MapReduce Algorithm Design

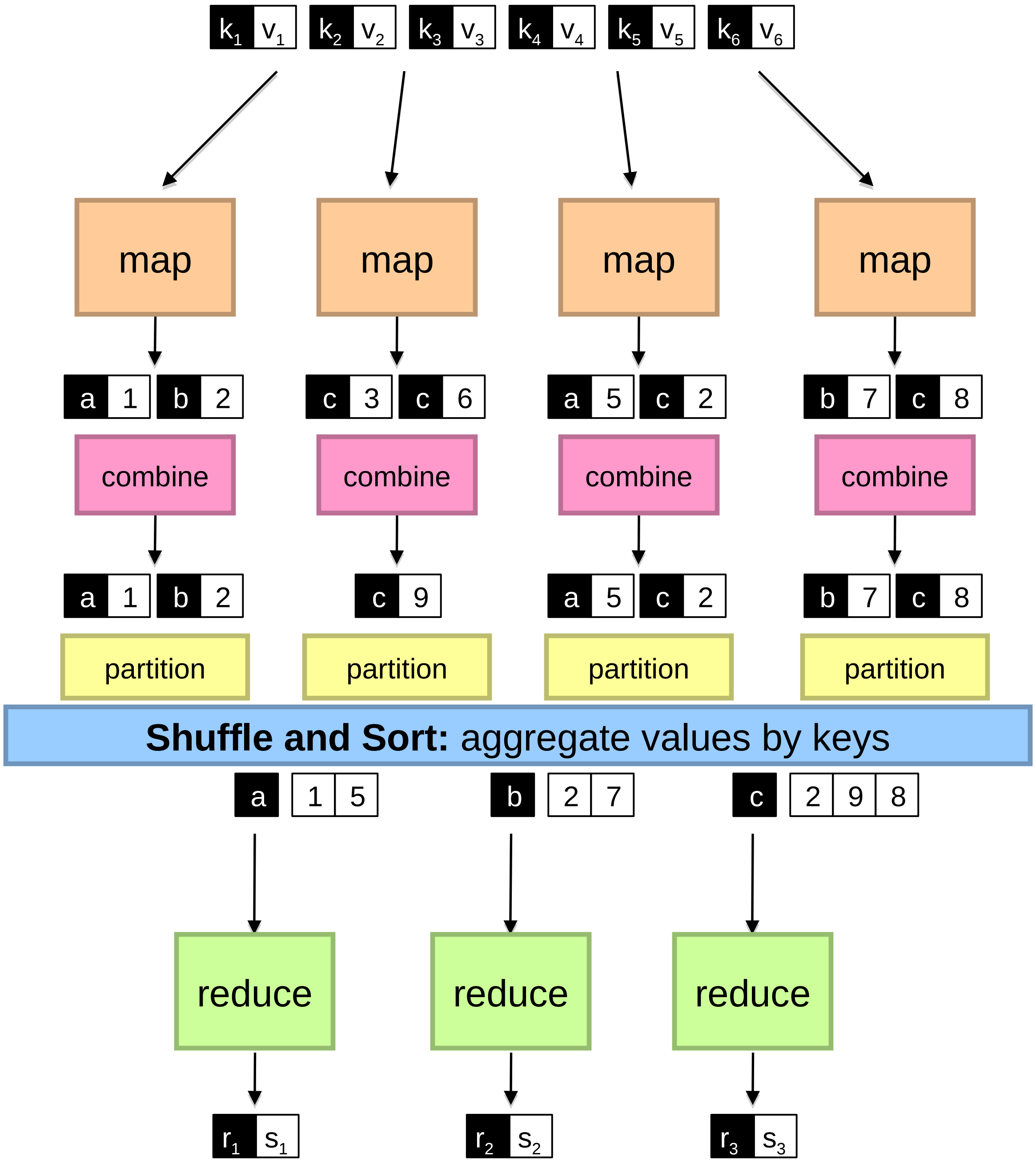
Adapted from Jimmy Lin's slides

MapReduce: Recap

- Programmers must specify:
  - **map** (k, v) → <k’, v’>*
  - **reduce** (k’, v’) → <k’, v’>*
  - – All values with the same key are reduced together
- Optionally, also:
  - **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
- The execution framework handles everything else…

