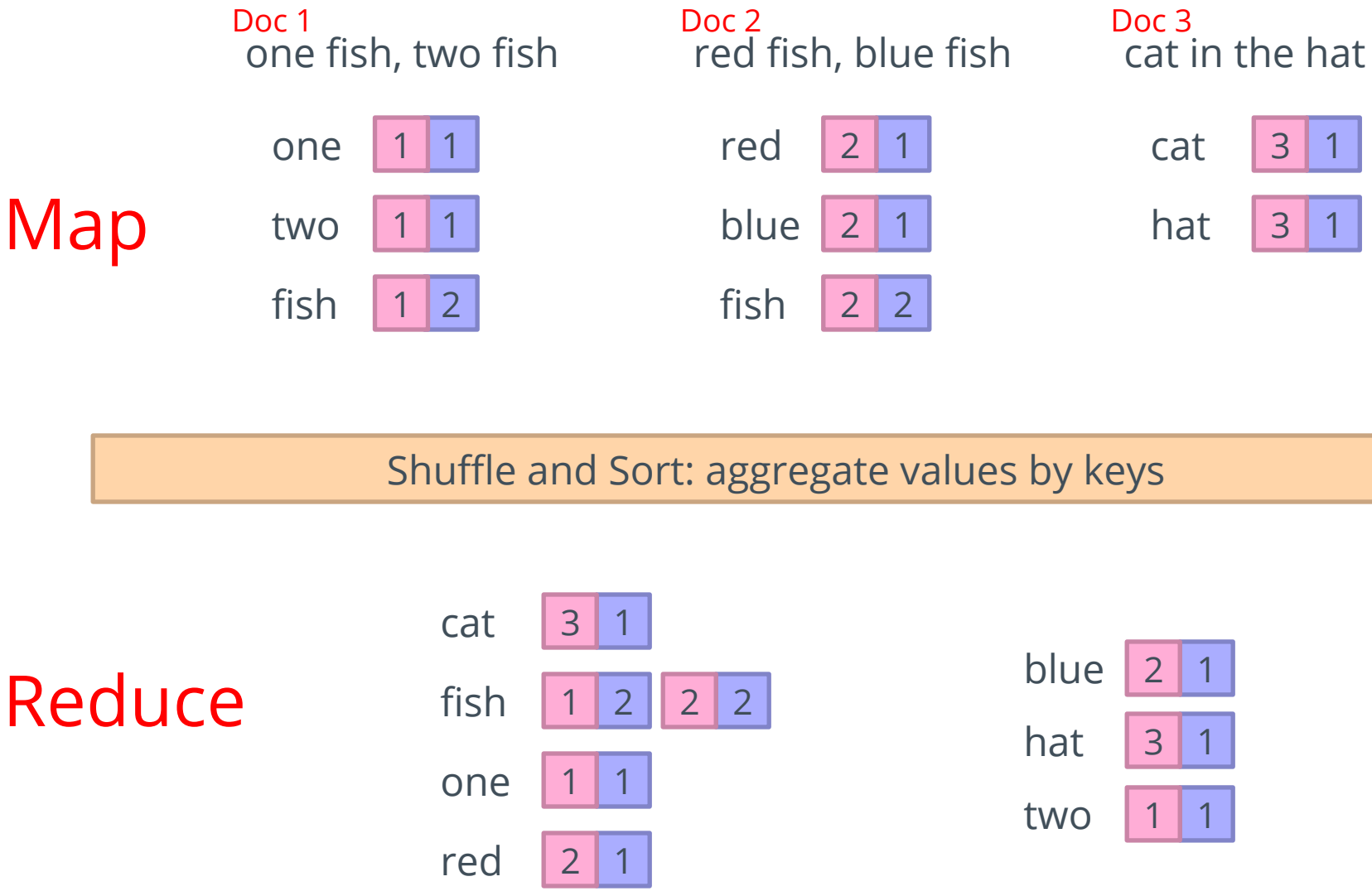
Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

MapReduce: Index Construction

- Map over all documents
  - Emit *term* as key, (*docno*, *tf)* as value
  - Emit other information as necessary (e.g., term position)
- Sort/shuffle: group postings by term
- Reduce
  - Gather and sort the postings (e.g., by *docno* or *tf*)
  - Write postings to disk
- MapReduce does all the heavy lifting!

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Inverted Indexing with MapReduce



Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Inverted Indexing: Pseudo-Code

```
1: class Mapper
2:     procedure Map(docid n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(term t, posting ⟨n, H{t}⟩)

1: class Reducer
2:     procedure Reduce(term t, postings [⟨a1, f1⟩, ⟨a2, f2⟩ . . .])
3:         P ← new List
4:         for all posting ⟨a, f⟩ ∈ postings [⟨a1, f1⟩, ⟨a2, f2⟩ . . .] do
5:             Append(P, ⟨a, f⟩)
6:         Sort(P)
7:         Emit(term t, postings P)
```

