# ESTR4300/ ENGG5108

# MapReduce Design Patterns

Prof. Wing C. Lau
Department of Information Engineering
wclau@ie.cuhk.edu.hk

# Acknowledgements


- The materials and slides used in this topic are adapted from the following source:
  - Ch 3 (plus Section 2.4 and 2.6) of the book titled "Data-Intensive Information Processing Applications," by Jimmy Lin et al, University of Maryland, full-version freely available online.





This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States. See http://creativecommons.org/licenses/by-nc-sa/3.0/us/ for details

- All copyrights belong to the original authors of the material.

# Today's Agenda

- "MapReduce algorithm design
  - How do you express everything in terms of m, r, c, p?
  - Toward "design patterns"

# 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:

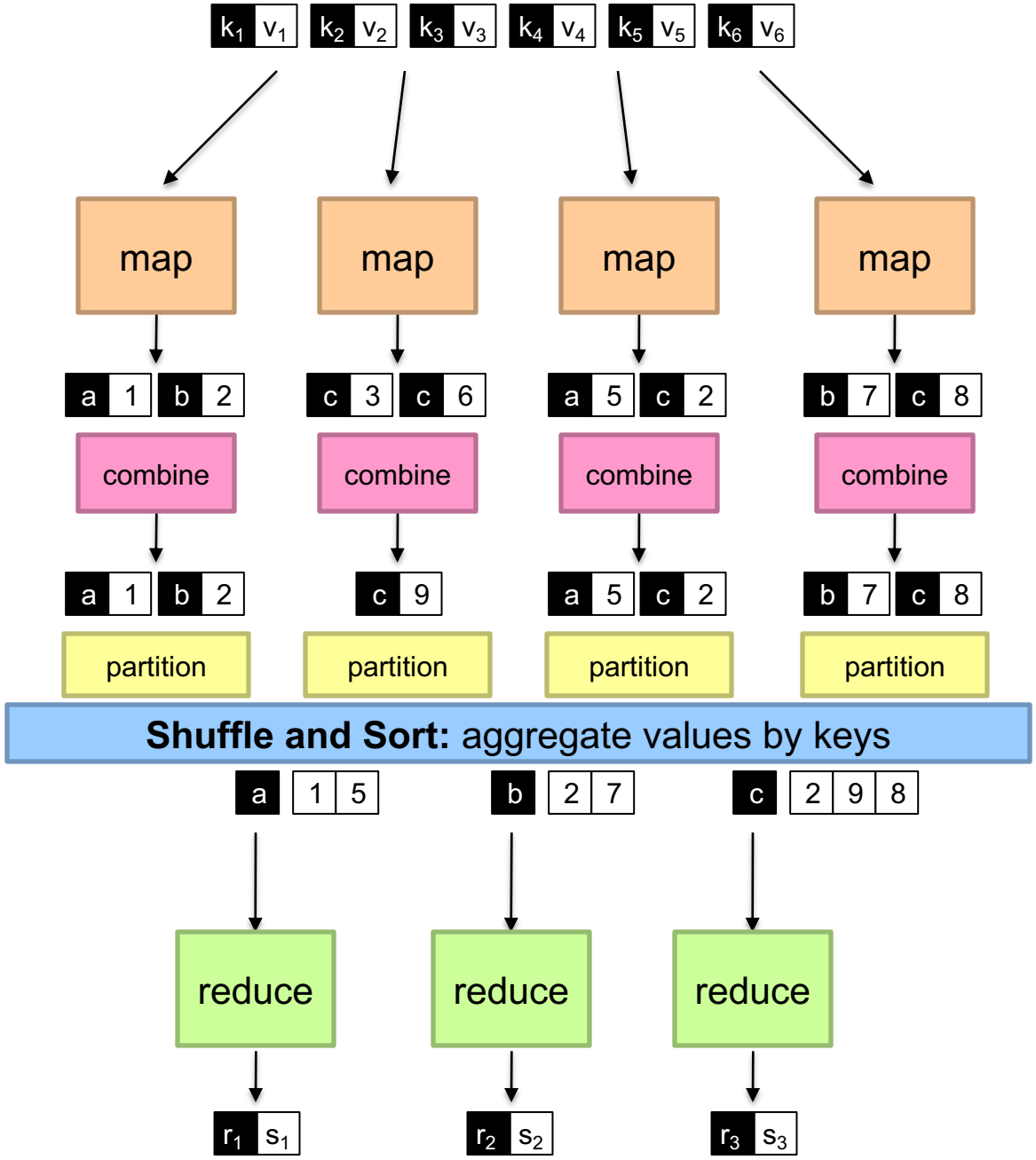
  **partition** (k', number of partitions) → partition for k'