“Everything Else”

- The execution framework handles everything else…
  - Scheduling: assigns workers to map and reduce tasks
  - “Data distribution”: moves processes to data
  - Synchronization: gathers, sorts, and shuffles intermediate data
  - Errors and faults: detects worker failures and restarts
- Limited control over data and execution flow
  - All algorithms must 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

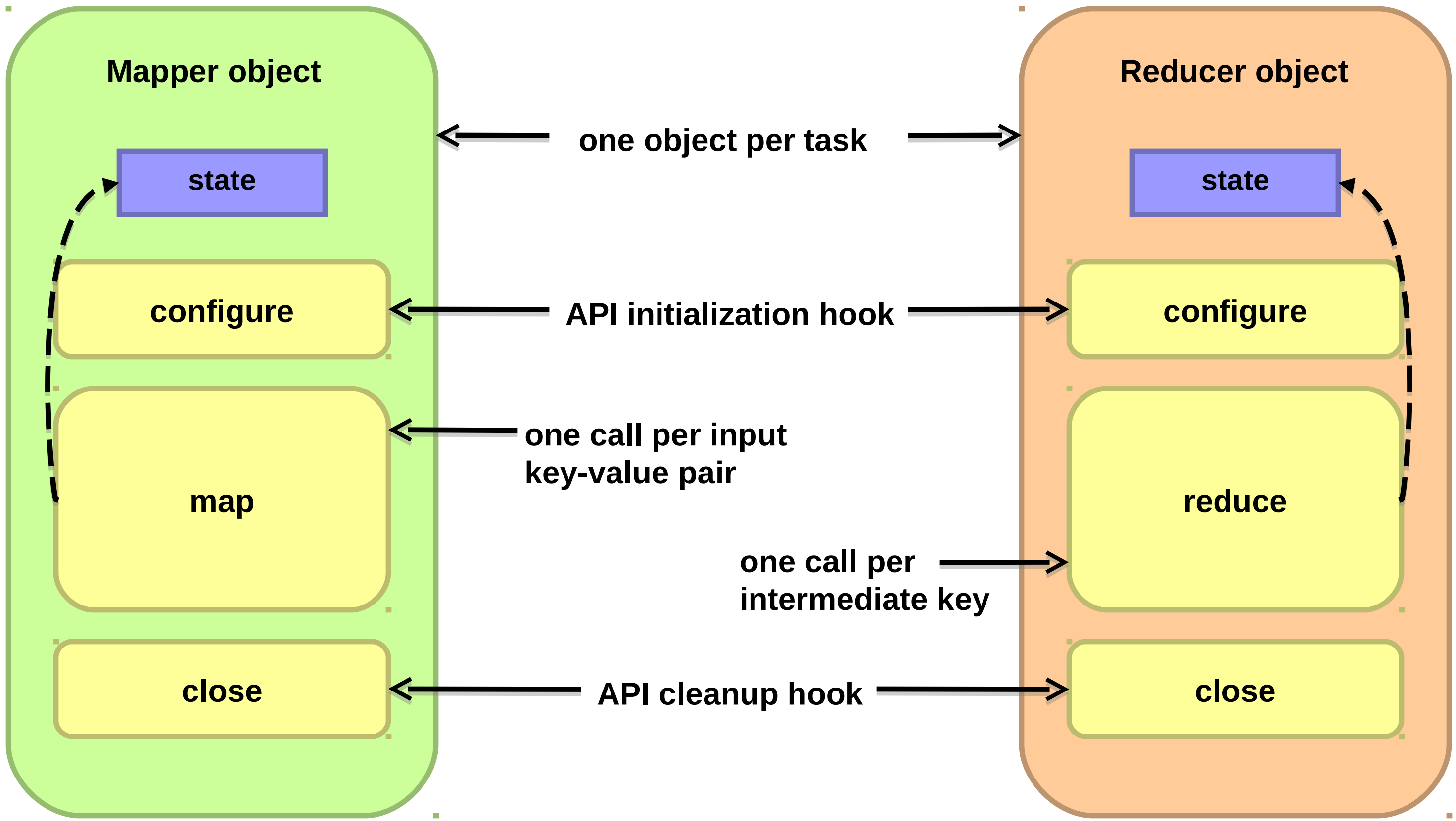


k_1 v_1 k_2 v_2 k_3 v_3 k_4 v_4 k_5 v_5 k_6 v_6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combine
combine
combine
combine
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 8
reduce
reduce
reduce
r_1 s_1
r_2 s_2
r_3 s_3

