Advanced Cloud Computing
MapReduce Algorithm Design

Wei Wang
CSE@HKUST
Spring 2022

THE DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
計算機科學及工程學系

How to express everything in terms of m, r, c, p?

Source: Wikipedia (The Scream)

Zen of MapReduce

Source: Wikipedia (Japanese rock garden)

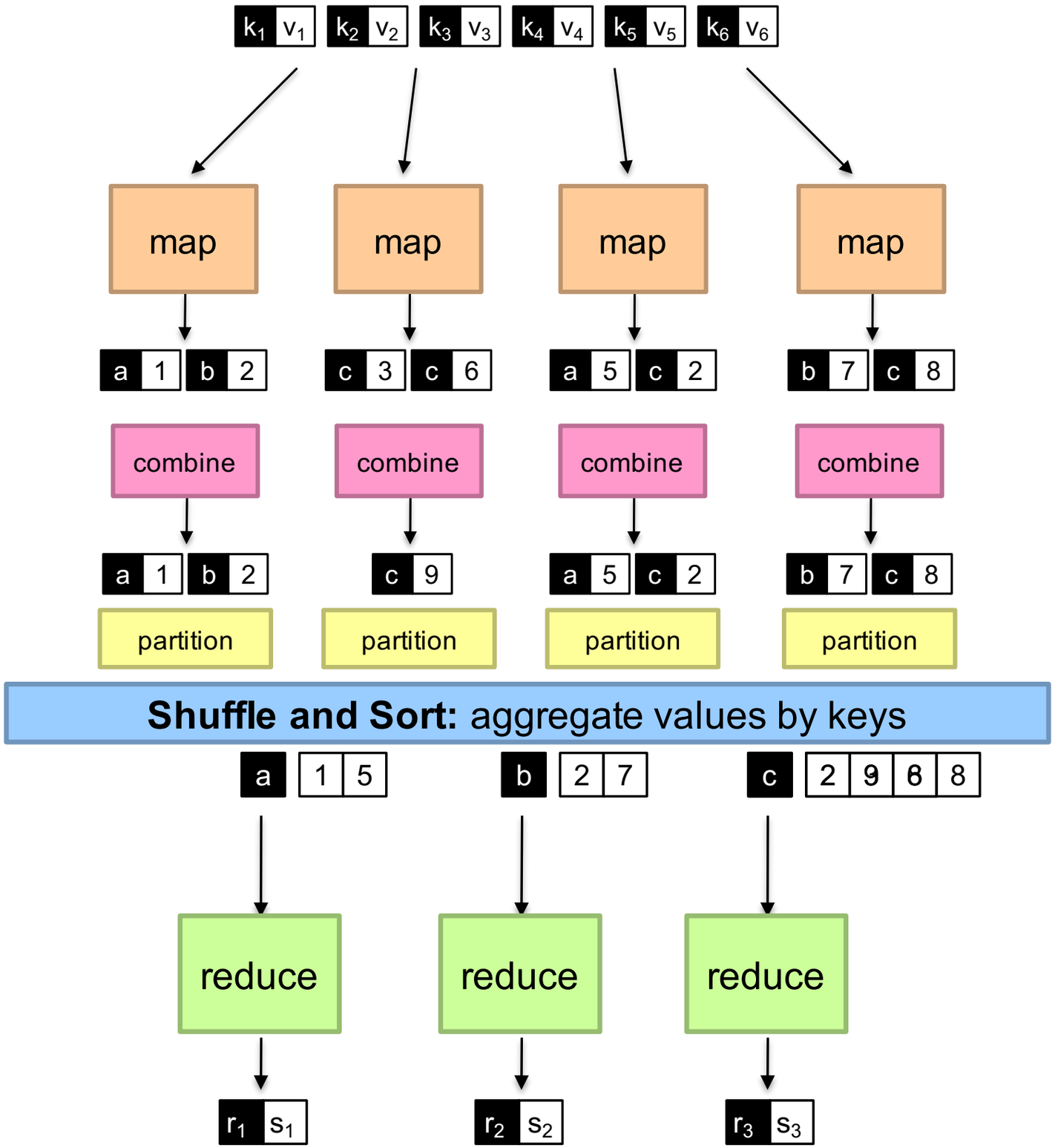
MapReduce

MapReduce: Recap

Programmers must specify:

