

CENG 595

Distributed Data Processing and Analysis

«Big Data»

MapReduce to Spark (2)

Spring 2017

Erdogan Dogdu

Çankaya University

Department of Computer Engineering

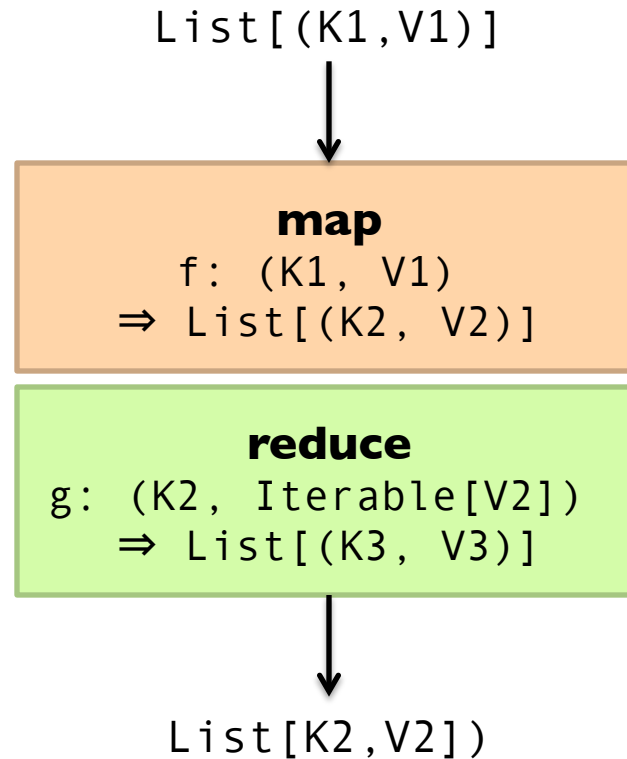
Slides adapted from Jimmy Lin's class
<https://lintool.github.io/bigdata-2016w/syllabus.html>

An aerial photograph of a large industrial facility, likely a datacenter, situated in a rural area. The facility consists of several large, white, rectangular buildings with flat roofs, arranged in a grid-like pattern. In the foreground, there is a large parking lot filled with many white semi-trailers. To the right, a large building with a complex roof structure, possibly a cooling or power plant, is visible. The surrounding landscape is a mix of green fields and brown, tilled soil. In the background, a line of trees and distant hills are visible under a sky with a bright, low sun, creating a warm, orange glow.

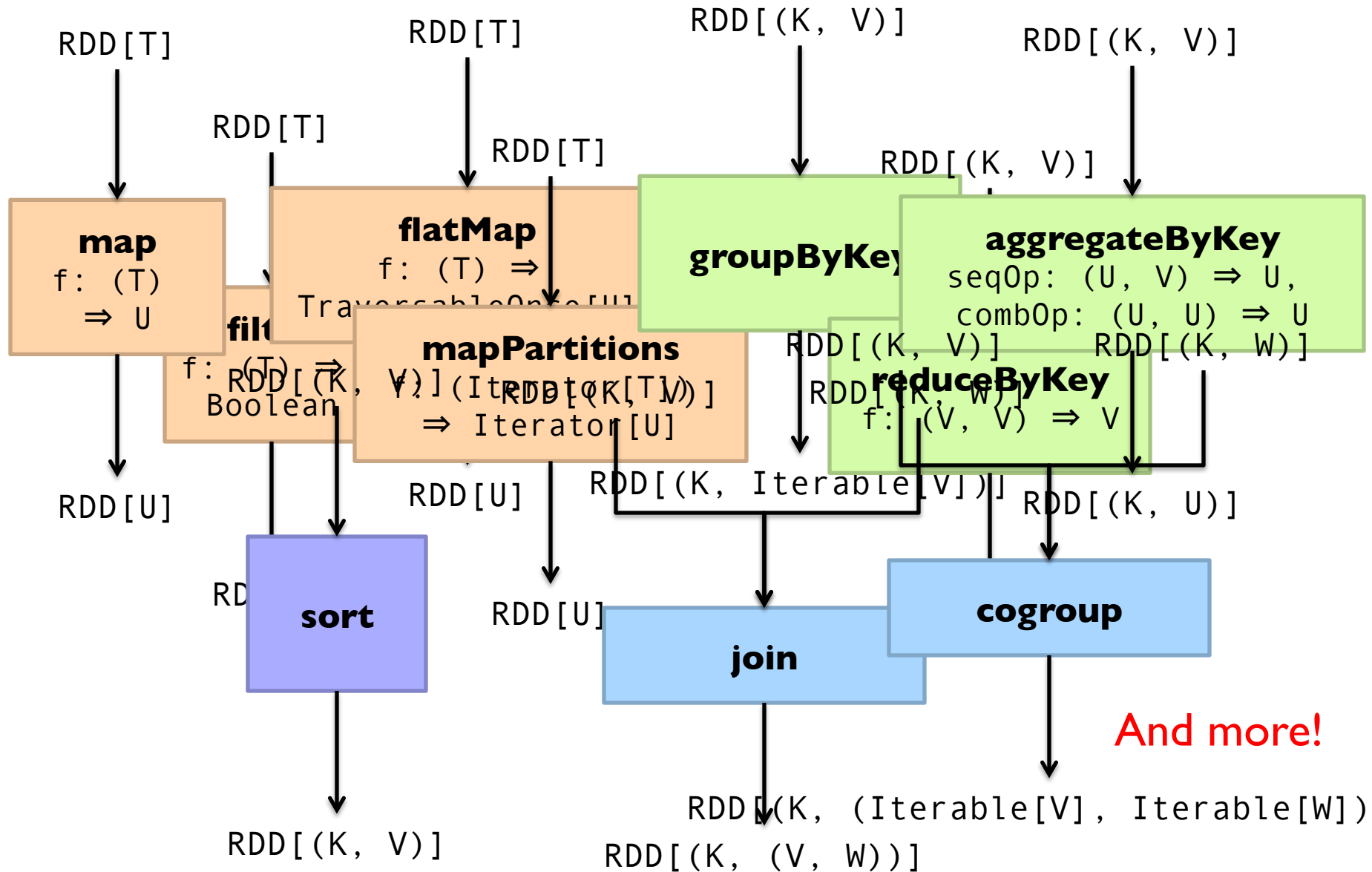
The datacenter *is* the computer!

