



Data-Intensive Distributed Computing

CS 451/651 431/631 (Winter 2018)

Part I: MapReduce Algorithm Design (4/4)
January 16, 2018

Jimmy Lin
David R. Cheriton School of Computer Science
University of Waterloo

These slides are available at <http://lintool.github.io/bigdata-2018w/>



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





Source: Wikipedia (Japanese rock garden)

Perfect X

What's the point?

More details: Lee et al. The Unified Logging Infrastructure for Data Analytics at Twitter.
PVLDB, 5(12):1771-1780, 2012.

MapReduce Algorithm Design

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

Toward “design patterns”



MapReduce

MapReduce

Programmer specifies four functions:

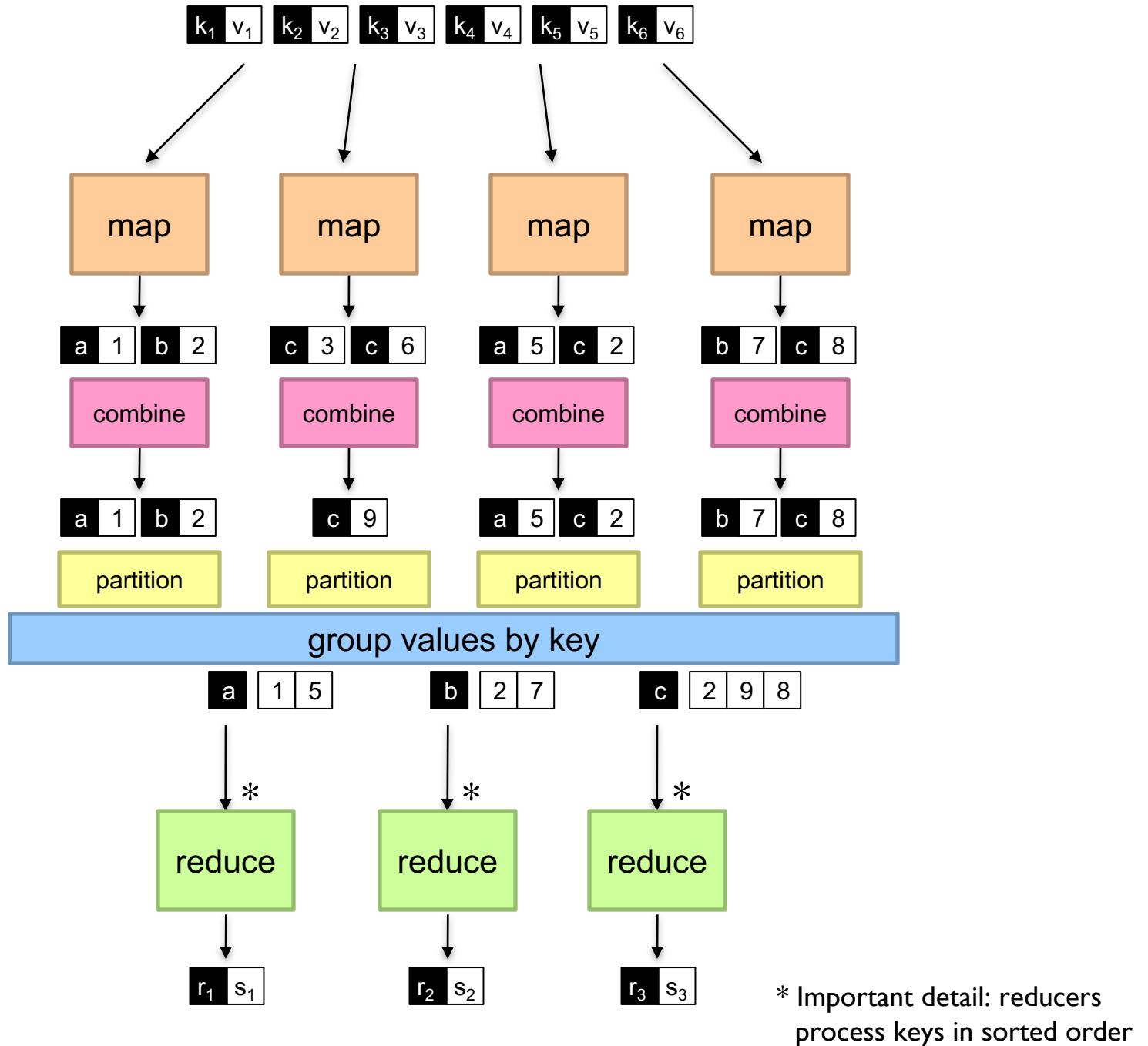
map $(k_1, v_1) \rightarrow \text{List}[(k_2, v_2)]$
reduce $(k_2, \text{List}[v_2]) \rightarrow \text{List}[(k_3, v_3)]$

All values with the same key are sent to the same reducer

partition $(k', p) \rightarrow 0 \dots p-1$
Often a simple hash of the key, e.g., $\text{hash}(k') \bmod n$
Divides up key space for parallel reduce operations

combine $(k_2, \text{List}[v_2]) \rightarrow \text{List}[(k_2, v_2)]$
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”

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

But...