- **map** (k, v) → [<k_2, v_2>]
- **reduce** (k_2, [v_2]) → [<k_3, v_3>]
  - all values with the same key are reduced together
- Optionally also:
  - **combine** (k, [v]) → <k, v>
  - **partition** (k, # of partitions)

The execution framework handles **everything else**…


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
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

“Everything else”

Scheduling

- assigns workers to map and reduce tasks

Data distribution

- moves process to data

Synchronization

- gathers, sorts, and shuffles intermediate data

Errors and faults

- detects worker failures and restarts

Limited control

All algorithms must be expressed in **m**, **r**, **c**, **p**

You don't know

- where mappers and reducers run
- when a mapper or reducer begins or finishes
- which input a particular mapper is processing
- which intermediate key a particular reducer is processing

But still we can control

Cleverly-constructed data structures

- bring partial results together

Sort order of intermediate keys

- control order in which reducers process keys

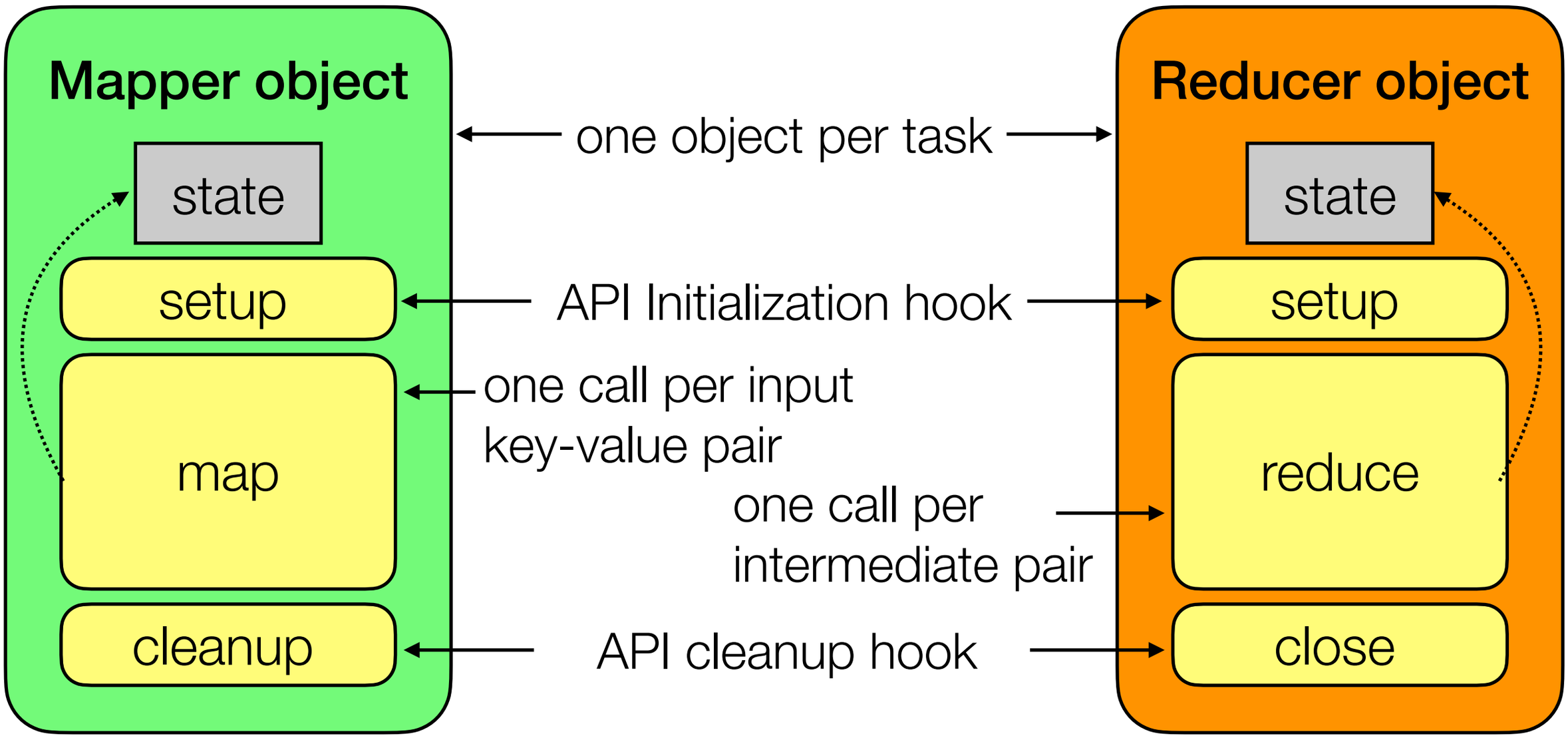
Partitioner

- control which reducer processes which keys

Preserving state in mappers and reducers

- capture dependencies across multiple keys and values

Preserving state



Scalable Hadoop Algorithms

Avoid object creation

- inherently costly operation
- garbage collection

Avoid buffering

- limited heap size
- works for small datasets, but won’t scale!

Importance of local aggregation

Ideal scaling characteristics

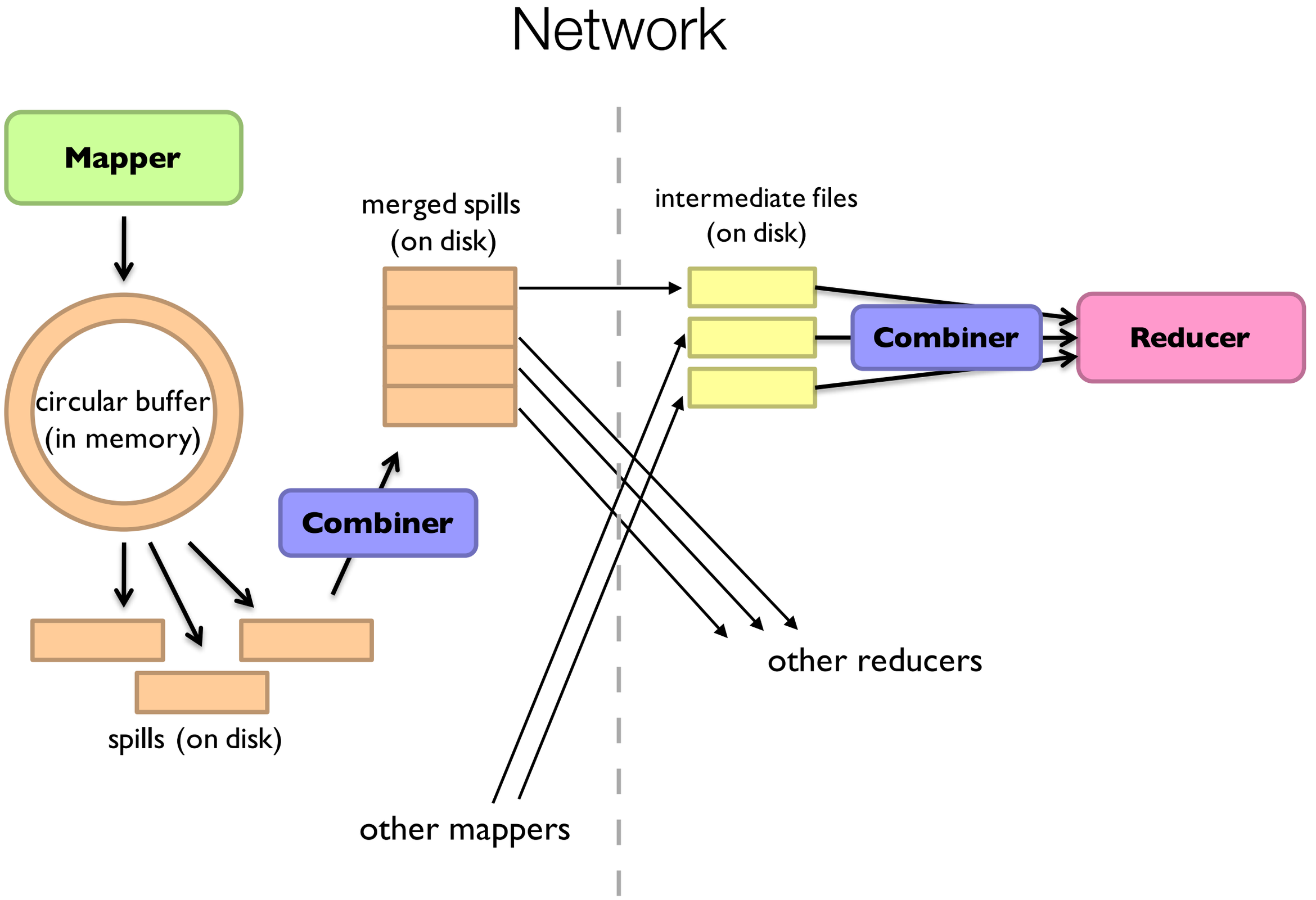
- twice the data, twice the running time
- twice the resources, half the running time

