



# MapReduce Algorithm Design

Bu eğitim sunumları İstanbul Kalkınma Ajansı'nın 2016 yılı Yenilikçi ve Yaratıcı İstanbul Mali Destek Programı kapsamında yürütülmekte olan TR10/16/YNY/0036 no'lu İstanbul Big Data Eğitim ve Araştırma Merkezi Projesi dahilinde gerçekleştirilmiştir. İçerik ile ilgili tek sorumluluk Bahçeşehir Üniversitesi'ne ait olup İSTKA veya Kalkınma Bakanlığı'nın görüşlerini yansıtmamaktadır.

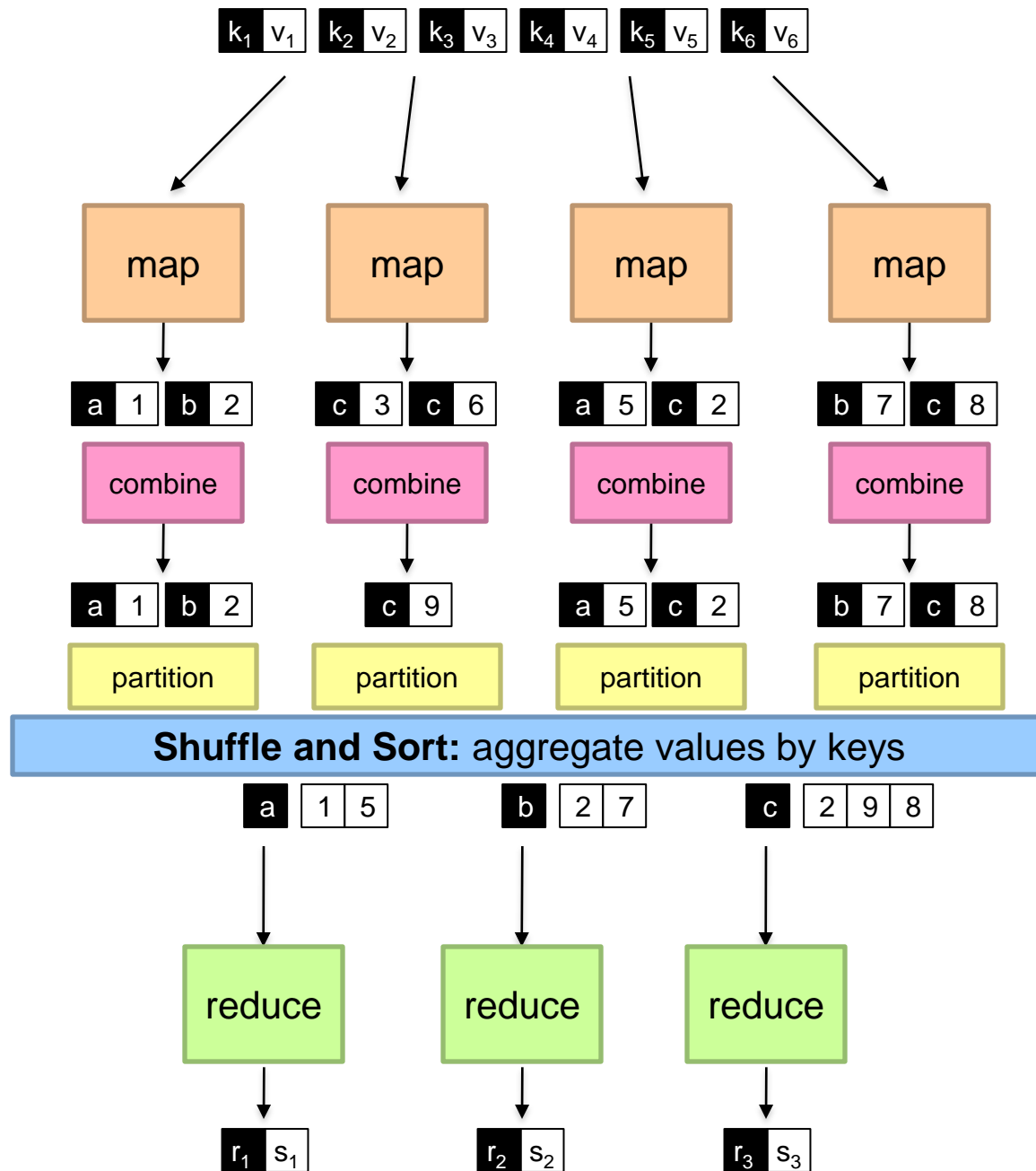
Adopted from Jimmy Lin's slides (at UMD)