You have 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

Preserving state in mappers and reducers

Capture dependencies across multiple keys and values

Cleverly-constructed data structures

Bring partial results together

Define custom sort order of intermediate keys

Control order in which reducers process keys

Two Practical Tips

Avoid object creation
(Relatively) 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

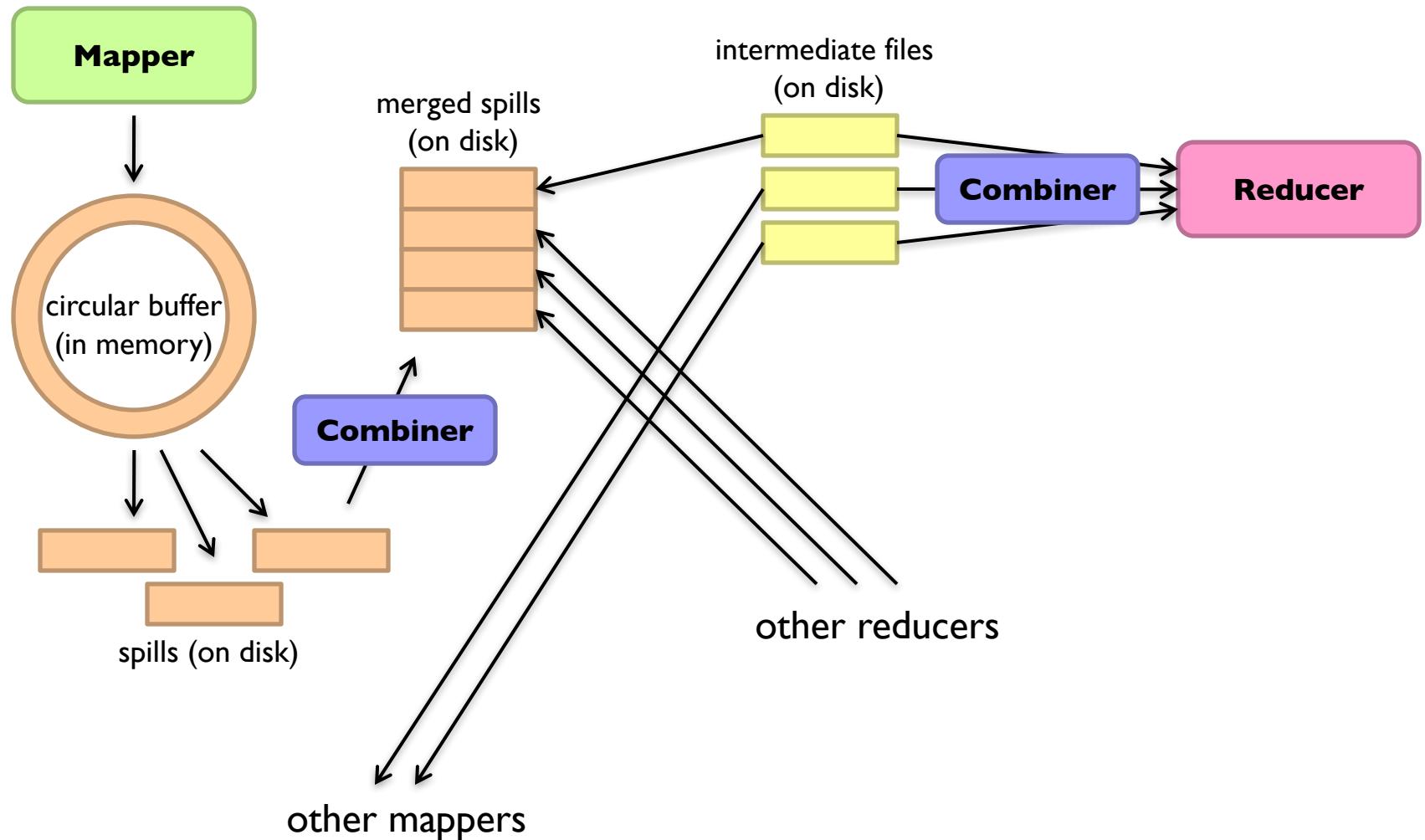
Communication kills performance

Thus... avoid communication!