What's the instruction set?

MapReduce



Spark



What's an RDD?

Resilient Distributed Dataset (RDD)

= immutable = partitioned

Wait, so how do you actually do anything?

Developers define *transformations* on RDDs

Framework keeps track of lineage

Spark Word Count

```
val textFile = sc.textFile(args.input())
```

```
textFile
```

```
  .flatMap(line => tokenize(line))
```

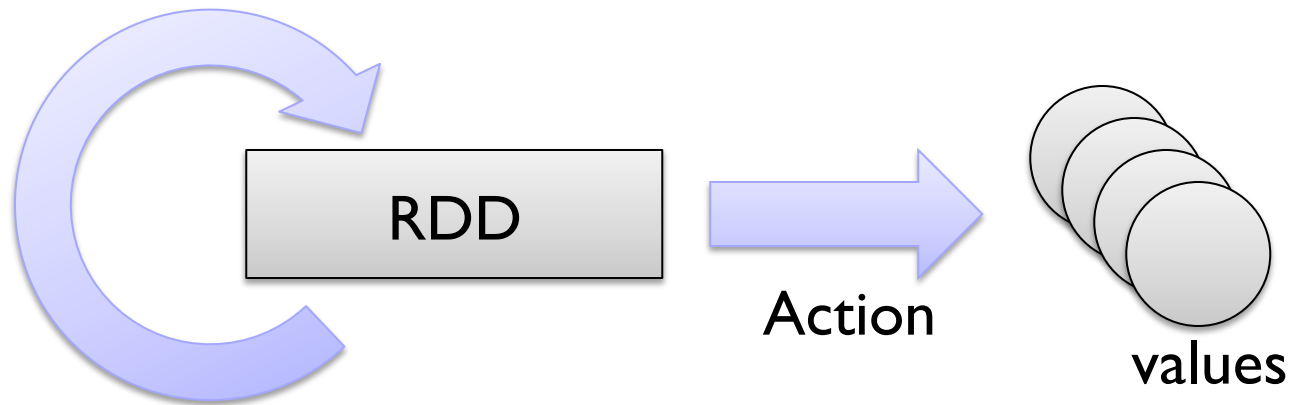
```
  .map(word => (word, 1))
```

```
  .reduceByKey(_ + _)
```

```
  .saveAsTextFile(args.output())
```

RDD Lifecycle

Transformation



Transformations are lazy:
Framework keeps track of lineage

Actions trigger actual execution

Spark Word Count

RDDs

```
val textFile = sc.textFile(args.input())  
val a = textFile.flatMap(line => line.split(" "))  
val b = a.map(word => (word, 1))  
val c = b.reduceByKey(_ + _)
```

Diagram illustrating the Spark Word Count program structure:

- RDDs** (Left side):
 - val textFile = sc.textFile(args.input())
 - val a = textFile.flatMap(line => line.split(" "))
 - val b = a.map(word => (word, 1))
 - val c = b.reduceByKey(_ + _)
- Transformations** (Right side):
 - val a = textFile.flatMap(line => line.split(" "))
 - val b = a.map(word => (word, 1))
 - val c = b.reduceByKey(_ + _)

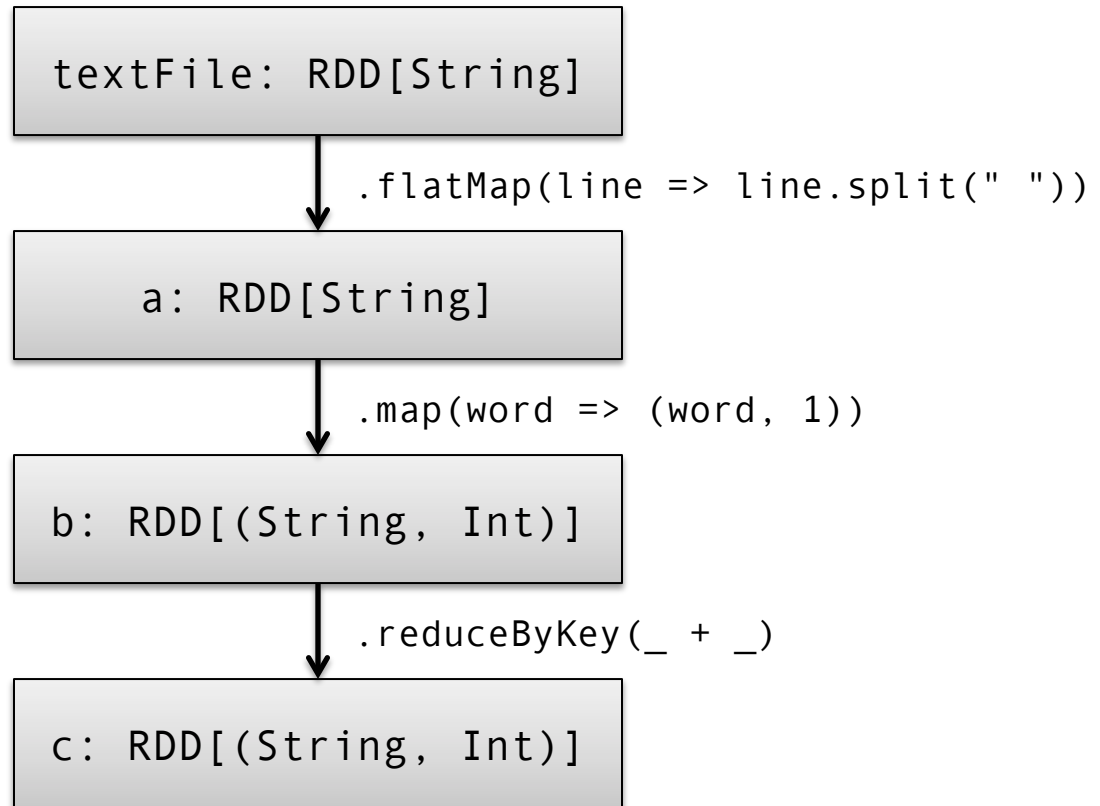
```
c.saveAsTextFile(args.output())
```

Action

Transformations

RDDs and Lineage

On HDFS



Action!