# MapReduce: Recap

- Programmers must specify:
  - map**  $(k, v) \rightarrow \langle k', v' \rangle^*$
  - reduce**  $(k', v') \rightarrow \langle k', v' \rangle^*$ 
    - All values with the same key are reduced together
- Optionally, also:
  - partition**  $(k', \text{number of partitions}) \rightarrow \text{partition for } k'$ 
    - Often a simple hash of the key, e.g.,  $\text{hash}(k') \bmod n$
    - Divides up key space for parallel reduce operations
  - combine**  $(k', v') \rightarrow \langle k', v' \rangle^*$ 
    - 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”

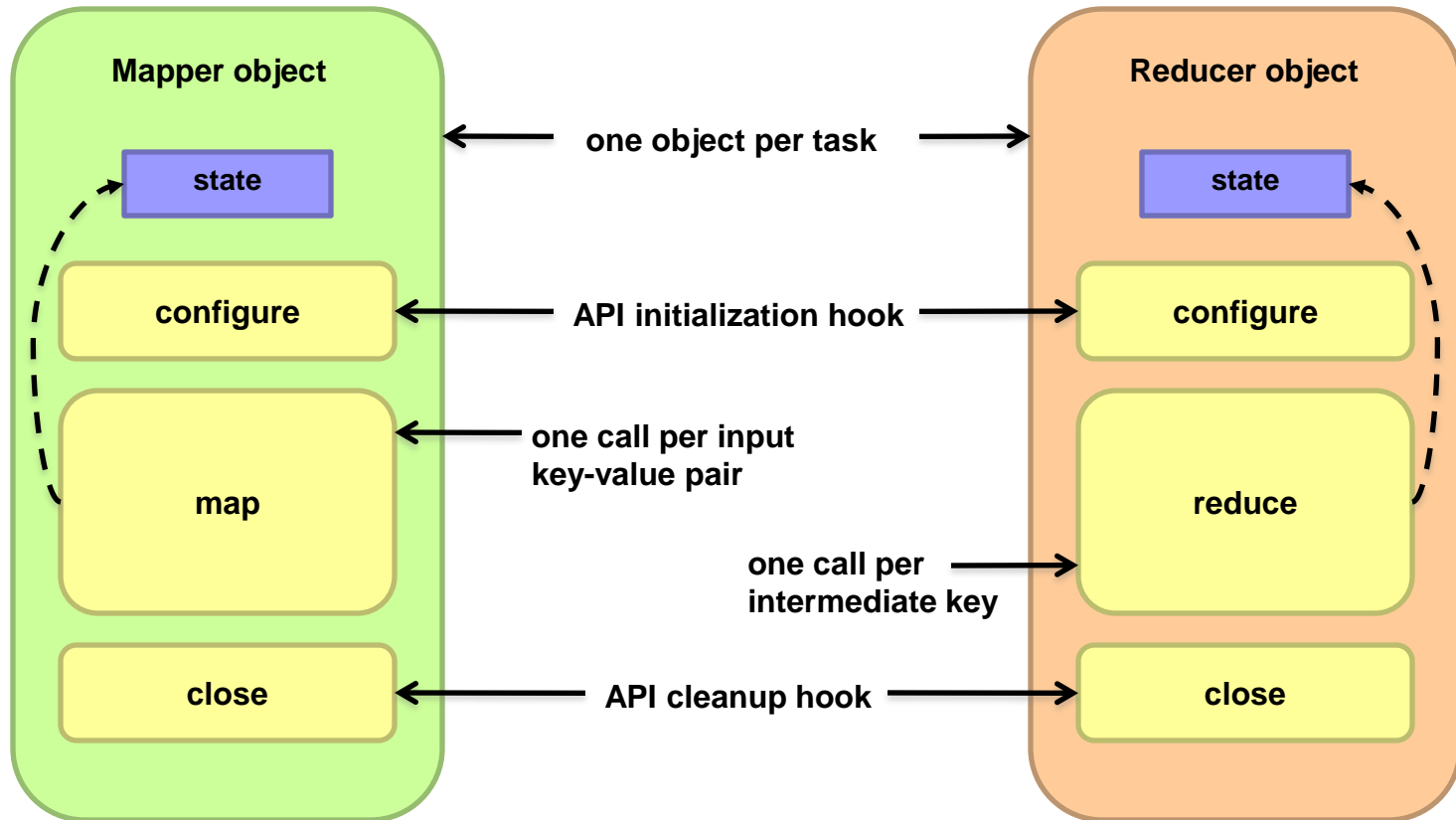
- The execution framework handles everything else...
  - Scheduling: assigns workers to map and reduce tasks
  - “Data distribution”: moves processes to data
  - Synchronization: gathers, sorts, and shuffles intermediate data
  - Errors and faults: detects worker failures and restarts
- Limited control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