Reduce intermediate data via local aggregation

Combiners can help

Distributed Group By in MapReduce



Word Count: Baseline

```
class Mapper {  
    def map(key: Long, value: String) = {  
        for (word <- tokenize(value)) {  
            emit(word, 1)  
        }  
    }  
}  
  
class Reducer {  
    def reduce(key: String, values: Iterable[Int]) = {  
        var sum = 0  
        for (value <- values) {  
            sum += value  
        }  
        emit(key, sum)  
    }  
}
```

What's the impact of combiners?

Word Count: Mapper Histogram

```
class Mapper {  
    def map(key: Long, value: String) = {  
        val counts = new Map()  
        for (word <- tokenize(value)) {  
            counts(word) += 1  
        }  
  
        for ((k, v) <- counts) {  
            emit(k, v)  
        }  
    }  
}
```

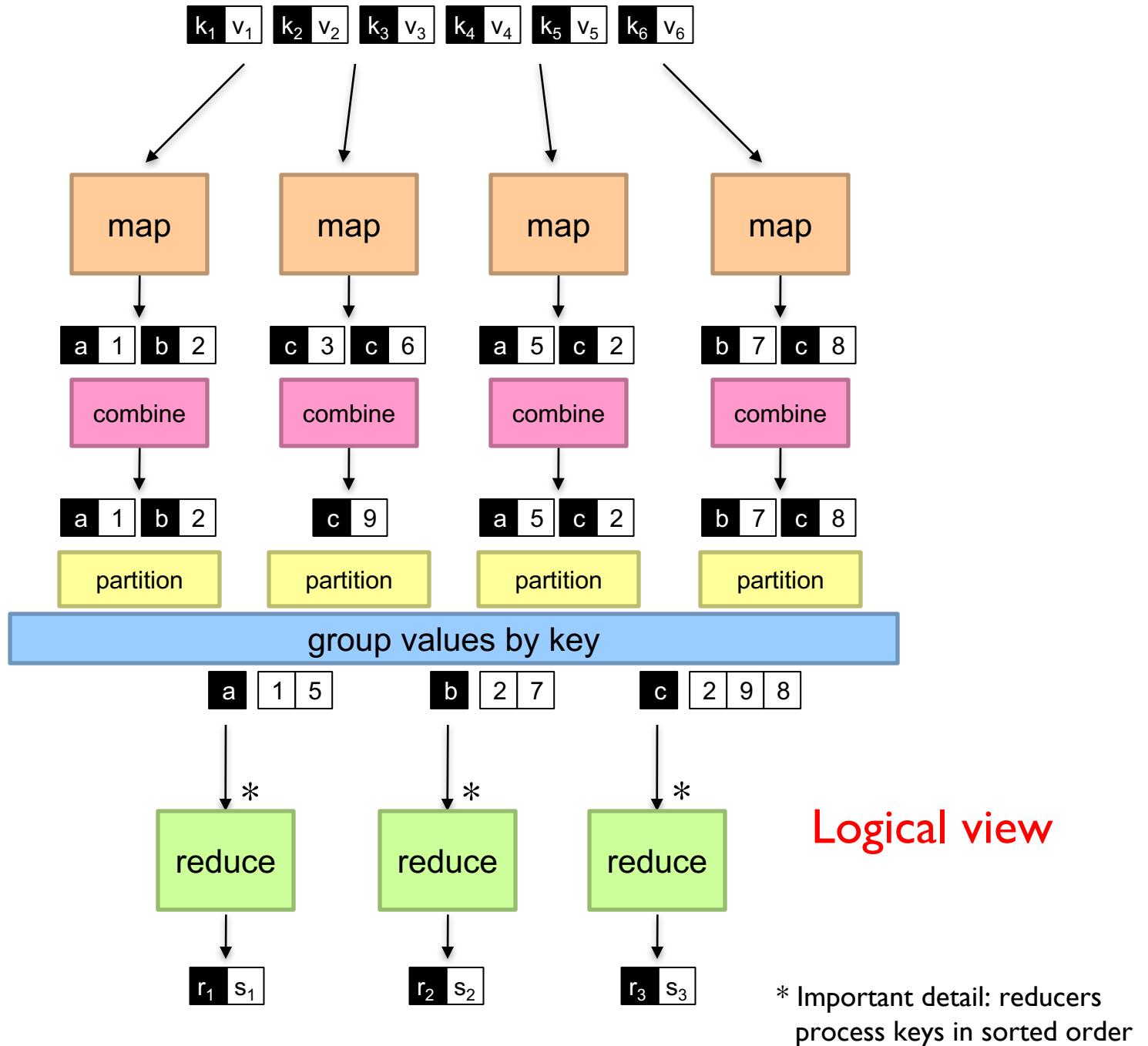
Are combiners still needed?