Why can’t we achieve this?

- synchronization requires communication
- communication kills performance

Thus… avoid communication, as much as possible!

- reduce intermediate data via local aggregation
- combiners can help


Network
Mapper
circular buffer
(in memory)
spills (on disk)
Combiner
merged spills
(on disk)
intermediate files
(on disk)
Combiner
Reducer
other reducers
other mappers

WordCount: 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 sum)
```

What's the impact of combiners?

WordCount: Version 1

H{*t*}: a hash table

```
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})
```

Do combiners still help?

WordCount: Version 2

H{*t*}: a hash table

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

```
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})
```

Do combiners still help?

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
- Order matters! May lead to order-dependent bugs!

Combiner design

Combiners and reducers share same method signature

- sometimes, reducers can serve combiners
- often, not…

Combiners are optional optimizations

- should not affect algorithm correctness
- may be run 0, 1, or multiple times (indefinite)

Example: find the mean of integers associated with the same key

Computing the mean: Version 1

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)

1: class Reducer
2:     method Reduce(string t, integers [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

Why can't we use reducer as combiner?

Computing the mean: Version 2

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)
1: class Combiner
2:     method Combine(string t, integers [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         Emit(string t, pair (sum, cnt))          ▷ Separate sum and count
1: class Reducer
2:     method Reduce(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

Why doesn't this work?

Computing the mean: Version 3

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, pair (r, 1))
```

Fixed?

```
1: class Combiner
2:     method Combine(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         Emit(string t, pair (sum, cnt))
```

```
1: class Reducer
2:     method Reduce(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

Computing the mean: Version 4