# 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 – Why Do This?

- Random sampling
- Randomly select subset of the data (downsample)
- Multiple random subsets for
  - Model generation and testing – cross validation
  - Train on 80%, test on 20%, for 5-fold cross validation
- Anonymizing data
  - Replace PII with a random key

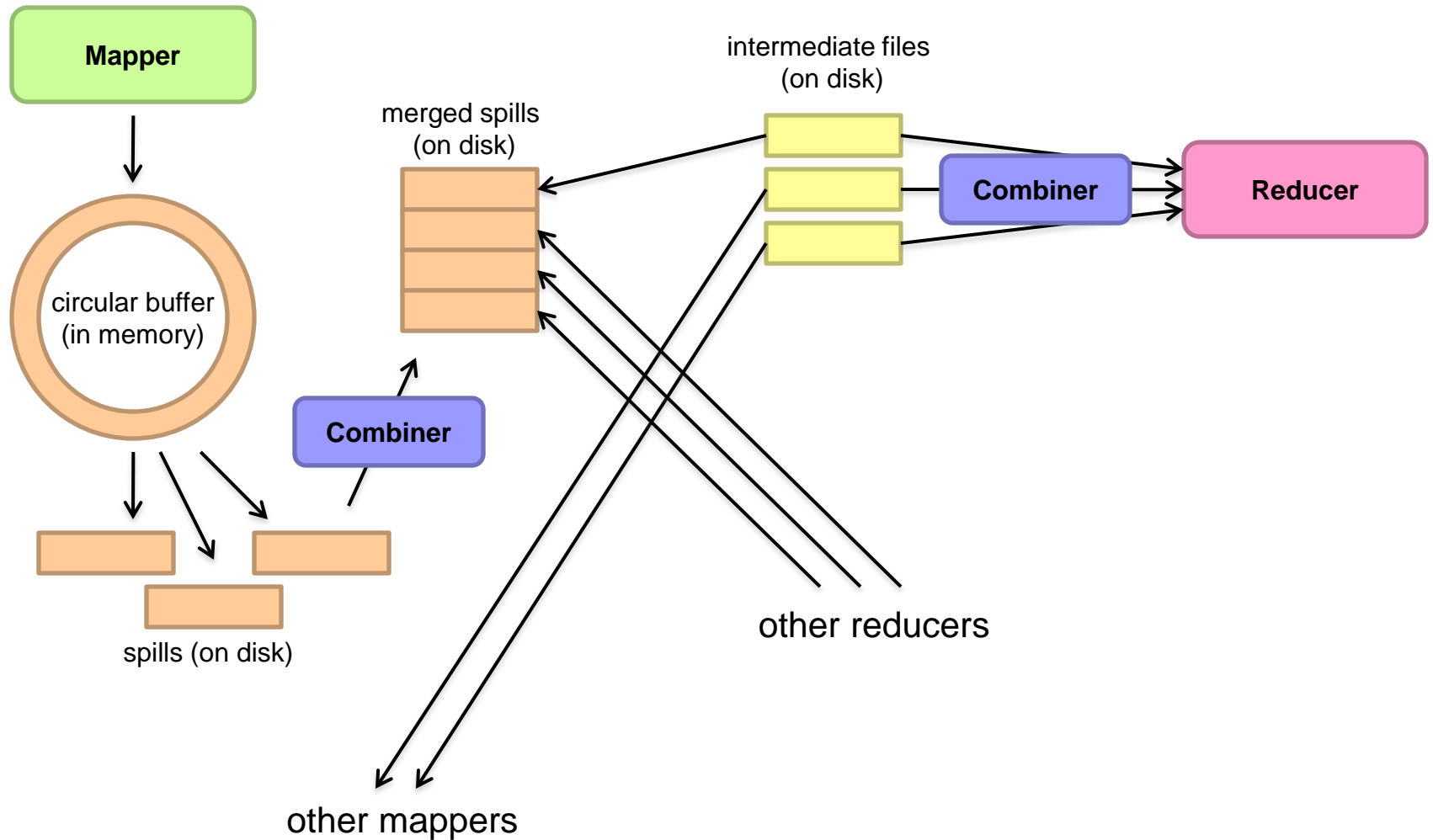
# Shuffle

- Mapper generates a random key for each output
- If you want to reuse a mapper, you could add a partitioner that generates a random partition #
  - Mapper code is then unchanged
- Reducer can sort based on some other random key
  - Further shuffling the data (input order now gone)

# Shuffle and Sort in Hadoop

- Probably the most complex aspect of MapReduce
- Map side
  - Map outputs are buffered in memory in a circular buffer
  - When buffer reaches threshold, contents are “spilled” to disk
  - Spills merged in a single, partitioned file (sorted within each partition): combiner runs during the merges
- Reduce side
  - First, map outputs are copied over to reducer machine
  - “Sort” is a multi-pass merge of map outputs (happens in memory and on disk): combiner runs during the merges
  - Final merge pass goes directly into reducer