  - 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'>*

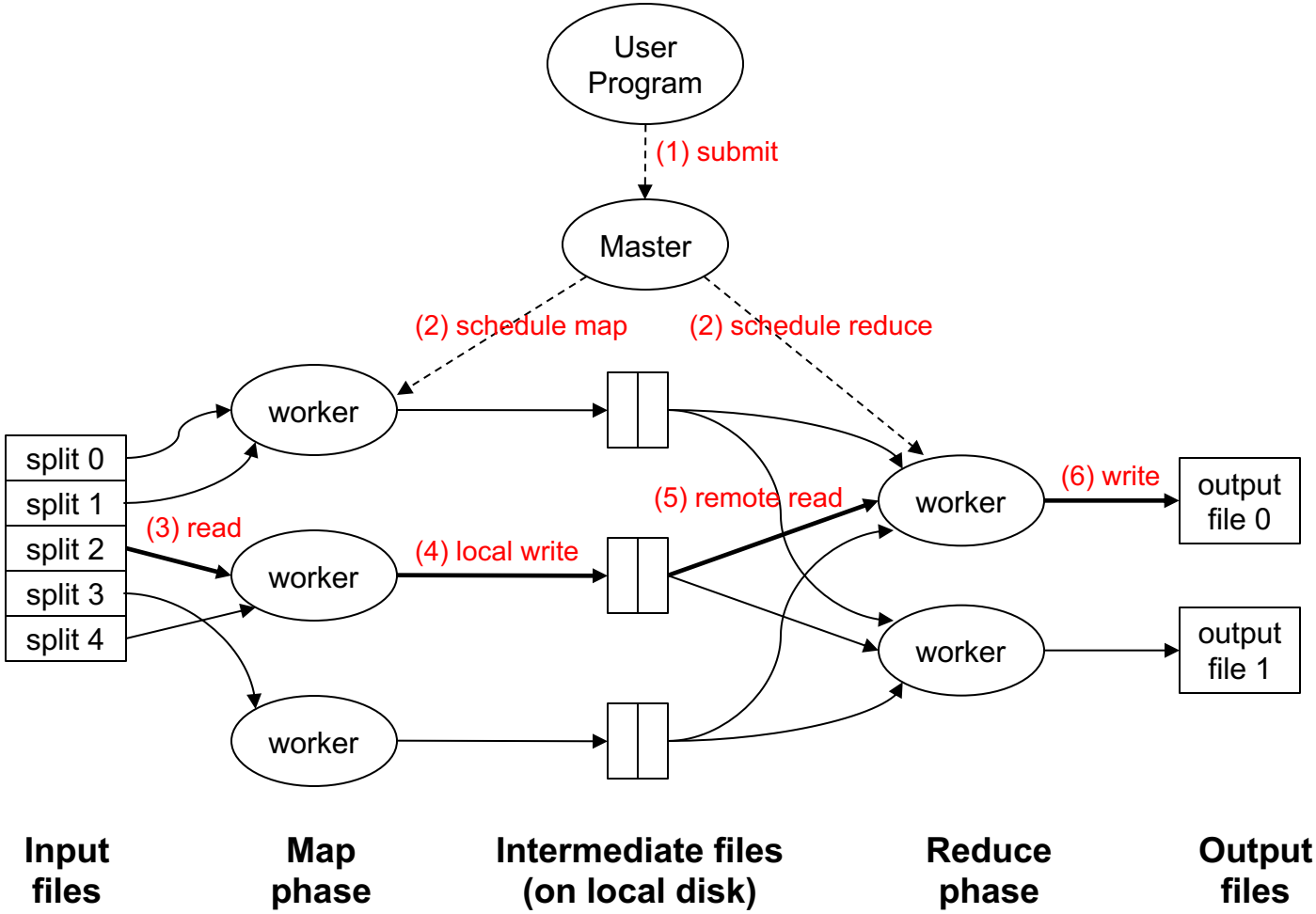
  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic

- The execution framework handles everything else…


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 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

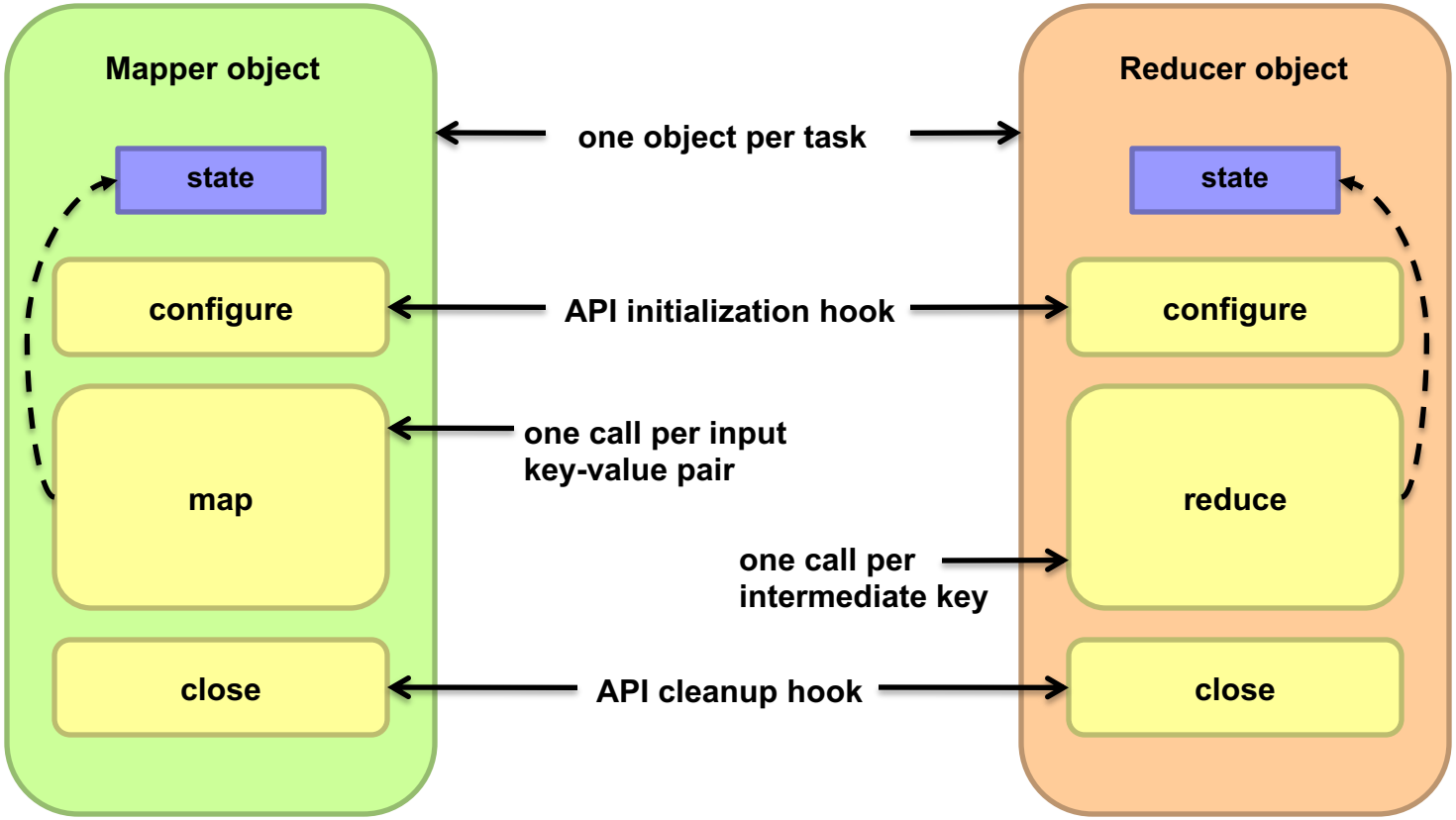
# “Everything Else”