Performance

Word count on 10% sample of Wikipedia (on Altiscale)

	Running Time	# Pairs
Baseline	~140s	246m
Histogram	~140s	203m

Can we do even better?



MapReduce API*

`Mapper<Kin, Vin, Kout, Vout>`

`void setup(Mapper.Context context)`

Called once at the start of the task

`void map(Kin key, Vin value, Mapper.Context context)`

Called once for each key/value pair in the input split

`void cleanup(Mapper.Context context)`

Called once at the end of the task

`Reducer<Kin, Vin, Kout, Vout>/Combiner<Kin, Vin, Kout, Vout>`

`void setup(Reducer.Context context)`

Called once at the start of the task

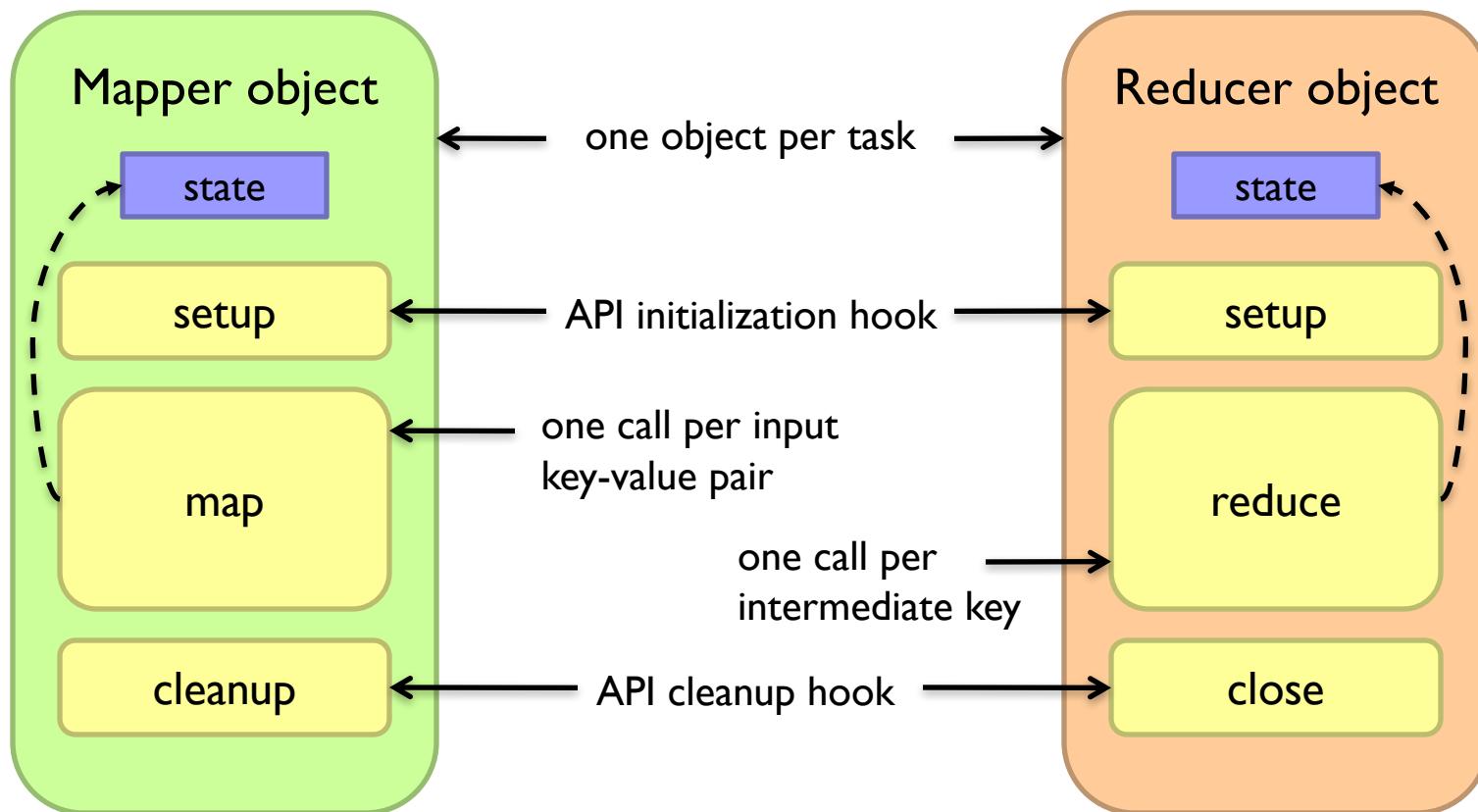
`void reduce(Kin key, Iterable<Vin> values, Reducer.Context context)`