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

Preserving State



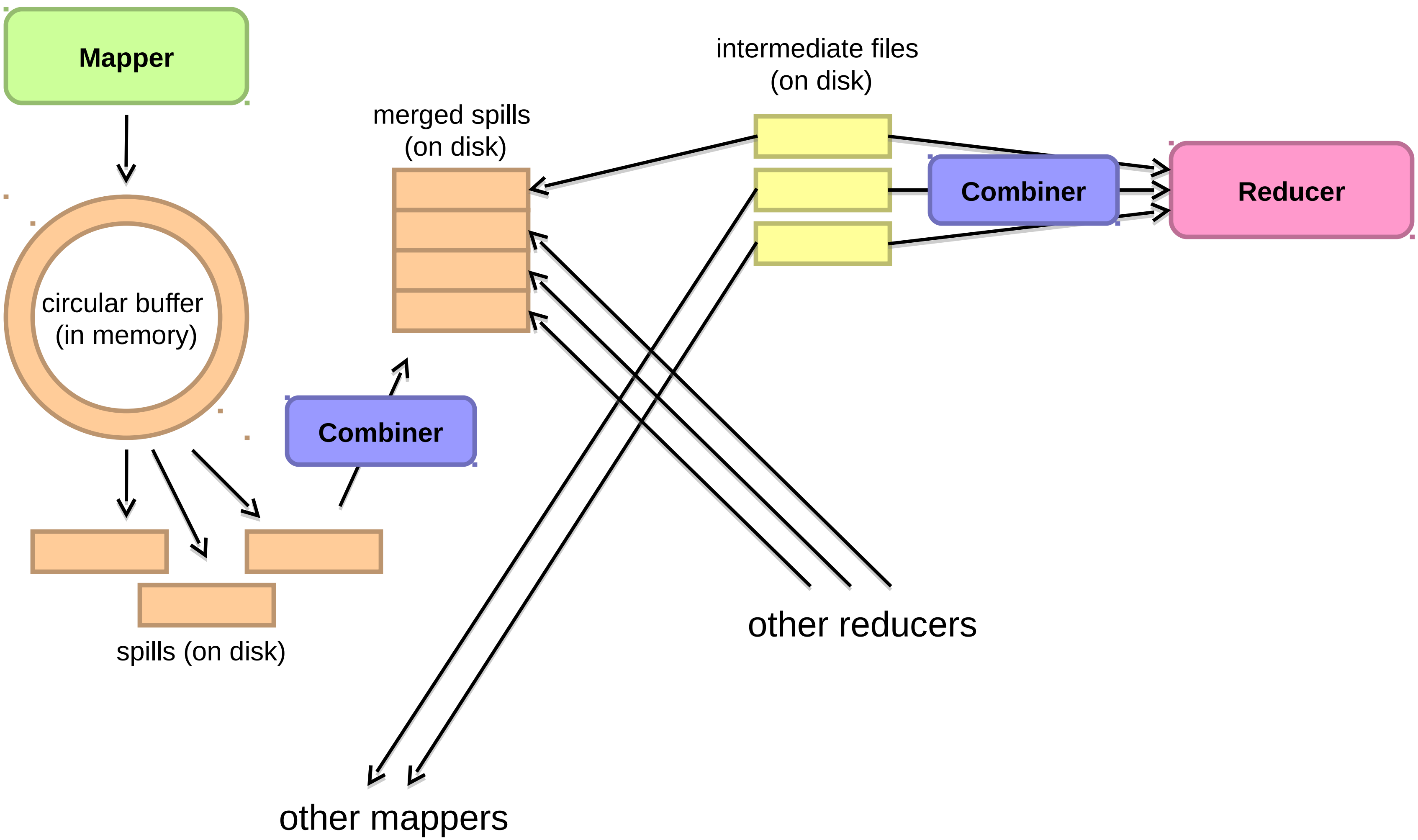
Scalable Hadoop Algorithms: Themes

- 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!
  - Reduce intermediate data via local aggregation
  - Combiners can help

Shuffle and Sort



Word Count: Baseline

class Mapper
  method Map(docid a, doc d)
    for all term $t \in$ doc d **do**
      Emit(term t, count 1)

class Reducer
  method Reduce(term t, counts $[c_1, c_2, \ldots]$)
    $sum \leftarrow 0$
    for all count $c \in$ counts $[c_1, c_2, \ldots]$ **do**
      $sum \leftarrow sum + c$
    Emit(term t, count s)

What’s the impact of combiners?

Word Count: Version 1

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

Are combiners still needed?

Word Count: Version 2

class MAPPER
    method INITIALIZE
        $H \leftarrow$ new ASSOCIATIVEARRAY
    method MAP(docid a, doc d)
        for all term $t \in$ doc d **do**
            $H\{t\} \leftarrow H\{t\} + 1$ ▷ Tally counts *across* documents
    method CLOSE
        for all term $t \in H$ **do**
            EMIT(term t, count $H\{t\}$)

Key: preserve state across
input key-value pairs!

Are combiners still needed?

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

Combiner Design

- Combiners and reducers share same method signature
  - Sometimes, reducers can serve as combiners
  - Often, not…