- 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




Adapted from (Dean and Ghemawat, OSDI 2004)

# Preserving State



# 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

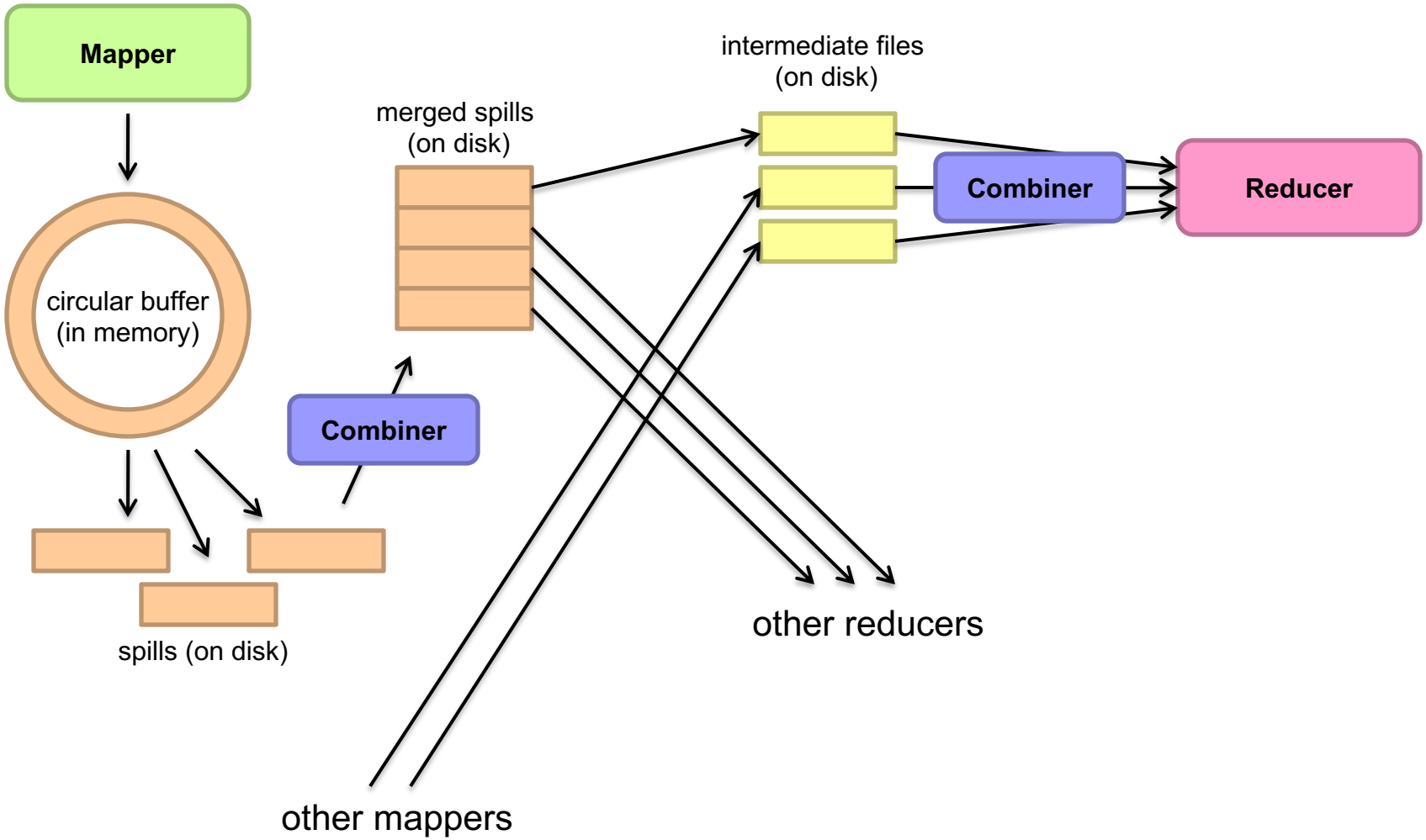
# Scalable Hadoop Algorithms: Themes

- 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

```
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)
```

