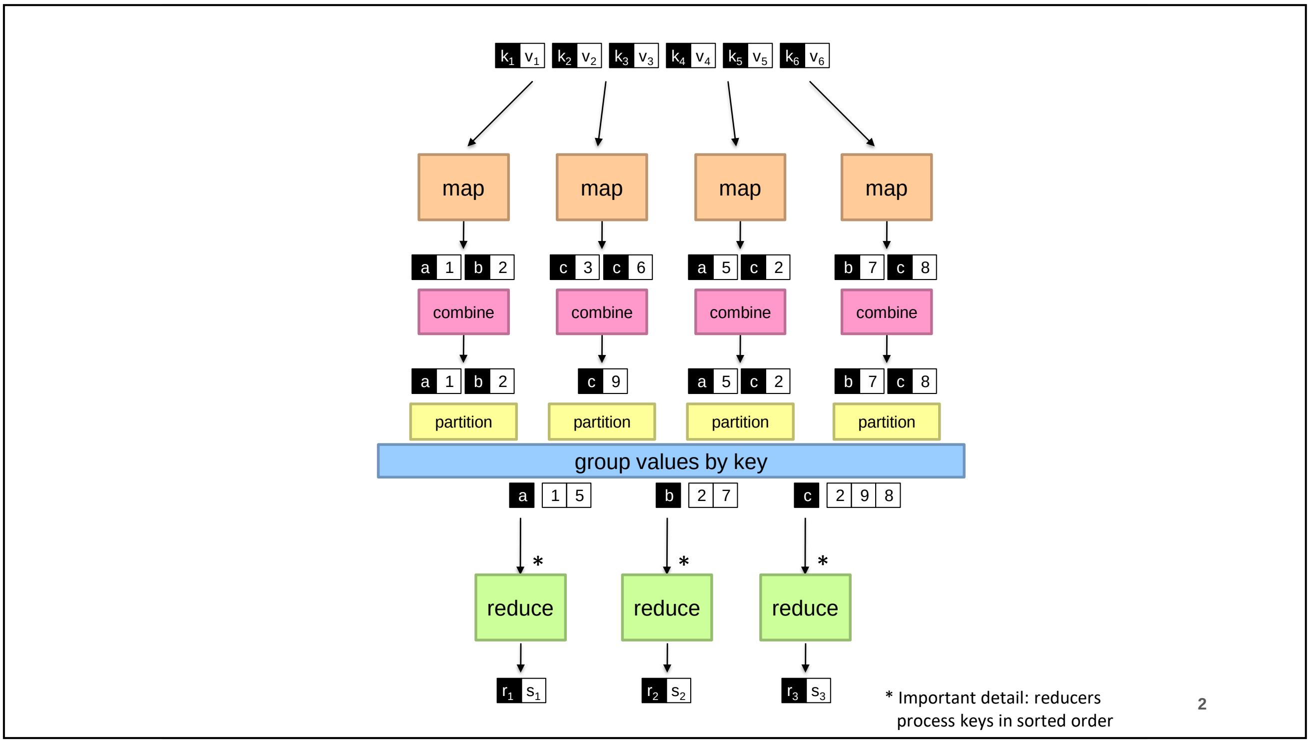
Data-Intensive Distributed Computing

431/451/631/651 (Fall 2021)

Part 1: MapReduce Algorithm Design (3/3)

Ali Abedi

These slides are available at https://www.student.cs.uwaterloo.ca/~cs451/



We now talk more about combiner design

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

Combiner Design

Combiners and reducers share same method signature

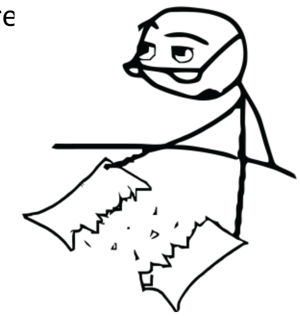
Sometimes, reducers can serve as combiners

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

4

Computing the Mean: Version 1

```
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)
  }
}
```

(a, 7)
(a,18)
(c, 4)
(b,1)
(c, 10)
(a, 3)
…

Why can’t we use reducer as combiner?

AVG (4, 4, 2, 2, 2) != AVG (AVG (4, 4), AVG(2, 2, 2)) = 3

5

No, because we cannot take partial averages! The math will be wrong

"I hope you don't mind, but the size of the file I just sent you was 198,753 gigs."