**Remember,
transformations are lazy!**

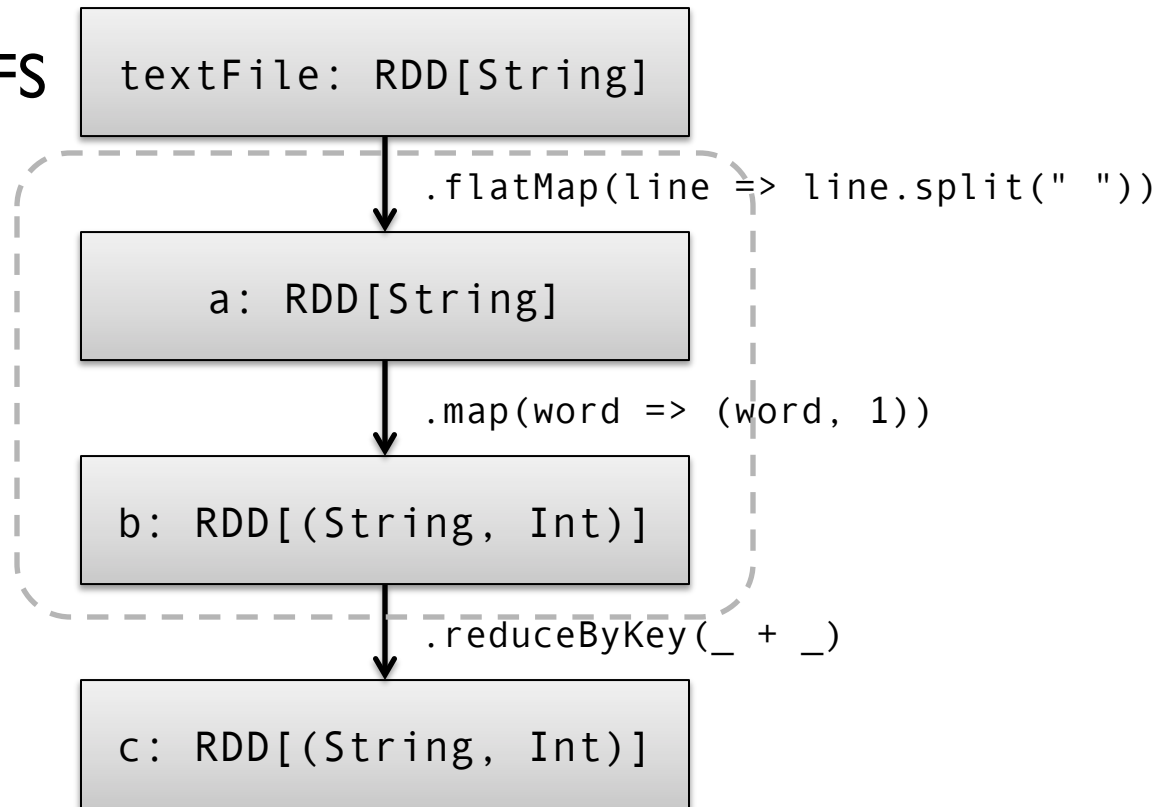
RDDs and Optimizations

Lazy evaluation creates optimization opportunities

On HDFS

RDDs don't need
to be materialized!

Want MM?