**What's the impact of combiners?**

# 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})
```

**Are combiners still needed?**

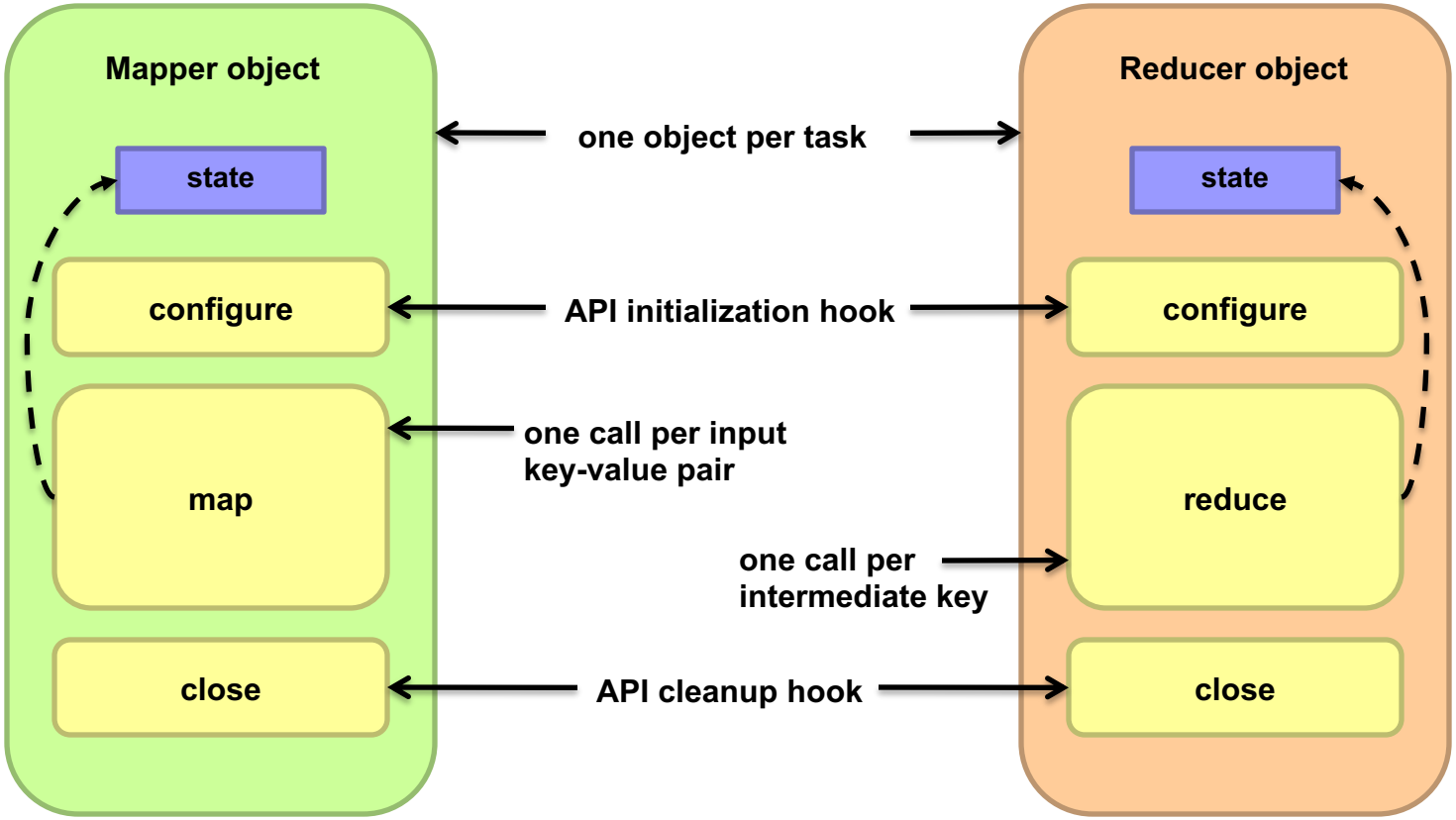
# Word Count: Version 2
# (The so-called "in-mapper combining" pattern)

```
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!**

**Are combiners still needed?**

# Recap: Preserving State



# 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
  - Violating "Pure form" of Functional programming

# Combiner Design

- Combiners & Reducers MUST 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

```
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))
```

```
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, pair (r_avg, cnt))
```

**Fixed?**

# 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: 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

```
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 (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 }
```