# Shuffle and Sort

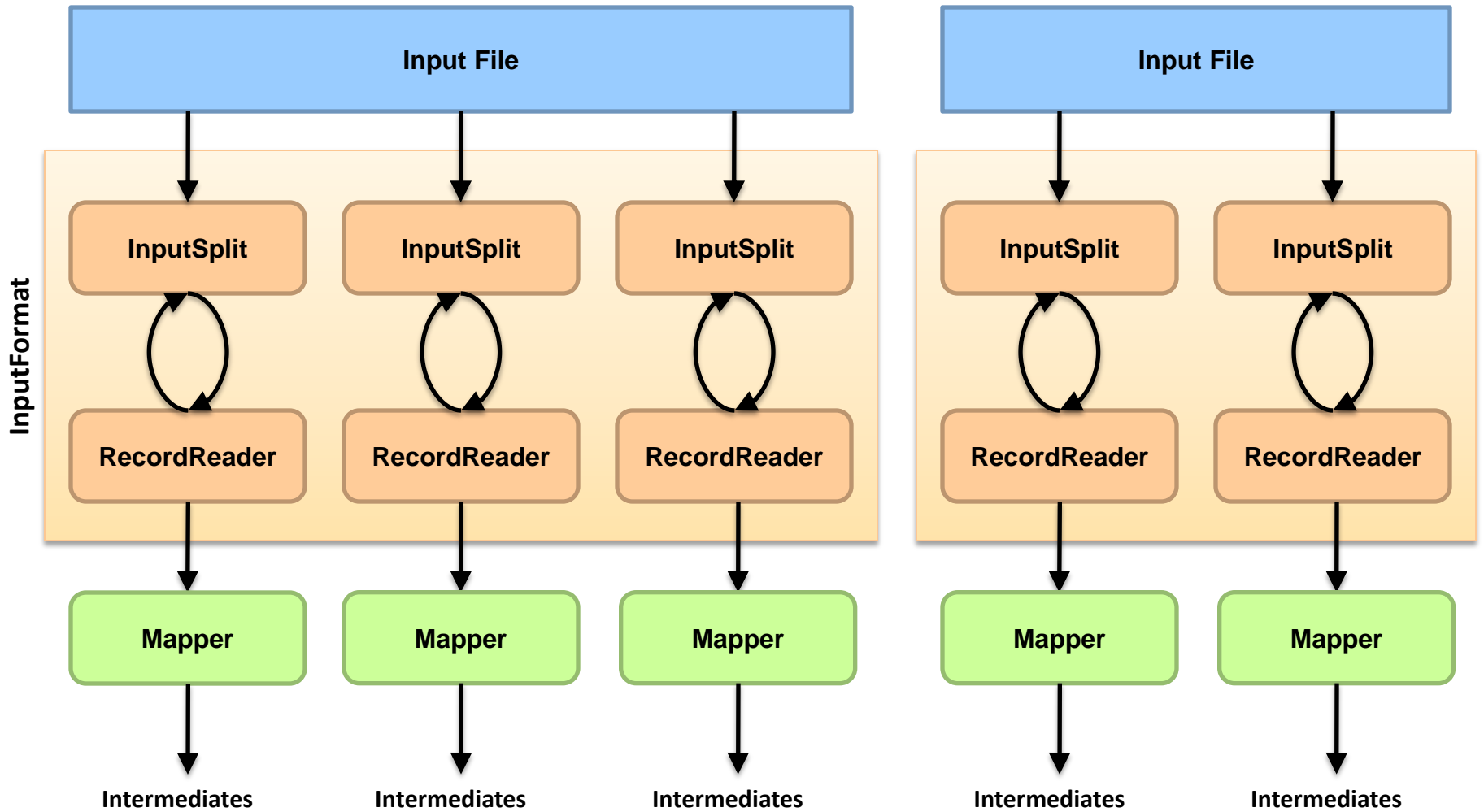


# Anatomy of a Job

- MapReduce program in Hadoop = Hadoop job
  - Jobs are divided into map and reduce tasks
  - An instance of running a task is called a task attempt (occupies a slot)
  - Multiple jobs can be composed into a workflow
- Job submission:
  - Client (i.e., driver program) creates a job, configures it, and submits it to jobtracker
  - That's it! The Hadoop cluster takes over...