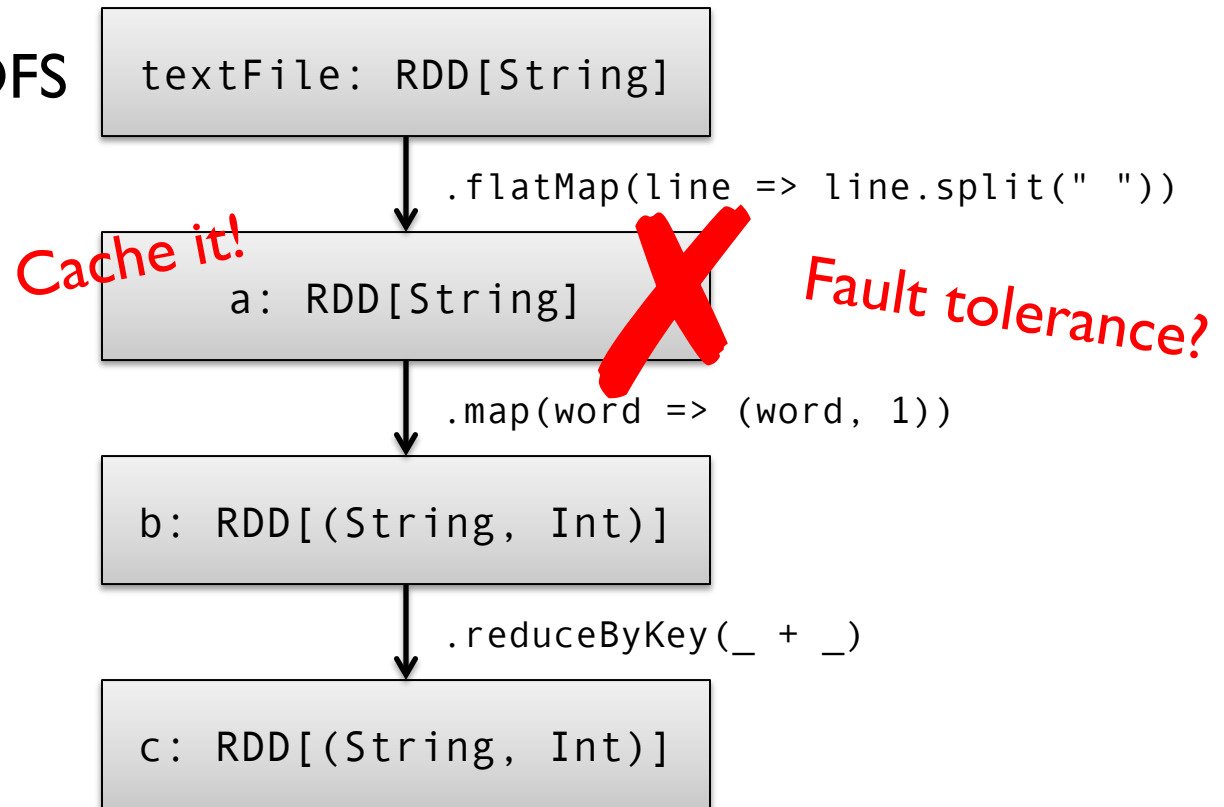


Action!

RDDs and Caching

RDDs can be materialized in memory!