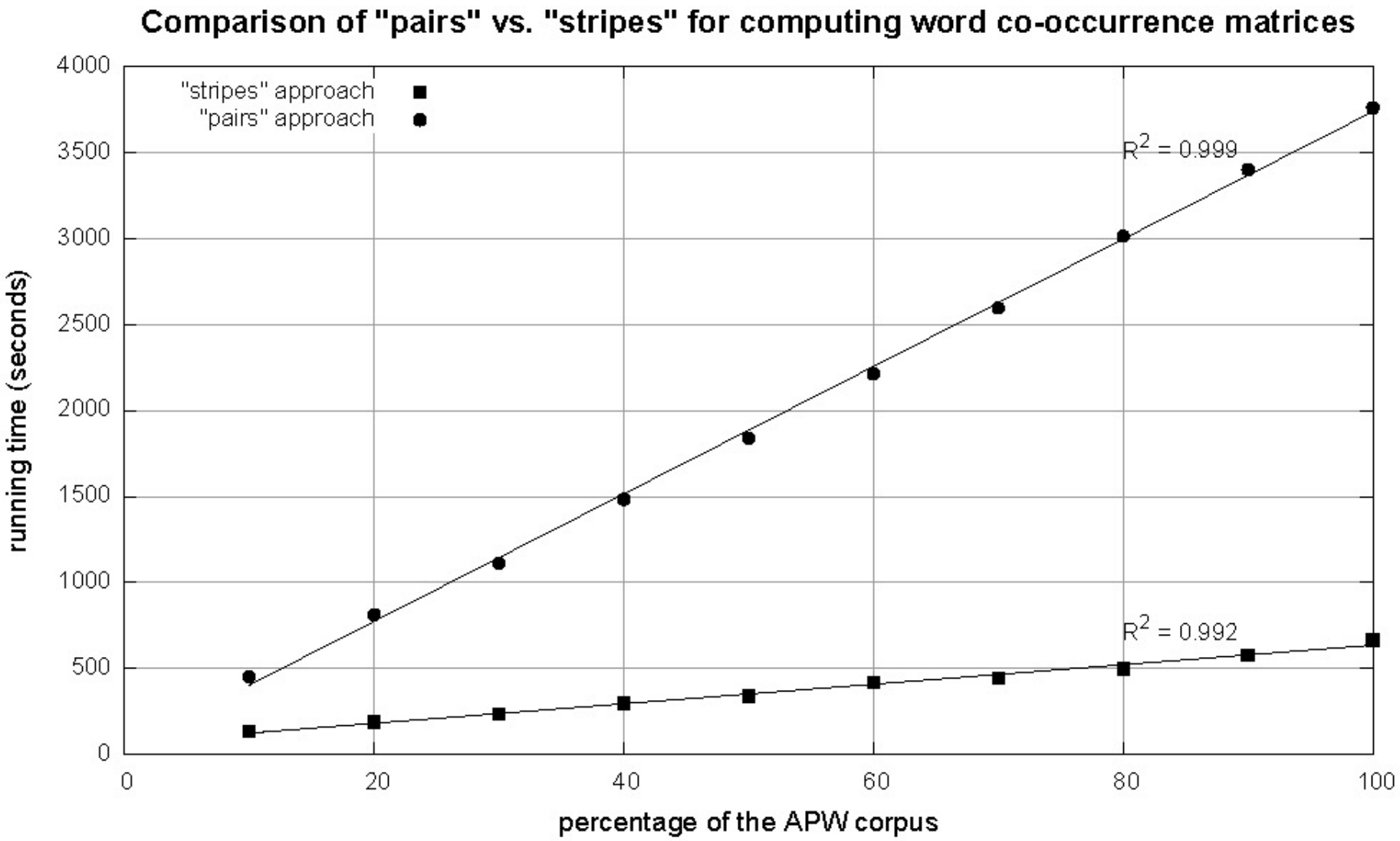
**Key Idea: cleverly-constructed data structure brings together partial results**