Computing the Mean: Version 2

```
class Mapper {
  def map(key: String, value: Int) =
    emit(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)
  }
}
```

(a, 7)
(a,18)
(c, 4)
(b,1)
(c, 10)
(a, 3)
…

Why doesn't this work?

7

The input to reducer might be coming from mapper or combiner however the output of mapper and combiner differ. This implementation assumes that combiners always run but this is not true.

Computing the Mean: Version 3

```
class Mapper {
  def map(key: String, value: Int) =
    emit(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)
  }
}
```


The problem is fixed by modifying the output of mapper to match the output of combiner.

Performance

200m integers across three char keys

		Time
V1	Baseline	~120s
V3	+ Combiner	~90s

(a, 7)
(a,18)
(c, 4)
(b,1)
(c, 10)
(a, 3)
…

9

Using combiner significantly improves the performance.

In-Mapper Combiner

10

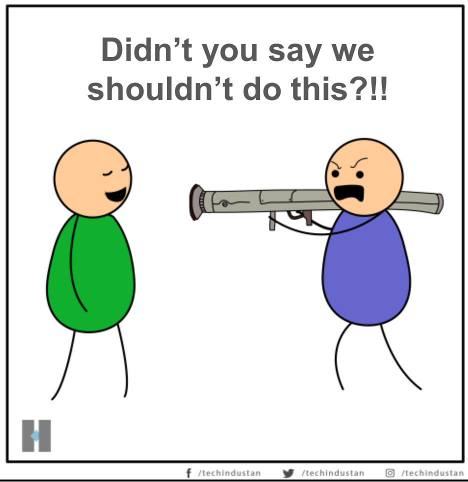
Word count with in-mapper combiner

```
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!



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

12

In-mapper is faster than regular combiners because it is done in memory, in contrast with regular combining which is a disk to disk operation.

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) {
      emit(key, (sums(key), counts(key)))
    }
  }
}
```

(a, 7)
(a,18)
(c, 4)
(b,1)
(c, 10)
(a, 3)
…

13

Using IMC to improve the performance of computing the mean.

Performance

200m integers across three char keys

		Time
V1	Baseline	~120s
V3	+ Combiner	~90s
V4	+ IMC	~60s

14

Algorithm Design

Term co-occurrence

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

Applications in lots of other domains

How many times two words co-occur?

Two approaches:

Pairs

Stripes

17

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]) = {
    for (value <- values) {
      sum += value
   }
   emit(key, sum)
  }
}
```

“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

20

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

21

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)
    }
  }
}
```