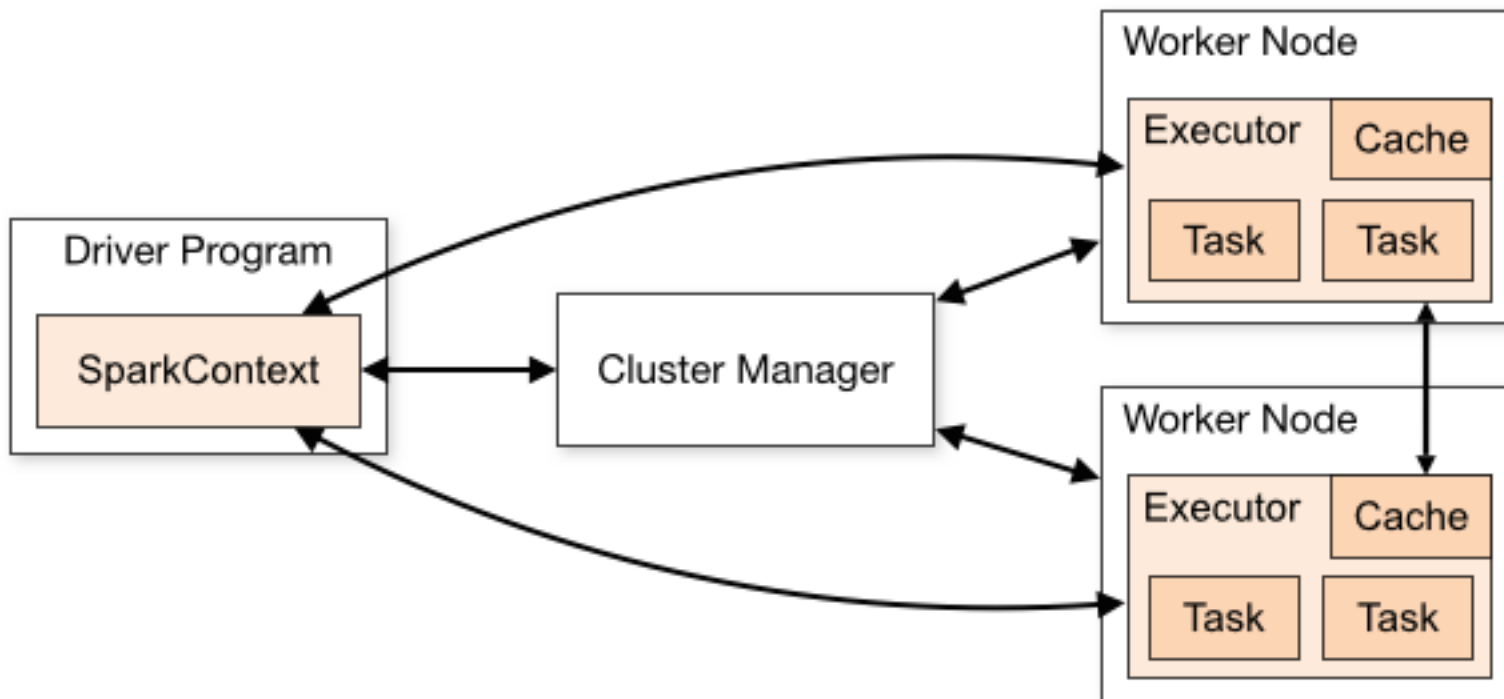
On HDFS



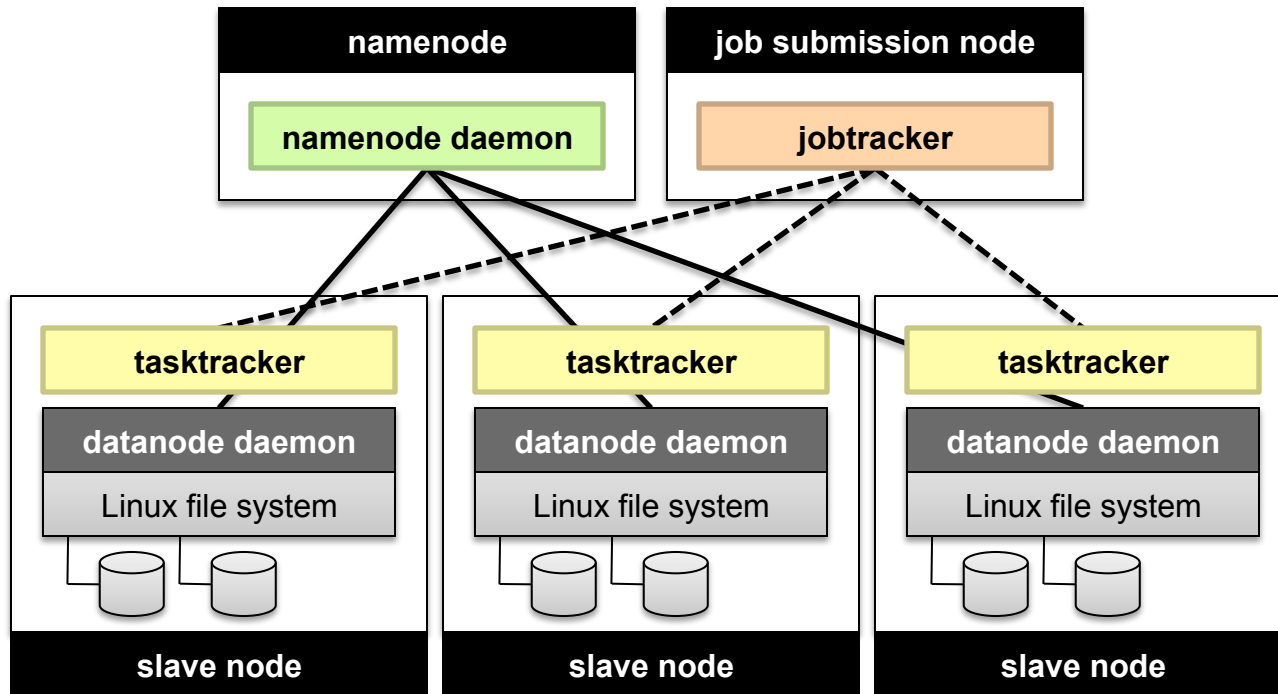
Action!

Spark works even if the RDDs are *partially* cached!