# Stripes: Pseudo-Code

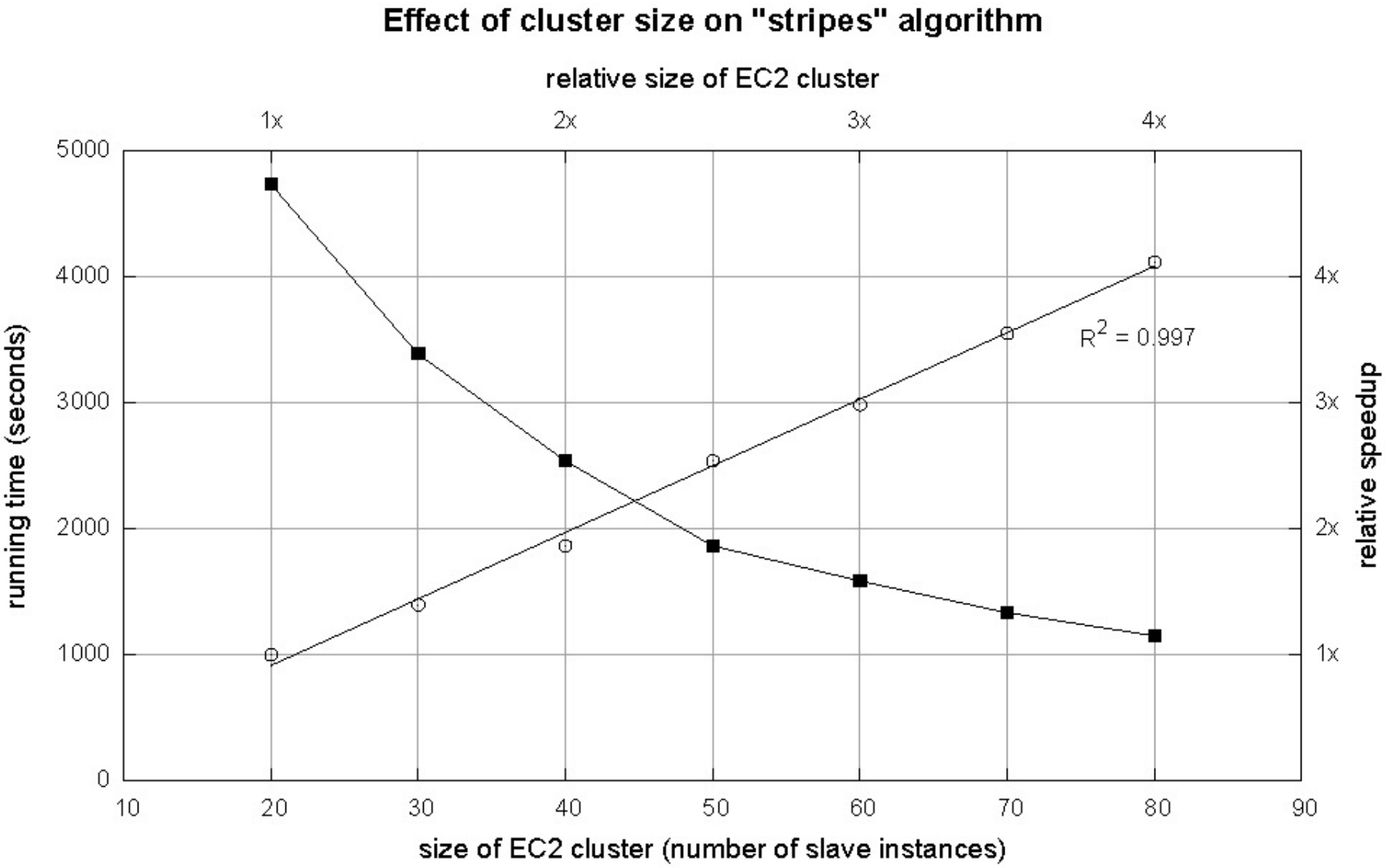
```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             H ← new AssociativeArray
5:             for all term u ∈ Neighbors(w) do
6:                 H{u} ← H{u} + 1                ▷ Tally words co-occurring with w
7:             Emit(Term w, Stripe H)

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)


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)
4x
3x
2x
1x
relative speedup
$R^2$ = 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?

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

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

# f(B|A): “Stripes”

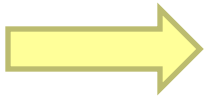
$a \rightarrow \{b_1:3, b_2:12, b_3:7, b_4:1, \ldots\}$

- 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**

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