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Inverted Indexing: Pseudo-Code

```
1: class Mapper
2:     procedure Map(docid n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(term t, posting ⟨n, H{t}⟩)

1: class Reducer
2:     procedure Reduce(term t, postings [⟨a1, f1⟩, ⟨a2, f2⟩ . . .])
3:         P ← new List
4:         for all posting ⟨a, f⟩ ∈ postings [⟨a1, f1⟩, ⟨a2, f2⟩ . . .] do
5:             Append(P, ⟨a, f⟩)
6:         Sort(P)
7:         Emit(term t, postings P)
```

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Inverted Indexing: Pseudo-Code

```
1: class Mapper
2:     procedure Map(docid n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(term t, posting ⟨n, H{t}⟩)

1: class Reducer
2:     procedure Reduce(term t, postings [⟨a1, f1⟩, ⟨a2, f2⟩ . . .])
3:         P ← new List
4:         for all posting ⟨a, f⟩ ∈ postings [⟨a1, f1⟩, ⟨a2, f2⟩ . . .] do
5:             Append(P, ⟨a, f⟩)
6:         Sort(P)
7:         Emit(term t, postings P)
```

What's the problem?

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Scalability Bottleneck

- Initial implementation: terms as keys, postings as values
  - Reducers must buffer all postings associated with key (to sort)
  - What if we run out of memory to buffer postings

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Secondary Sorting

- MapReduce sorts input to reducers by key
  - Values may be arbitrarily ordered
- What if want to sort value also?
  - E.g., $k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r)\ldots$

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Secondary Sorting: Solutions

- Solution 1:
  - Buffer values in memory, then sort
  - Why is this a bad idea?
- Solution 2:
  - "Value-to-key conversion" design pattern: form composite intermediate key, (k, v_1)
  - Let execution framework do the sorting
  - Preserve state across multiple key-value pairs to handle processing
  - Anything else we need to do?

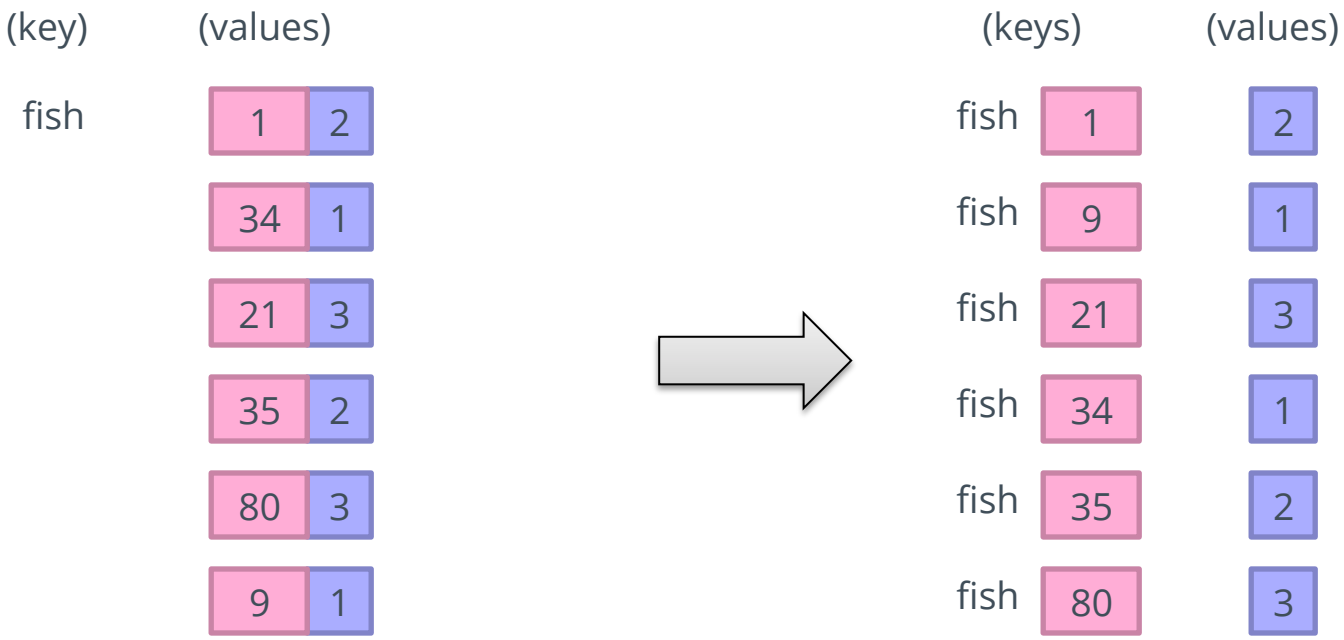
Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Another Try...

```
1: class Mapper
2:     method Map(docid n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(tuple ⟨t, n⟩, tf H{t})

1: class Reducer
2:     method Initialize
3:         t_prev ← ∅
4:         P ← new PostingsList
5:     method Reduce(tuple ⟨t, n⟩, tf [f])
6:         if t ≠ t_prev ∧ t_prev ≠ ∅ then
7:             Emit(term t_prev, postings P)
8:             P.Reset()
9:         P.Add(⟨n, f⟩)
10:        t_prev ← t
11:    method Close
12:        Emit(term t, postings P)
```

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Another Try…



How is this different?

- Let the framework do the sorting
- Directly write postings to disk!

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Questions?