Spark Architecture



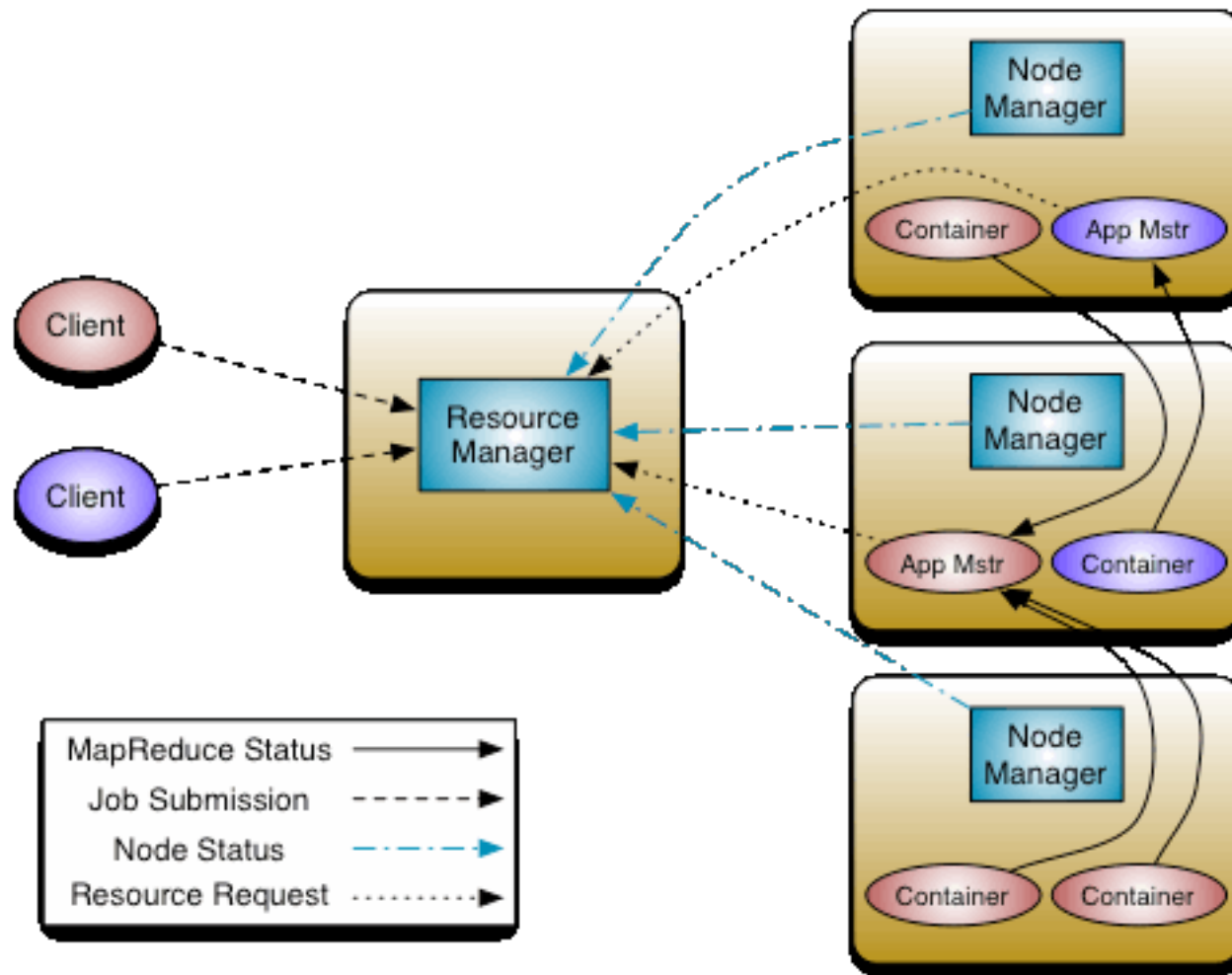
Compare



YARN

- Hadoop's (original) limitations:
 - Can only run MapReduce
 - What if we want to run other distributed frameworks?
- YARN = Yet-Another-Resource-Negotiator
 - Provides API to develop any generic distribution application
 - Handles scheduling and resource request
 - MapReduce (MR2) is one such application in YARN