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

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

e.g. where k may be the ID of a sensor ; v is the time of measurement and r and the measured value of this sensor.

# 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?

# Map-Reduce Join Patterns

- **Compute the natural join *R(A,B)* ⋈ *S(B,C)***
- *R* and *S* are each stored in files
- Tuples are pairs *(a,b)* or *(b,c)*

| A | B |
|---|---|
| $a_1$ | $b_1$ |
| $a_2$ | $b_1$ |
| $a_3$ | $b_2$ |
| $a_4$ | $b_3$ |

**R**

⋈

| B | C |
|---|---|
| $b_2$ | $c_1$ |
| $b_2$ | $c_2$ |
| $b_3$ | $c_3$ |

**S**

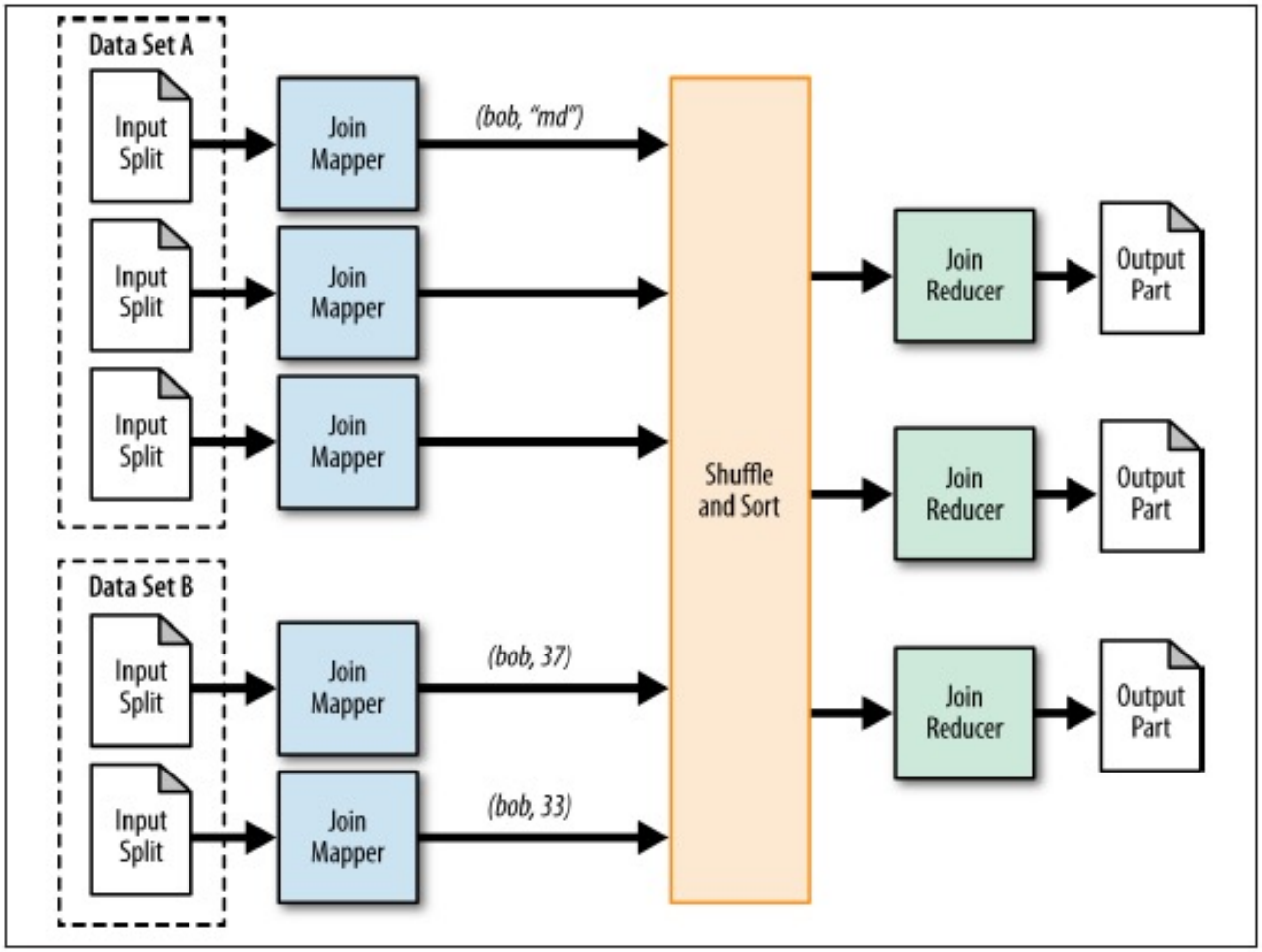
=

| A | C |
|---|---|
| $a_3$ | $c_1$ |
| $a_3$ | $c_2$ |
| $a_4$ | $c_3$ |

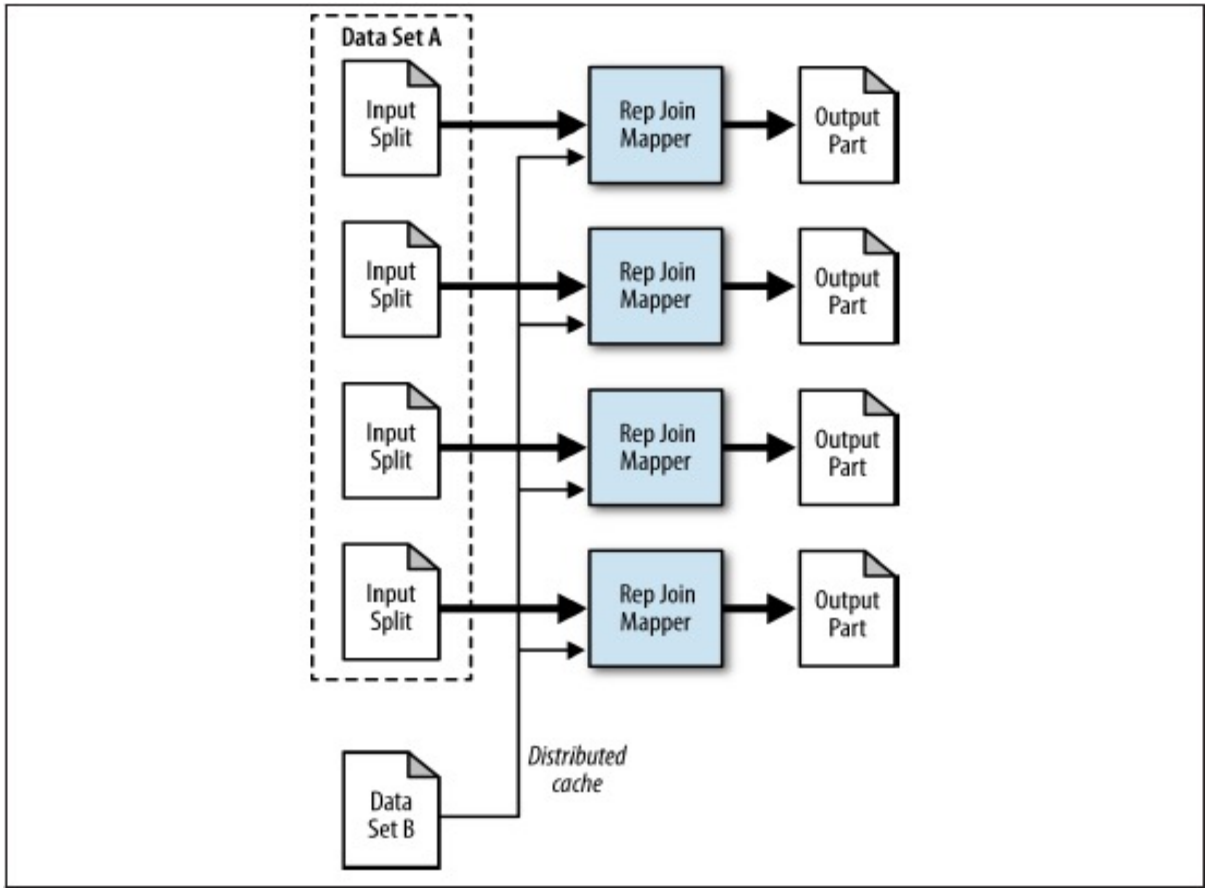
# Re-partition Join

- **Use a hash function *h* from B-values to *1...k***