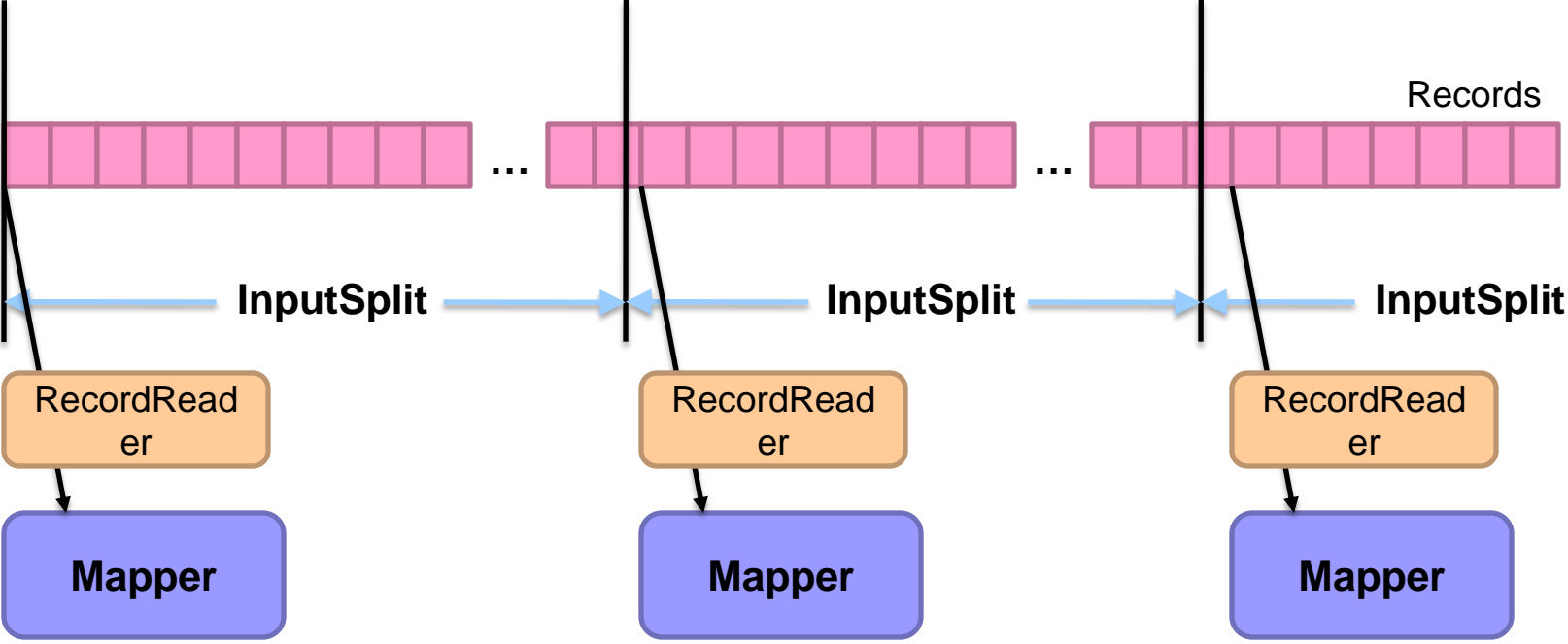
# Anatomy of a Job

- Behind the scenes:
  - Input splits are computed (on client end)
  - Job data (jar, configuration XML) are sent to JobTracker
  - JobTracker puts job data in shared location, enqueues tasks
  - TaskTrackers poll for tasks
  - Off to the races...