Called once for each key

`void cleanup(Reducer.Context context)`

Called once at the end of the task

Preserving State



Pseudo-Code

```
class Mapper {  
    def setup() = {  
        ...  
    }  
  
    def map(key: Long, value: String) = {  
        ...  
    }  
  
    def cleanup() = {  
        ...  
    }  
}
```

Word Count: Preserving State

```
class Mapper {  
    val counts = new Map()  
  
    def map(key: Long, value: String) = {  
        for (word <- tokenize(value)) {  
            counts(word) += 1  
        }  
    }  
  
    def cleanup() = {  
        for ((k, v) <- counts) {  
            emit(k, v)  
        }  
    }  
}
```

Key idea: 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

Performance

Word count on 10% sample of Wikipedia (on Altiscale)

	Running Time	# Pairs
Baseline	~140s	246m
Histogram	~140s	203m
IMC	~80s	5.5m

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 integers associated with the same key

Computing the Mean: Version I

```
class Mapper {  
    def map(key: String, value: Int) = {  
        emit(key, value)  
    }  
}  
  
class Reducer {  
    def reduce(key: String, values: Iterable[Int]) {  
        for (value <- values) {  
            sum += value  
            cnt += 1  
        }  
        emit(key, sum/cnt)  
    }  
}
```

Why can't we use reducer as combiner?

Computing the Mean: Version 2

```
class Mapper {  
    def map(key: String, value: Int) =  
        context.write(key, value)  
}  
class Combiner {  
    def reduce(key: String, values: Iterable[Int]) = {  
        for (value <- values) {  
            sum += value  
            cnt += 1  
        }  
        emit(key, (sum, cnt))  
    }  
}  
class Reducer {  
    def reduce(key: String, values: Iterable[Pair]) = {  
        for ((s, c) <- values) {  
            sum += s  
            cnt += c  
        }  
        emit(key, sum/cnt)  
    }  
}
```

Why doesn't this work?

Computing the Mean: Version 3

```
class Mapper {  
    def map(key: String, value: Int) =  
        context.write(key, (value, 1))  
}  
class Combiner {  
    def reduce(key: String, values: Iterable[Pair]) = {  
        for ((s, c) <- values) {  
            sum += s  
            cnt += c  
        }  
        emit(key, (sum, cnt))  
    }  
}  
class Reducer {  
    def reduce(key: String, values: Iterable[Pair]) = {  
        for ((s, c) <- values) {  
            sum += s  
            cnt += c  
        }  
        emit(key, sum/cnt)  
    }  
}
```

Fixed?

Computing the Mean: Version 4

```
class Mapper {
    val sums = new Map()
    val counts = new Map()

    def map(key: String, value: Int) = {
        sums(key) += value
        counts(key) += 1
    }

    def cleanup() = {
        for (key <- counts.keys) {
            context.write(key, (sums(key), counts(key)))
        }
    }
}
```

Are combiners still needed?

Performance

200m integers across three char keys

	Java	Scala
V1	~120s	~120s
V3	~90s	~120s
V4	~60s	~90s (default HashMap) ~70s (optimized HashMap)

MapReduce API*

`Mapper<Kin, Vin, Kout, Vout>`

`void setup(Mapper.Context context)`

Called once at the start of the task

`void map(Kin key, Vin value, Mapper.Context context)`

Called once for each key/value pair in the input split

`void cleanup(Mapper.Context context)`

Called once at the end of the task

`Reducer<Kin, Vin, Kout, Vout>/Combiner<Kin, Vin, Kout, Vout>`

`void setup(Reducer.Context context)`

Called once at the start of the task

`void reduce(Kin key, Iterable<Vin> values, Reducer.Context context)`

Called once for each key

`void cleanup(Reducer.Context context)`

Called once at the end of the task

Algorithm Design: Running Example

Term co-occurrence matrix for a text collection

$M = N \times N$ matrix ($N = \text{vocabulary size}$)