- Remember: combiner are optional optimizations
  - Should not affect algorithm correctness
  - May be run 0, 1, or multiple times
- Example: find average of all integers associated with the same key

Computing the Mean: Version 1

```
class Mapper
    method Map(string t, integer r)
        Emit(string t, integer r)

class Reducer
    method Reduce(string t, integers [r1, r2, ...])
        sum ← 0
        cnt ← 0
        for all integer r ∈ integers [r1, r2, ...] do
            sum ← sum + r
            cnt ← cnt + 1
        r_avg ← sum/cnt
        Emit(string t, integer r_avg)
```

Why can’t we use reducer as combiner?

Computing the Mean: Version 2

class MAPPER
  method MAP(string t, integer r)
    EMIT(string t, integer r)

class COMBINER
  method COMBINE(string t, integers $[r_1, r_2, \ldots]$)
    $sum \leftarrow 0$
    $cnt \leftarrow 0$
    for all integer $r \in$ integers $[r_1, r_2, \ldots]$ **do**
      $sum \leftarrow sum + r$
      $cnt \leftarrow cnt + 1$
    EMIT(string t, pair (sum, cnt))   ▷ Separate sum and count

class REDUCER
  method REDUCE(string t, pairs $[(s_1, c_1), (s_2, c_2) \ldots]$)
    $sum \leftarrow 0$
    $cnt \leftarrow 0$
    for all pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2) \ldots]$ **do**
      $sum \leftarrow sum + s$
      $cnt \leftarrow cnt + c$
    $r_{avg} \leftarrow sum/cnt$
    EMIT(string t, integer r_{avg})

Why doesn't this work?