```
1: class Mapper
2:     method Initialize
3:         S ← new AssociativeArray
4:         C ← new AssociativeArray
5:     method Map(string t, integer r)
6:         S{t} ← S{t} + r
7:         C{t} ← C{t} + 1
8:     method Close
9:         for all term t ∈ S do
10:            Emit(term t, pair (S{t}, C{t}))
```

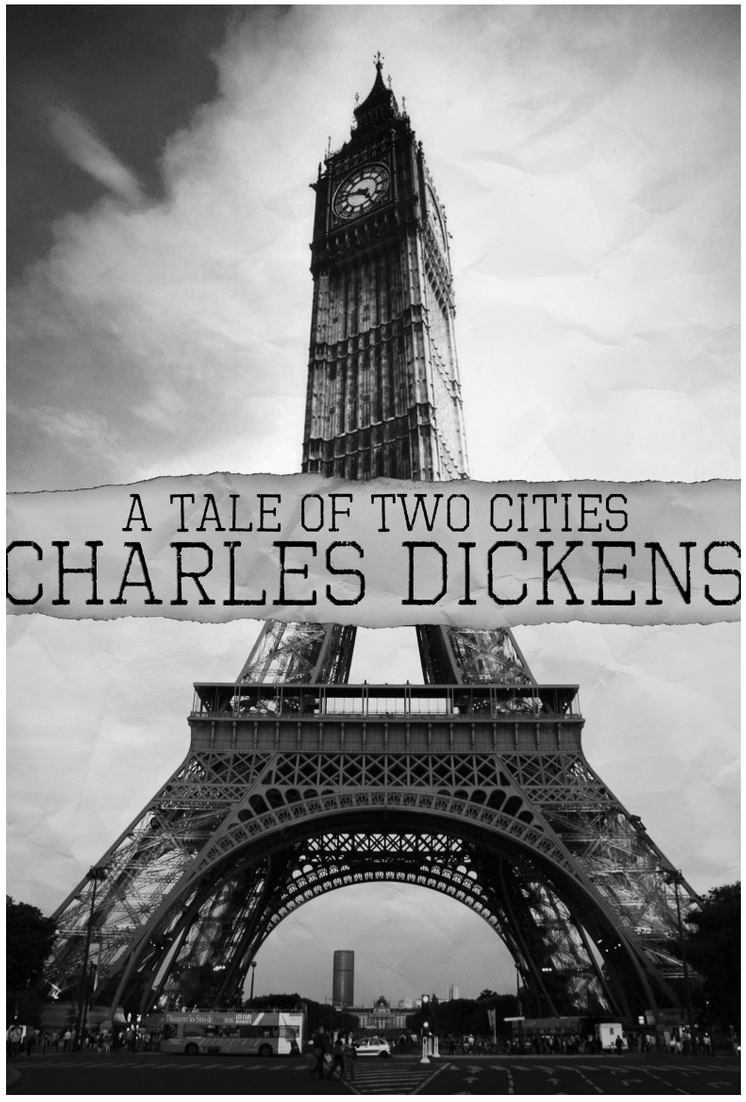
Are combiners still needed?

Algorithm design: a running example

Co-occurrence (bigram) count

“It was the best of times; It was the worse of times”

- “It was” → 2
- “was the” → 2
- “the best” → 1
- “best of” → 1
- “of times” → 2
- “the worst” → 1
- “worst of” → 1



Co-occurrence matrix

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 do we care?

- distributional profiles as a way of measuring semantic distance
- semantic distance useful for many language processing tasks

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?

First design pattern: “pairs”

“Pairs” approach

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 to minimize shuffling traffics!

“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)
```

“Pairs” analysis

Advantages

- easy to implement, easy to understand

Disadvantages

- lots of pairs to sort and shuffle around
  - What’s the upper bound?
- not many opportunities for combiners to work

Second design pattern:

“stripes”