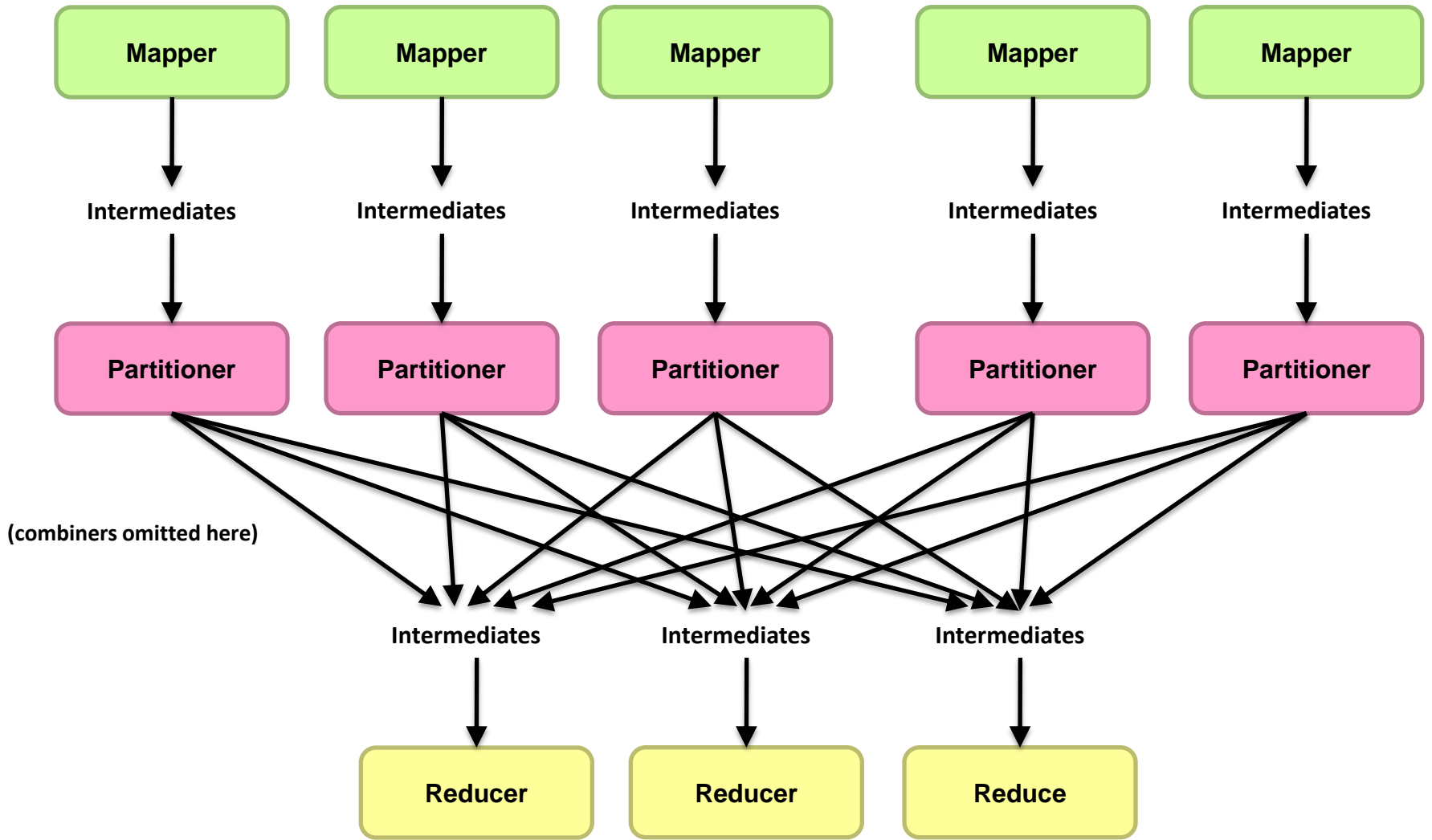


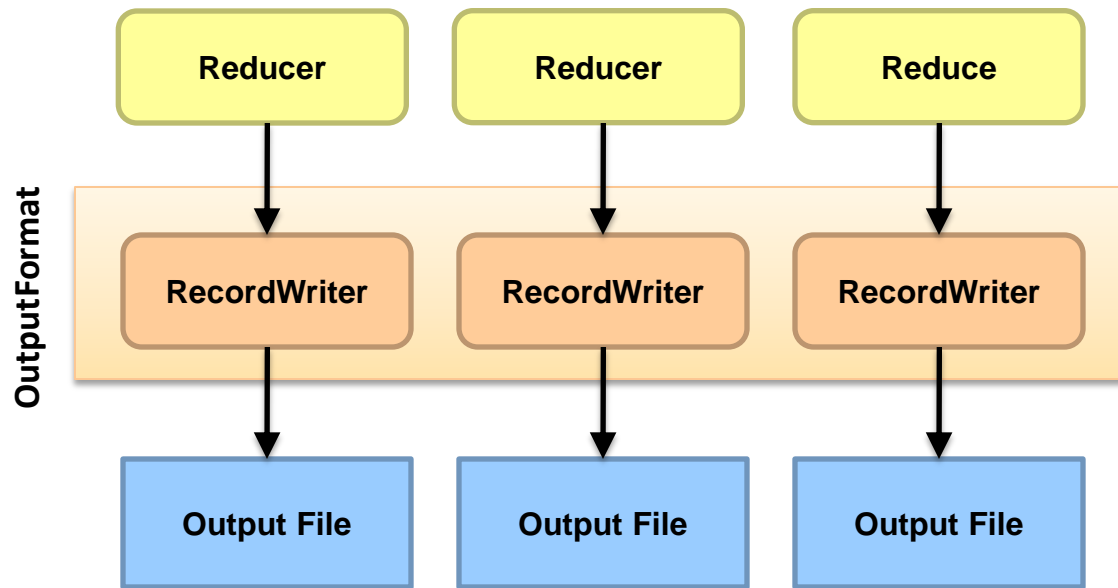
Source: redrawn from a slide by Cloduera, cc-licensed

Client









# Word Count: Baseline

```
1: class MAPPER
2:   method MAP(docid  $a$ , doc  $d$ )
3:     for all term  $t \in$  doc  $d$  do
4:       EMIT(term  $t$ , count 1)

1: class REDUCER
2:   method REDUCE(term  $t$ , counts [ $c_1, c_2, \dots$ ])
3:      $sum \leftarrow 0$ 
4:     for all count  $c \in$  counts [ $c_1, c_2, \dots$ ] do
5:        $sum \leftarrow sum + c$ 
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:      $H \leftarrow$  new ASSOCIATIVEARRAY
4:     for all term  $t \in$  doc  $d$  do
5:        $H\{t\} \leftarrow H\{t\} + 1$ 
6:     for all term  $t \in H$  do
7:       EMIT(term  $t$ , count  $H\{t\}$ )
```

▷ Tally counts for entire document

Are combiners still needed?