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

Applications in lots of other domains

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 {  
    def map(key: Long, value: String) = {  
        for (u <- tokenize(value)) {  
            for (v <- neighbors(u)) {  
                emit((u, v), 1)  
            }  
        }  
    }  
}  
  
class Reducer {  
    def reduce(key: Pair, values: Iterable[Int]) = {  
        var sum = 0  
        for (value <- values) {  
            sum += value  
        }  
        emit(key, sum)  
    }  
}
```

Pairs: Pseudo-Code

One more thing...

```
class Partitioner {  
    def getPartition(key: Pair, value: Int, numTasks: Int): Int = {  
        return key.left % numTasks  
    }  
}
```

“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

$(a, b) \rightarrow 1$

$(a, c) \rightarrow 2$

$(a, d) \rightarrow 5 \quad a \rightarrow \{ b: 1, c: 2, d: 5, e: 3, f: 2 \}$

$(a, e) \rightarrow 3$

$(a, f) \rightarrow 2$

Each mapper takes a sentence:

Generate all co-occurring term pairs

For each term, emit $a \rightarrow \{ b: \text{count}_b, c: \text{count}_c, d: \text{count}_d \dots \}$

Reducers perform element-wise sum of associative arrays

$a \rightarrow \{ b: 1, \quad d: 5, e: 3 \}$

$+ \quad a \rightarrow \{ b: 1, c: 2, d: 2, \quad f: 2 \}$

$\hline a \rightarrow \{ b: 2, c: 2, d: 7, e: 3, f: 2 \}$

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

Stripes: Pseudo-Code

```
class Mapper {
    def map(key: Long, value: String) = {
        for (u <- tokenize(value)) {
            val map = new Map()
            for (v <- neighbors(u)) {
                map(v) += 1
            }
            emit(u, map)
        }
    }
}