Computing the Mean: Version 3

class Mapper
 method Map(string t, integer r)
  Emit(string t, pair $(r, 1)$)

class Combiner
 method Combine(string t, pairs $[(s_1, c_1), (s_2, c_2) \ldots]$)
  $sum \leftarrow 0$
  $cnt \leftarrow 0$
  for all pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2) \ldots]$ **do**
   $sum \leftarrow sum + s$
   $cnt \leftarrow cnt + c$
  Emit(string t, pair (sum, cnt))

class Reducer
 method Reduce(string t, pairs $[(s_1, c_1), (s_2, c_2) \ldots]$)
  $sum \leftarrow 0$
  $cnt \leftarrow 0$
  for all pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2) \ldots]$ **do**
   $sum \leftarrow sum + s$
   $cnt \leftarrow cnt + c$
  $r_{avg} \leftarrow sum/cnt$
  Emit(string t, pair (r_{avg}, cnt))

Fixed?

Computing the Mean: Version 4

```
class Mapper
    method Initialize
        S ← new AssociativeArray
        C ← new AssociativeArray
    method Map(string t, integer r)
        S{t} ← S{t} + r
        C{t} ← C{t} + 1
    method Close
        for all term t ∈ S do
            Emit(term t, pair (S{t}, C{t}))
```

Are combiners still needed?

Algorithm Design: Running 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

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 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!

Pairs: Pseudo-Code

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

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

“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

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 }

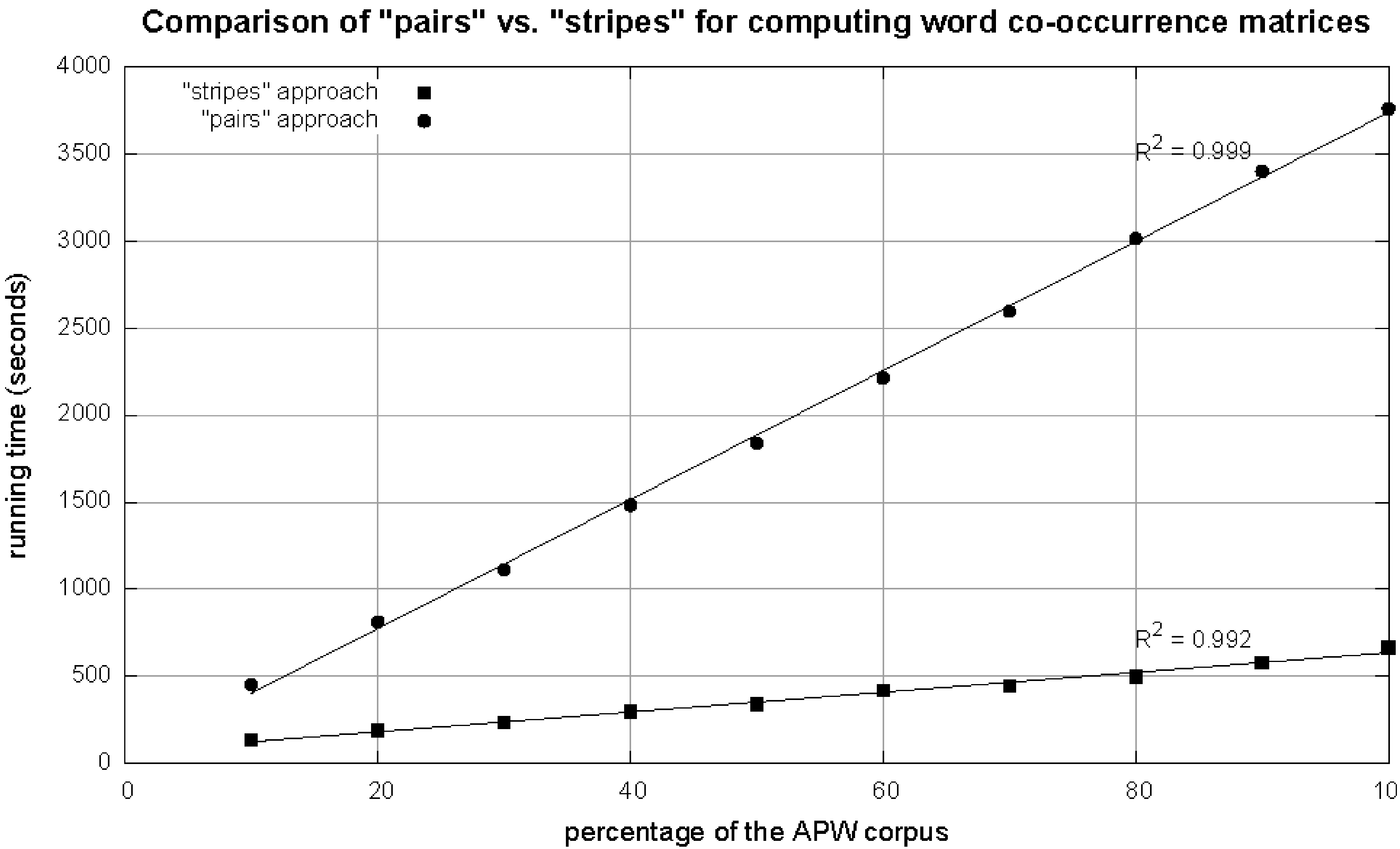
Key: cleverly-constructed data structure brings together partial results