“Stripes” approach

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 … }**

“Stripes” approach

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

“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)
```

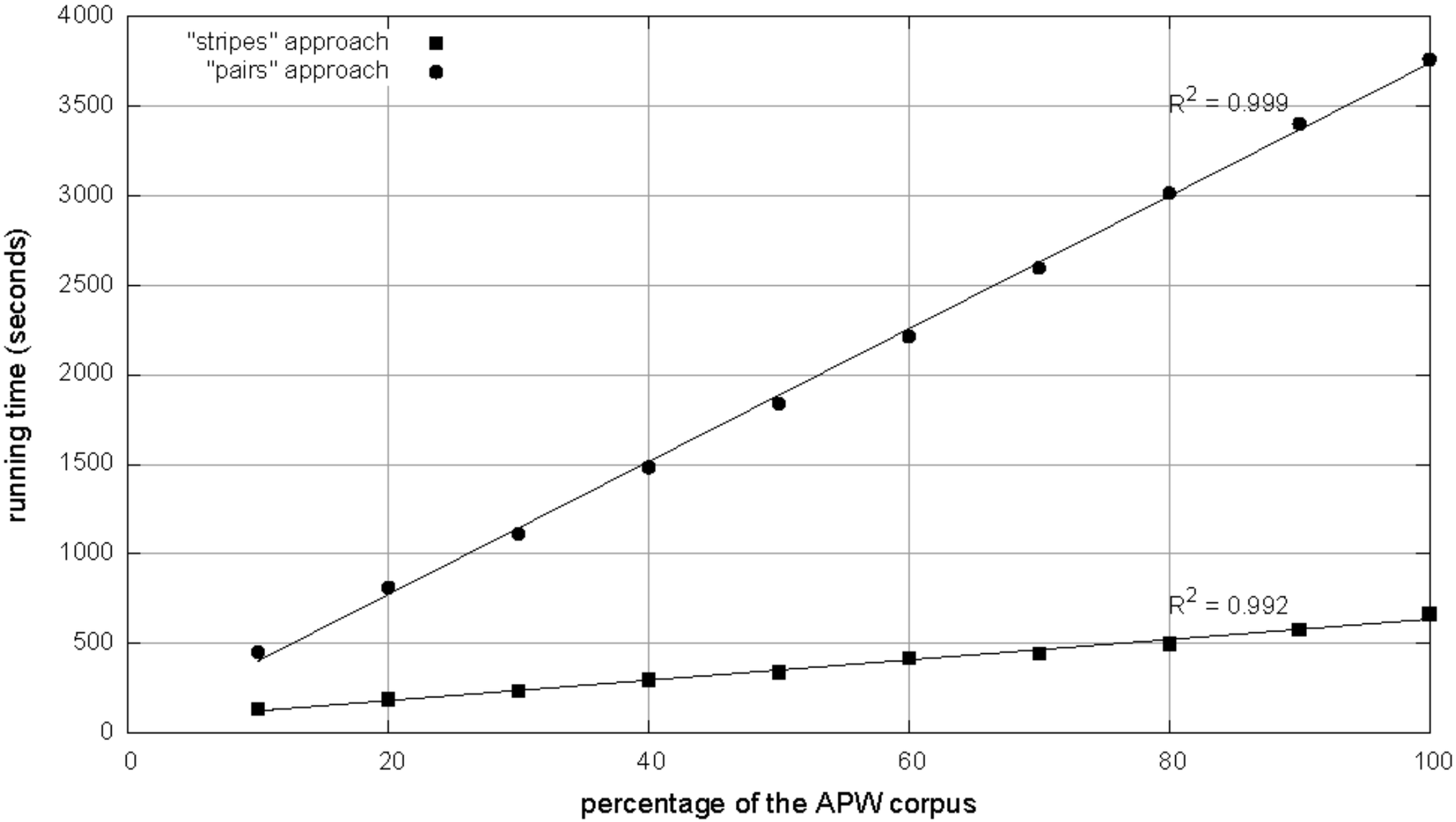
“Stripes” analysis

Advantages

- far less sorting and shuffling of key-value pairs
- can make better use of combiners

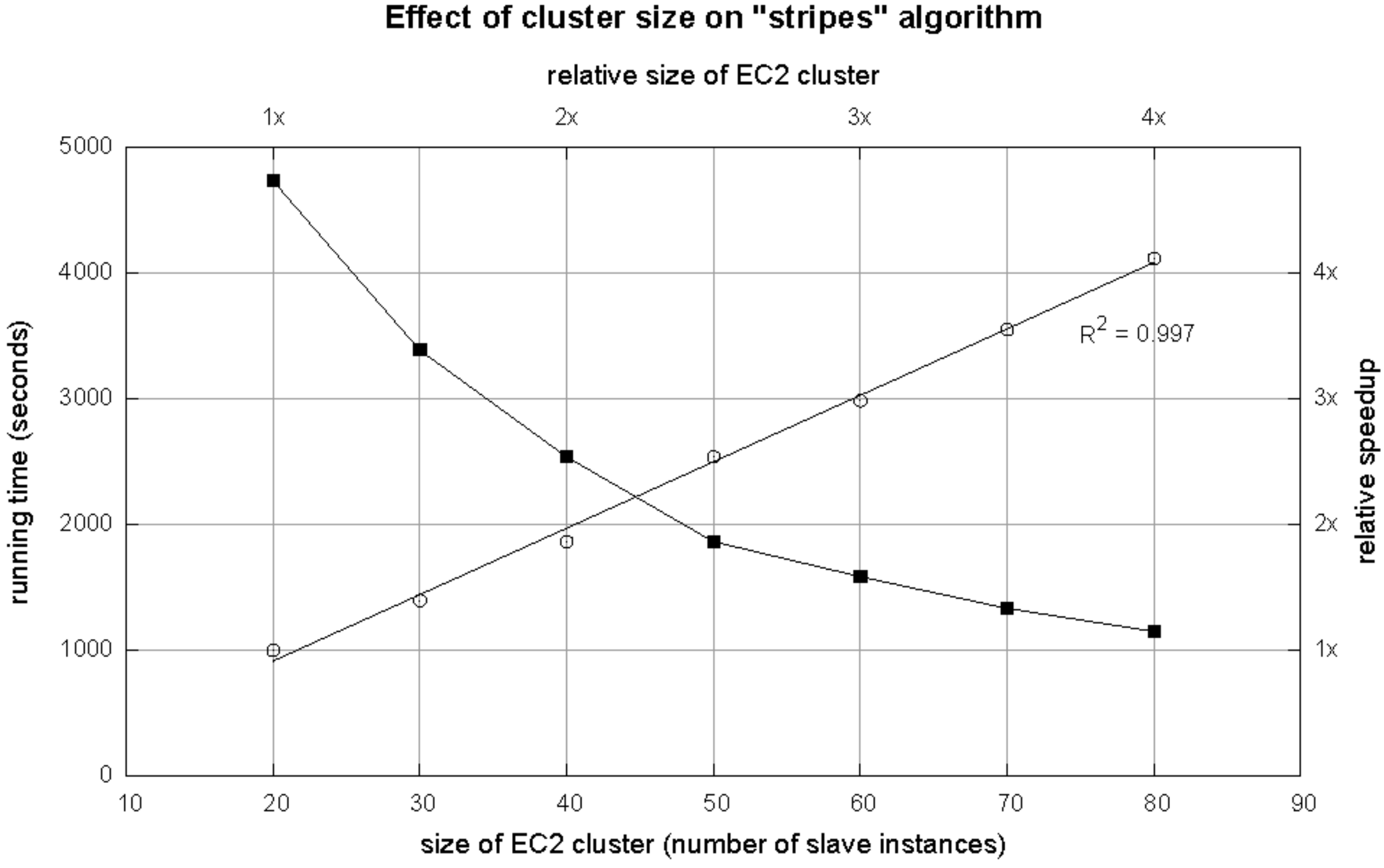
Disadvantages

- more difficult to implement
- underlying object more heavyweight
- fundamental limitation in terms of size of event space



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)

Lin et al., "Data-intensive text processing with MapReduce," Morgan & Claypool Publishers, 2010



Lin et al., “Data-intensive text processing with MapReduce,” Morgan & Claypool Publishers, 2010

Relative frequencies

How do we estimate relative frequencies from counts?

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

Why do we want to do this?

How do we do this with MapReduce?

f(B|A): “Stripes”

a → {b: 1, c: 2, d: 5, e: 3, f: 2}

Easy!

- one pass to compute **(a, *)**
- another pass to directly compute f(B|A)

f(B|A): “Pairs”

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