class Reducer {
    def reduce(key: String, values: Iterable[Map]) = {
        val map = new Map()
        for (value <- values) {
            map += value
        }
        emit(key, map)
    }
}
```

$a \rightarrow \{ b: 1, c: 2, d: 5, e: 3, f: 2 \}$

$a \rightarrow \{ b: 1, d: 5, e: 3 \}$

$+ a \rightarrow \{ b: 1, c: 2, d: 2, f: 2 \}$

$a \rightarrow \{ b: 2, c: 2, d: 7, e: 3, f: 2 \}$

“Stripes” Analysis

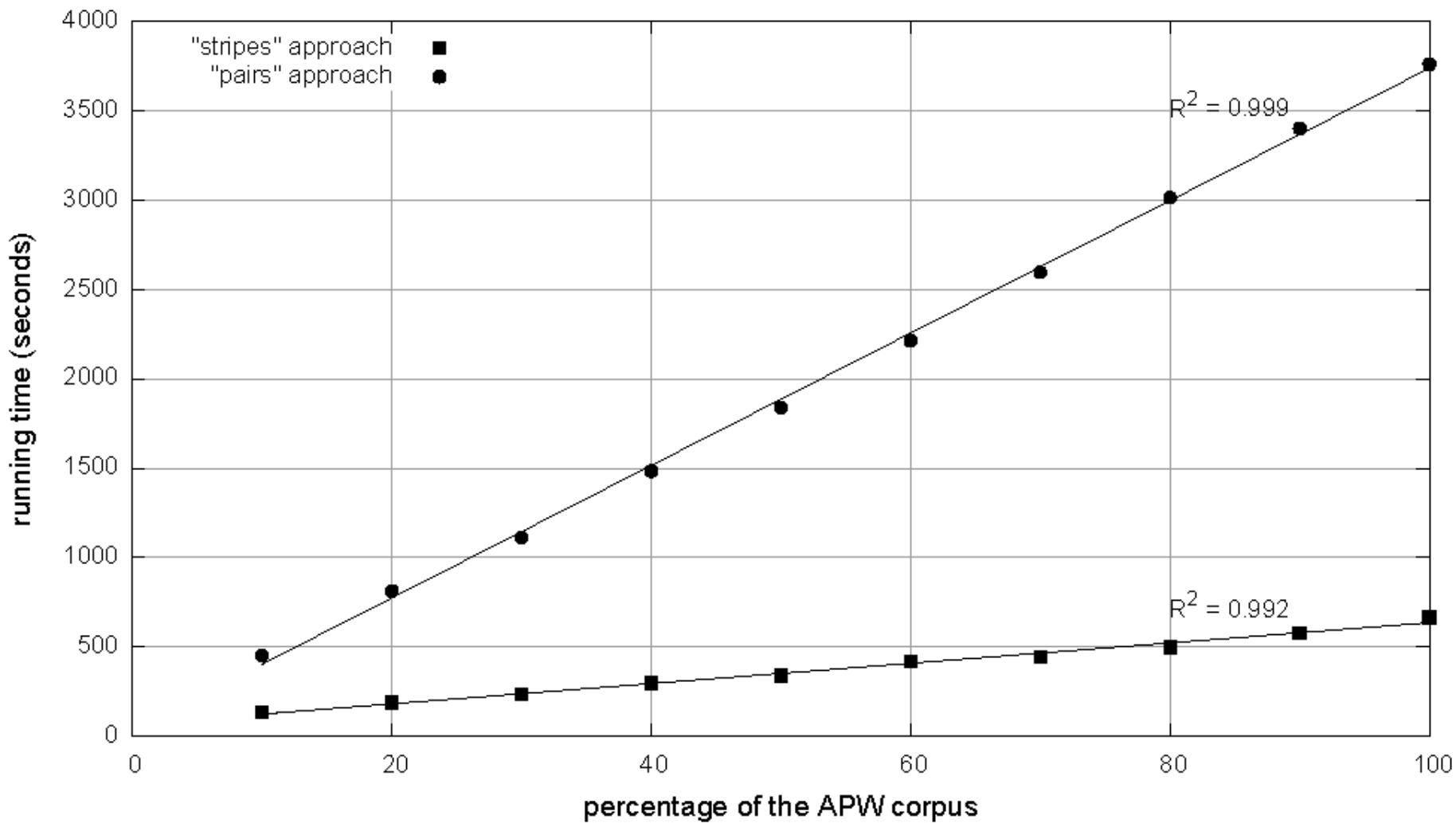
Advantages

- Far less sorting and shuffling of key-value pairs
- Can make better use of combiners

Disadvantages

- More difficult to implement
- Underlying object more heavyweight
- Overhead associated with data structure manipulations
- Fundamental limitation in terms of size of event space

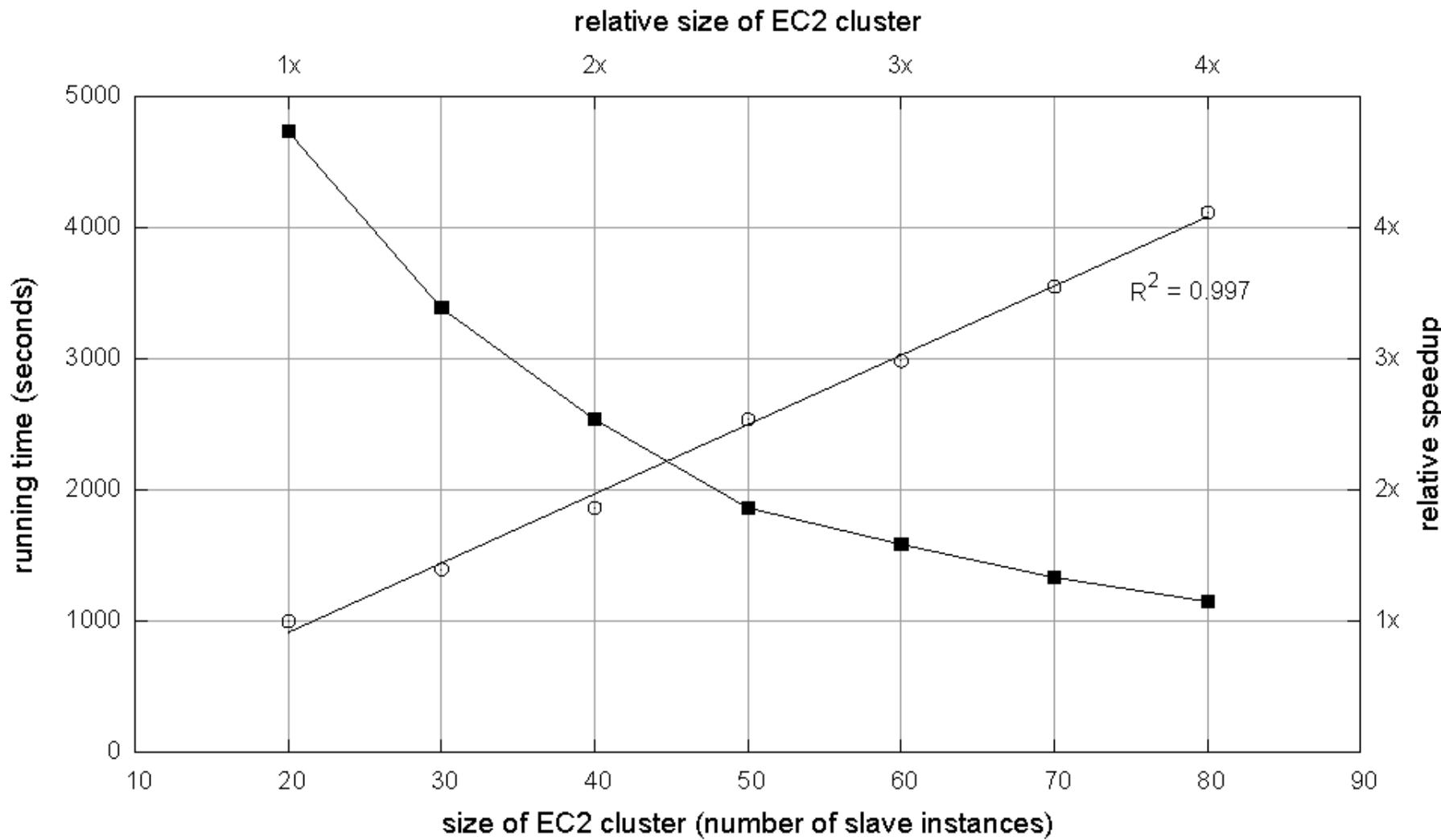
Comparison of "pairs" vs. "stripes" for computing word co-occurrence matrices



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



Stripes >> Pairs?

Important tradeoffs

Developer code vs. framework

CPU vs. RAM vs. disk vs. network

Number of key-value pairs: sorting and shuffling data across the network

Size and complexity of each key-value pair: de/serialization overhead

Cache locality and the cost of manipulating data structures

Additional issues

Opportunities for local aggregation (combining)

Load imbalance

Tradeoffs

Pairs:

- Generates a *lot* more key-value pairs
- Less combining opportunities
- More sorting and shuffling
- Simple aggregation at reduce

Stripes:

- Generates fewer key-value pairs
- More opportunities for combining
- Less sorting and shuffling
- More complex (slower) aggregation at reduce

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”

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

Easy!

One pass to compute $(a, *)$

Another pass to directly compute $f(B|A)$

$f(B|A)$: “Pairs”

What’s the issue?

Computing relative frequencies requires marginal counts
But the marginal cannot be computed until you see all counts
Buffering is a bad idea!

Solution:

What if we could get the marginal count to arrive at the reducer first?

$f(B|A)$: “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:

Emit extra $(a, *)$ for every b_n in mapper