Stripes: Pseudo-Code

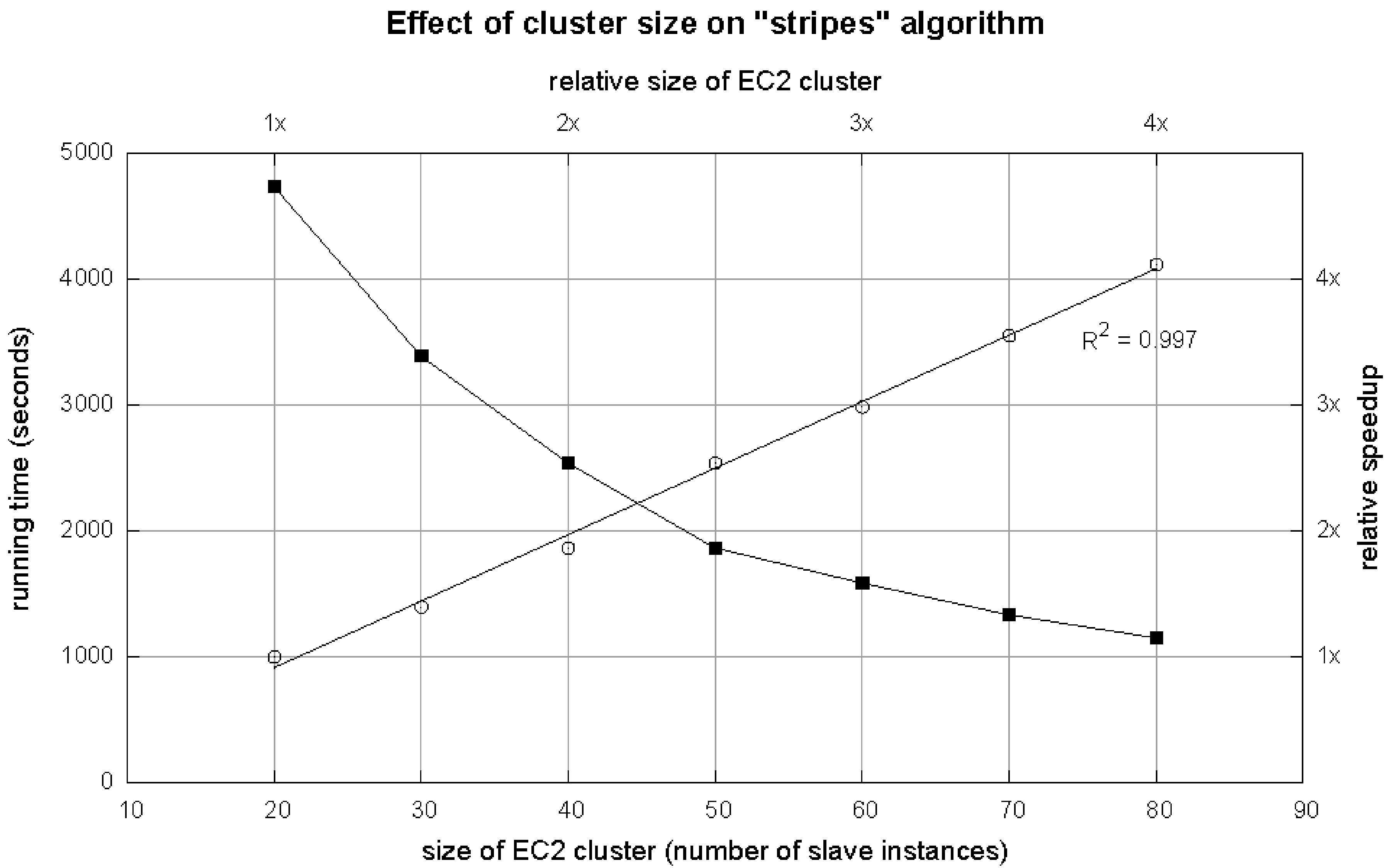
```
class Mapper
    method Map(docid a, doc d)
        for all term w ∈ doc d do
            H ← new AssociativeArray
            for all term u ∈ Neighbors(w) do
                H{u} ← H{u} + 1                    ▷ Tally words co-occurring with w
            Emit(Term w, Stripe H)

class Reducer
    method Reduce(term w, stripes [H_1, H_2, H_3, ...])
        H_f ← new AssociativeArray
        for all stripe H ∈ stripes [H_1, H_2, H_3, ...] do
            Sum(H_f, H)                            ▷ Element-wise sum
        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)


Effect of cluster size on "stripes" algorithm
relative size of EC2 cluster
1x
2x
3x
4x
5000
4000
3000
2000
1000
0
running time (seconds)
relative speedup
4x
3x
2x
1x
$R^2 = 0.997$
10
20
30
40
50
60
70
80
90
size of EC2 cluster (number of slave instances)

Relative Frequencies

- How do we estimate relative frequencies from counts?

$$f(B \mid A) = \frac{\mathrm{count}(A, B)}{\mathrm{count}(A)} = \frac{\mathrm{count}(A, B)}{\sum_{B'} \mathrm{count}(A, B')}$$

- Why do we want to do this?
- How do we do this with MapReduce?

f(B|A): “Stripes”

- Easy!
  - One pass to compute (a, *)
  - Another pass to directly compute f(B|A)

$f(B|A)$: "Pairs"

$(a, *) \rightarrow 32$ **Reducer holds this value in memory**