a → { b: 1, c: 2, d: 5, e: 3, f: 2 }

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

	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 }

“Stripes” Analysis

Advantages

Far less sorting and shuffling of key-value pairs

Can make better use of combiners

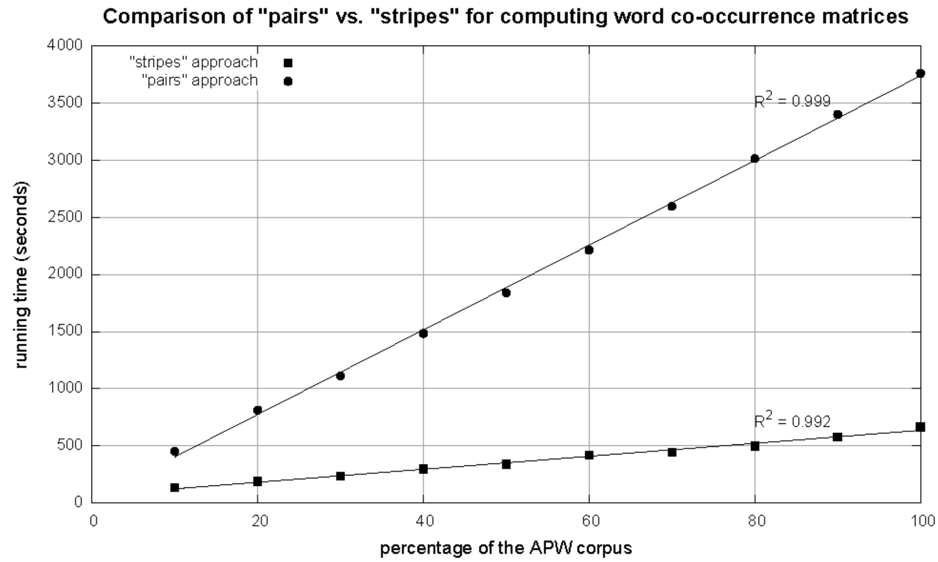
Disadvantages

More difficult to implement

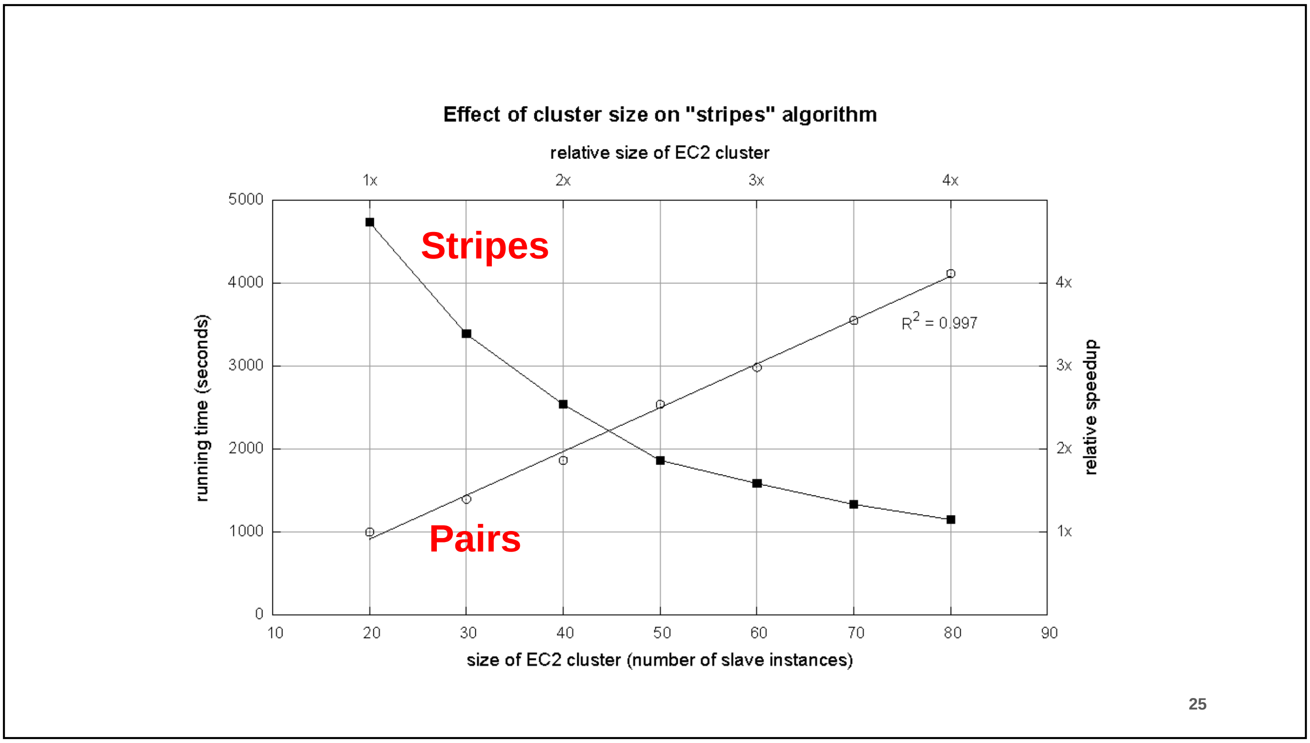
Underlying object more heavyweight

Overhead associated with data structure manipulations

Fundamental limitation in terms of size of event space



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)



There is a tradeoff at work here! Pairs will operate better than Stripes in a smaller cluster because communication is fairly limited anyways (less machines means that each machine does more of the work and that results can be aggregated more locally), and thus, the overhead of Stripes causes it to perform worse. However, as the cluster grows, communication increases, and Stripes start to shine

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 → {b_1:3, b_2 :12, b_3 :7, b_4 :1, … }

Easy!

One pass to compute (a, *)

Another pass to directly compute f(B|A)

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

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, *) → 32 Reducer holds this value in memory

(a, b_1) → 3	⇒	(a, b_1) → 3 / 32
(a, b_2) → 12		(a, b_2) → 12 / 32
(a, b_3) → 7		(a, b_3) → 7 / 32
(a, b_4) → 1		(a, b_4) → 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

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

Pairs: Pseudo-Code

One more thing…

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

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