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

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

Hold state in reducer across different key-value pairs

“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

Additional optimization

Apply in-memory combining pattern to accumulate marginal counts

Synchronization: Pairs vs. Stripes

Approach 1: turn synchronization into an ordering problem

Sort keys into correct order of computation

Partition key space so each reducer receives appropriate set of partial results

Hold state in reducer across multiple key-value pairs to perform computation

Illustrated by the “pairs” approach

Approach 2: 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 we want to sort value also?
E.g., $k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r) \dots$

Secondary Sorting: Solutions

Solution 1

Buffer values in memory, then sort
Why is this a bad idea?

Solution 2

“Value-to-key conversion” : form composite intermediate key, (k, v_i)
Let the 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

Preserving state in mappers and reducers

Capture dependencies across multiple keys and values

Cleverly-constructed data structures

Bring partial results together

Define custom sort order of intermediate keys

Control order in which reducers process keys

Issues and Tradeoffs

Important tradeoffs

Developer code vs. framework

CPU vs. RAM vs. disk vs. network

Number of key-value pairs: sorting and shuffling data across the network

Size and complexity of each key-value pair: de/serialization overhead

Cache locality and the cost of manipulating data structures

Additional issues

Opportunities for local aggregation (combining)

Local imbalance

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

A photograph of a traditional Japanese rock garden. In the foreground, a gravel path is raked into fine, parallel lines. Several large, dark, irregular stones are scattered across the garden. A small, shallow pond is visible in the middle ground, surrounded by more stones and some low-lying green plants. In the background, there are more stones, some small trees, and the wooden buildings of a residence with tiled roofs.

Questions?