$(a, b_1) \rightarrow 3$	$(a, b_1) \rightarrow 3 / 32$
$(a, b_2) \rightarrow 12$	$(a, b_2) \rightarrow 12 / 32$
$(a, b_3) \rightarrow 7$	$(a, b_3) \rightarrow 7 / 32$
$(a, b_4) \rightarrow 1$	$(a, b_4) \rightarrow 1 / 32$
…	…

- For this to work:
  - Must emit extra $(a, *)$ for every b_n in mapper
  - Must make sure all a's get sent to same reducer (use partitioner)
  - Must make sure $(a, *)$ comes first (define sort order)
  - Must hold state in reducer across different key-value pairs

“Order Inversion”

- Common design pattern
  - Computing relative frequencies requires marginal counts
  - But marginal cannot be computed until you see all counts
  - Buffering is a bad idea!
  - Trick: getting the marginal counts to arrive at the reducer before the joint counts
- Optimizations
  - Apply in-memory combining pattern to accumulate marginal counts
  - Should we apply combiners?

Synchronization: Pairs vs. Stripes

- Approach 1: 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
  - Illustrated by the "pairs" approach
- Approach 2: construct data structures that bring partial results together
  - Each reducer receives all the data it needs to complete the computation
  - Illustrated by the "stripes" approach

Secondary Sorting

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

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?

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