What’re the issues?

- computing relative frequencies requires marginal counts
- but the marginal cannot be computed until you see all counts

Buffering is a bad idea! (why?)

Solution: what if we could get the marginal count to arrive at the reducer first?

f(B|A): “Pairs”

(a, *) → 20 Reducer holds this value in memory

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

(a, b) → 1/20
(a, c) → 2/20
(a, d) → 5/20
(a, e) → 3/20
…

Emit extra **(a, *)** for every “**b**”, “**c**”, “**d**”, “**e**”… in mapper

Make sure all **a**’s get sent to the same reducer (partitioner)

Make sure **(a, *)** comes first (define sort order)

Hold state in reducer across different key-value pairs

From a reducer's perspective

key	**values**	
(dog, ∗)	[6327, 8514, . . .]	compute marginal: $\sum_{w'} N(\text{dog}, w') = 42908$
(dog, aardvark)	[2,1]	$f(\text{aardvark}\|\text{dog}) = 3/42908$
(dog, aardwolf)	[1]	$f(\text{aardwolf}\|\text{dog}) = 1/42908$
. . .		
(dog, zebra)	[2,1,1,1]	$f(\text{zebra}\|\text{dog}) = 5/42908$
(doge, ∗)	[682, . . .]	compute marginal: $\sum_{w'} N(\text{doge}, w') = 1267$

What to emit in mapper?

Emit extra **(a, *)** for every "**b**", "**c**", "**d**", "**e**"… in mapper