# Word Count: Version 2

```
1: class MAPPER
2:   method INITIALIZE
3:      $H \leftarrow \text{new ASSOCIATIVEARRAY}$ 
4:   method MAP(docid  $a$ , doc  $d$ )
5:     for all term  $t \in \text{doc } d$  do
6:        $H\{t\} \leftarrow H\{t\} + 1$ 
7:   method CLOSE
8:     for all term  $t \in H$  do
9:       EMIT(term  $t$ , count  $H\{t\}$ )
```

Key: preserve state across  
input key-value pairs!

▷ Tally counts *across* documents

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

```
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  $[r_1, r_2, \dots]$ )
3:      $sum \leftarrow 0$ 
4:      $cnt \leftarrow 0$ 
5:     for all integer  $r \in$  integers  $[r_1, r_2, \dots]$  do
6:        $sum \leftarrow sum + r$ 
7:        $cnt \leftarrow cnt + 1$ 
8:      $r_{avg} \leftarrow sum / cnt$ 
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 [ $r_1, r_2, \dots$ ])
3:      $sum \leftarrow 0$ 
4:      $cnt \leftarrow 0$ 
5:     for all integer  $r \in$  integers [ $r_1, r_2, \dots$ ] do
6:        $sum \leftarrow sum + r$ 
7:        $cnt \leftarrow cnt + 1$ 
8:     EMIT(string  $t$ , pair ( $sum, cnt$ ))
```

▷ Separate sum and count

```
1: class REDUCER
2:   method REDUCE(string  $t$ , pairs [ $(s_1, c_1), (s_2, c_2) \dots$ ])
3:      $sum \leftarrow 0$ 
4:      $cnt \leftarrow 0$ 
5:     for all pair  $(s, c) \in$  pairs [ $(s_1, c_1), (s_2, c_2) \dots$ ] do
6:        $sum \leftarrow sum + s$ 
7:        $cnt \leftarrow cnt + c$ 
8:      $r_{avg} \leftarrow sum / cnt$ 
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  $[(s_1, c_1), (s_2, c_2) \dots]$ )
3:      $sum \leftarrow 0$ 
4:      $cnt \leftarrow 0$ 
5:     for all pair  $(s, c) \in$  pairs  $[(s_1, c_1), (s_2, c_2) \dots]$  do
6:        $sum \leftarrow sum + s$ 
7:        $cnt \leftarrow cnt + c$ 
8:     EMIT(string  $t$ , pair ( $sum$ ,  $cnt$ ))

1: class REDUCER
2:   method REDUCE(string  $t$ , pairs  $[(s_1, c_1), (s_2, c_2) \dots]$ )
3:      $sum \leftarrow 0$ 
4:      $cnt \leftarrow 0$ 
5:     for all pair  $(s, c) \in$  pairs  $[(s_1, c_1), (s_2, c_2) \dots]$  do
6:        $sum \leftarrow sum + s$ 
7:        $cnt \leftarrow cnt + c$ 
8:      $r_{avg} \leftarrow sum / cnt$ 
9:     EMIT(string  $t$ , pair ( $r_{avg}$ ,  $cnt$ ))
```

Fixed?

# Computing the Mean: Version 4

```
1: class MAPPER
2:   method INITIALIZE
3:      $S \leftarrow \text{new ASSOCIATIVEARRAY}$ 
4:      $C \leftarrow \text{new ASSOCIATIVEARRAY}$ 
5:   method MAP(string  $t$ , integer  $r$ )
6:      $S\{t\} \leftarrow S\{t\} + r$ 
7:      $C\{t\} \leftarrow C\{t\} + 1$ 
8:   method CLOSE
9:     for all term  $t \in S$  do
10:       EMIT(term  $t$ , pair ( $S\{t\}$ ,  $C\{t\}$ ))
```

Are combinators still needed?

# Algorithm Design: Running Example

- Term co-occurrence matrix for a text collection
  - $M = N \times N$  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  $w \in \text{doc } d$  do
4:       for all term  $u \in \text{NEIGHBORS}(w)$  do
5:         EMIT(pair  $(w, u)$ , count 1)      ▷ Emit count for each co-occurrence

1: class REDUCER
2:   method REDUCE(pair  $p$ , counts  $[c_1, c_2, \dots]$ )
3:      $s \leftarrow 0$ 
4:     for all count  $c \in \text{counts } [c_1, c_2, \dots]$  do
5:        $s \leftarrow s + c$                   ▷ 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



# 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