- **A Map process turns:**
  - Each input tuple *R(a,b)* into key-value pair *(b,(a,R))*
  - Each input tuple *S(b,c)* into *(b,(c,S))*
- **Map processes** send each key-value pair with key *b* to Reduce process *h(b)*
  - Hadoop does this automatically; just tell it what *k* is.
- Each **Reduce process** matches all the pairs *(b,(a,R))* with all *(b,(c,S))* and outputs *(a,b,c)*.

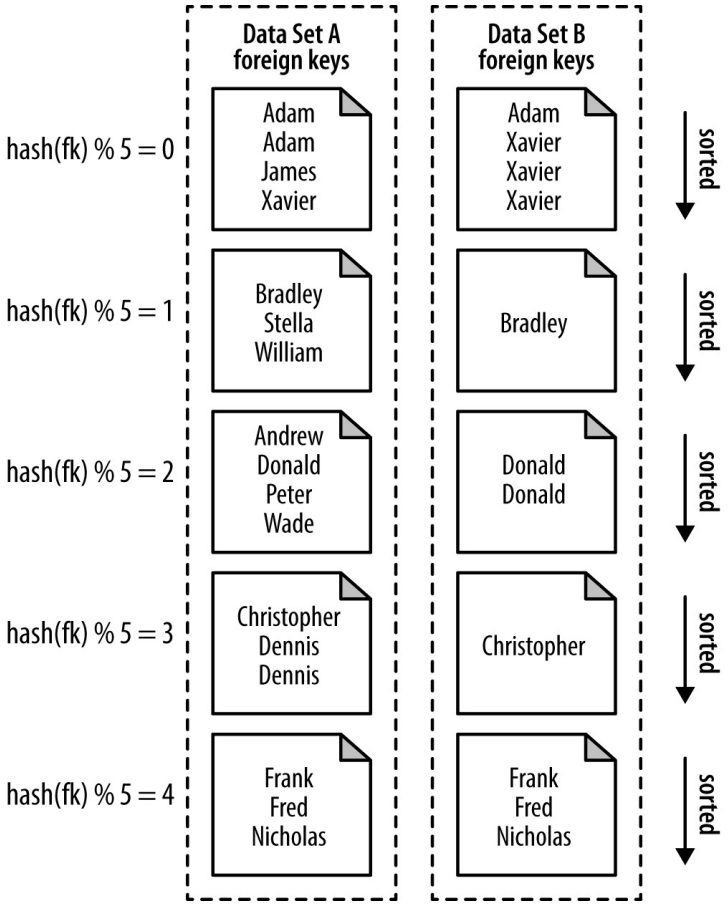
# Re-partition Join

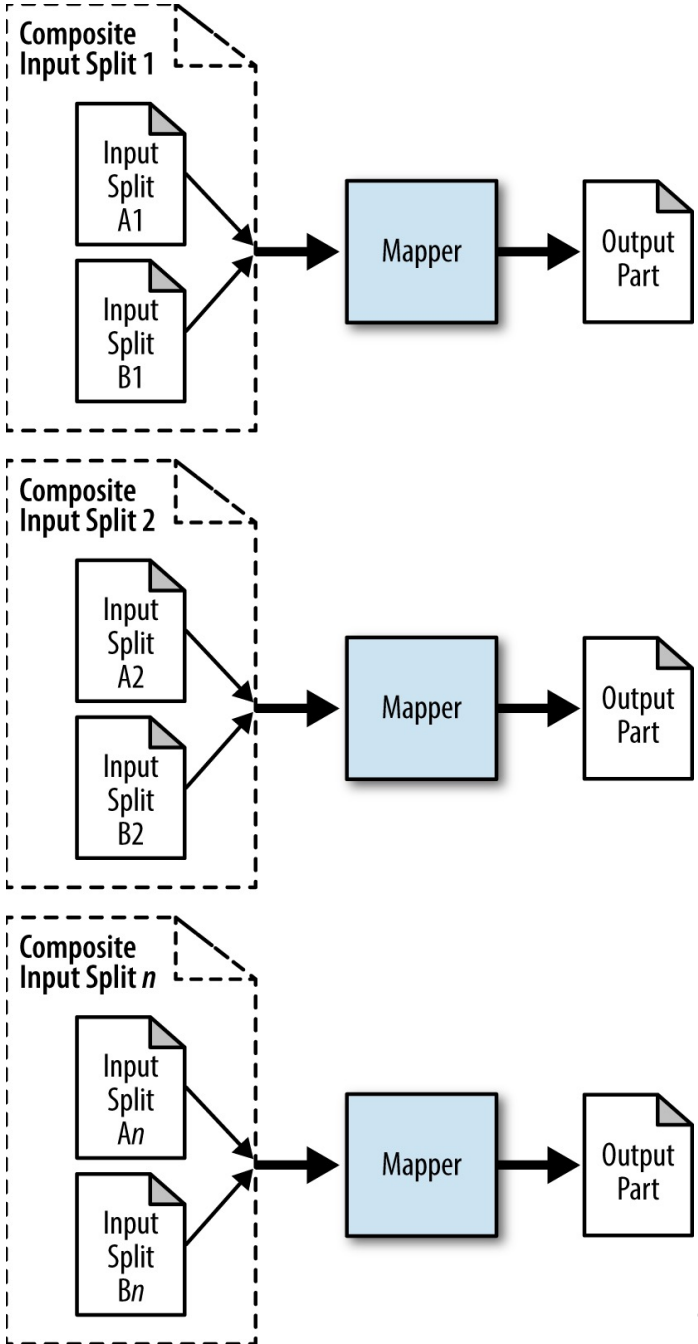


# Replicated Join



# Composite Join



# 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

# 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!
  - Word count: how many unique words in Wikipedia?
  - There's no such thing as "consistent data"
  - Watch out for corner cases
  - Isolate unexpected behavior, bring local




Source: Wikipedia (Japanese rock garden)