```
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)
6:                 Emit(pair (w, ""), count 1)
```

How to make sure that pair (w, "") is sorted in order before pair (w, u)?

Write your own data types

PairOfStrings implements **WritableComparable** interface

Must implement

- **write** for serialization
- **readFields** for deserialization
- **compareTo** to define sort order

Sample code available in Assignment-3

Partitioner

Make sure all **a**'s get sent to the same reducer (use partitioner)

```
private static class MyPartitioner extends
      Partitioner<PairOfStrings, IntWritable> {
   @Override
   public int getPartition(PairOfStrings key, IntWritable value,
         int numReduceTasks) {
      return (key.getLeftElement().hashCode() & Integer.MAX_VALUE)
            % numReduceTasks;
   }
}
```

Partition based on the left element only

Reducer

Make sure **(a, *)** comes first (already guaranteed)

Hold state in reducer across different key-value pairs

```
1: class Reducer
2:     method Reduce(pair p, counts [c_1, c_2, ...])
3:         s ← 0
4:         for all count c ∈ counts [c_1, c_2, ...] do
5:             s ← s + c                          ▷ Sum co-occurrence counts
6:         Emit(pair p, count s)
```

Reducer

Make sure **(a, *)** comes first (already guaranteed)

Hold state in reducer across different key-value pairs

```
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
```

```
if p.rightElement == "" then
    marginal ← s
else
    Emit(pair p, frequency s / marginal)
```

Where should *marginal* be declared and initialized?

You’ll write the complete code in Assignment-3

“Order inversion”

Common design pattern:

- take advantage of sorted key order at reducer to sequence computations
- get the marginal counts to arrive at the reducer before the joint counts

Optimization:

- apply in-memory combining pattern to accumulate marginal counts

Pairs vs. Stripes

Pairs

Turn synchronization into an ordering problem

- sort keys into correct order of computation
- partition key space so that each reducer gets the appropriate set of partial results
- hold state in reducer *across* multiple key-value pairs to perform computation

Stripes

Construct data structures that bring partial results together

- each reducer receives all data it needs to complete the computation
- be careful about the scalability issue: large stripes overflow the memory (and network)!
  - thus usually requires special treatment

Secondary sorting

Secondary sorting

In Hadoop, MapReduce sorts input to reducers by key

- values may be arbitrarily ordered
  - Google's proprietary implementation supports value sorting

What if we want to sort values as well?

- e.g., $k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r)\ldots$

Secondary sorting: solutions

Solution 1:

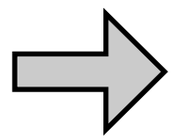
- buffer values in memory, then sort
- Is this a good idea? Why?

Solution 2:

- “value-to-key conversion” design pattern

“Value-to-key conversion”

Form composite intermediate key:

$k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r)\ldots$ 

$(k, v_1) \rightarrow r$
$(k, v_3) \rightarrow r$
$(k, v_4) \rightarrow r$
$(k, v_8) \rightarrow r$
…

Let execution framework do sorting

Preserve state *across* multiple key-value pairs to handle processing

Anything else we need to do?

Recap: tools for synchronization

Cleverly-constructed data structures

- bring data 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

Issues and tradeoffs

Number of key-value pairs

- Object creation overhead
- Time for sorting and shuffling pairs across the network

Size of each key-value pair

- De/serialization overhead

Issues and tradeoffs

Local aggregation

- Opportunities to perform local aggregation varies
- Combiners make a big difference
- Combiners vs. in-mapper combining
- RAM vs. disk vs. network

Debugging at scale

Work 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

Credits

Slides are adapted from Prof. Jimmy Lin’s slides at the University of Waterloo