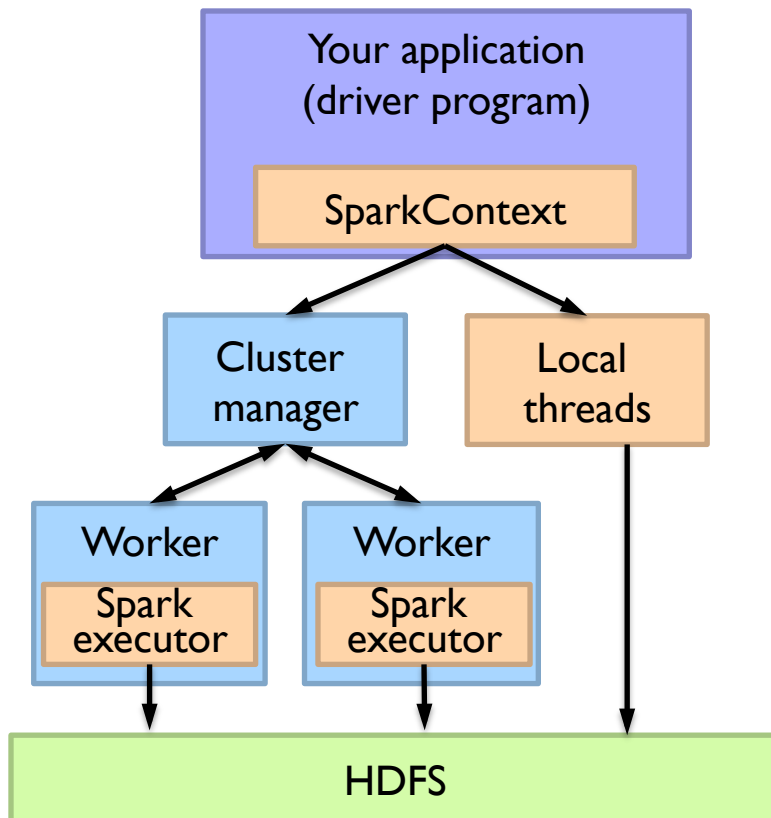
YARN



Spark Programs

Scala, Java, Python, R

`spark-shell` `spark-submit`

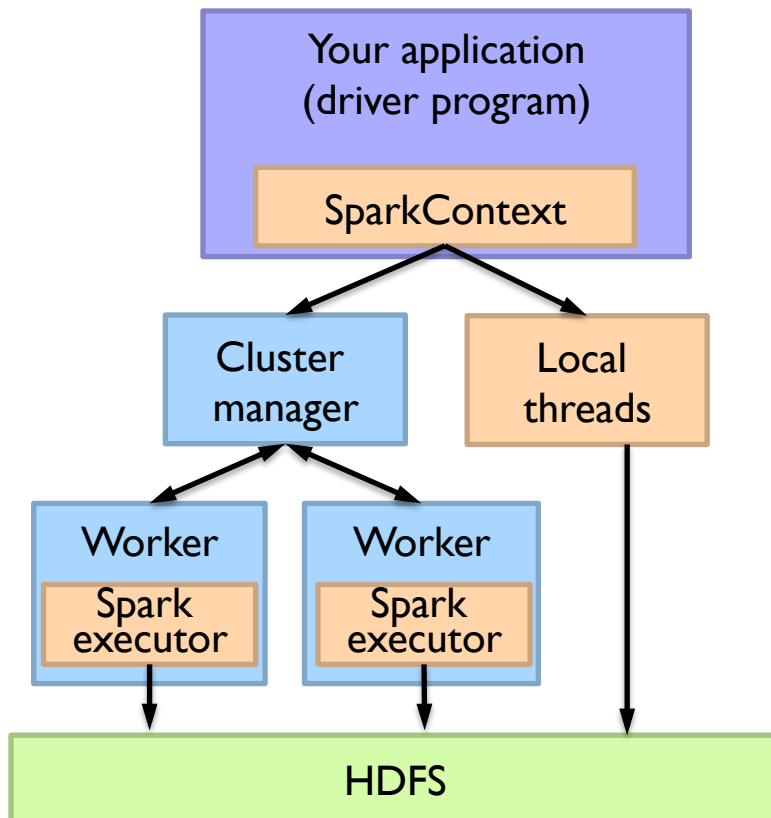


Spark context: tells the framework where to find the cluster

Use the Spark context to create RDDs

Spark Driver

spark-shell spark-submit



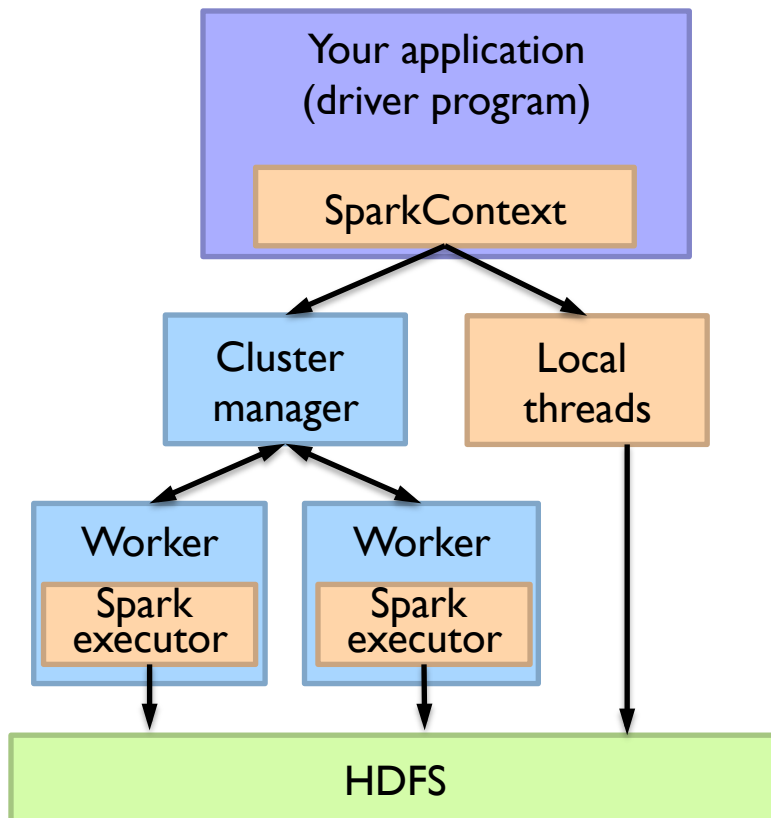
```
val textFile =  
  sc.textFile(args.input())
```

```
textFile  
  .flatMap(line => tokenize(line))  
  .map(word => (word, 1))  
  .reduceByKey(_ + _)  
  .saveAsTextFile(args.output())
```

What's happening
to the functions?

Spark Driver

spark-shell spark-submit



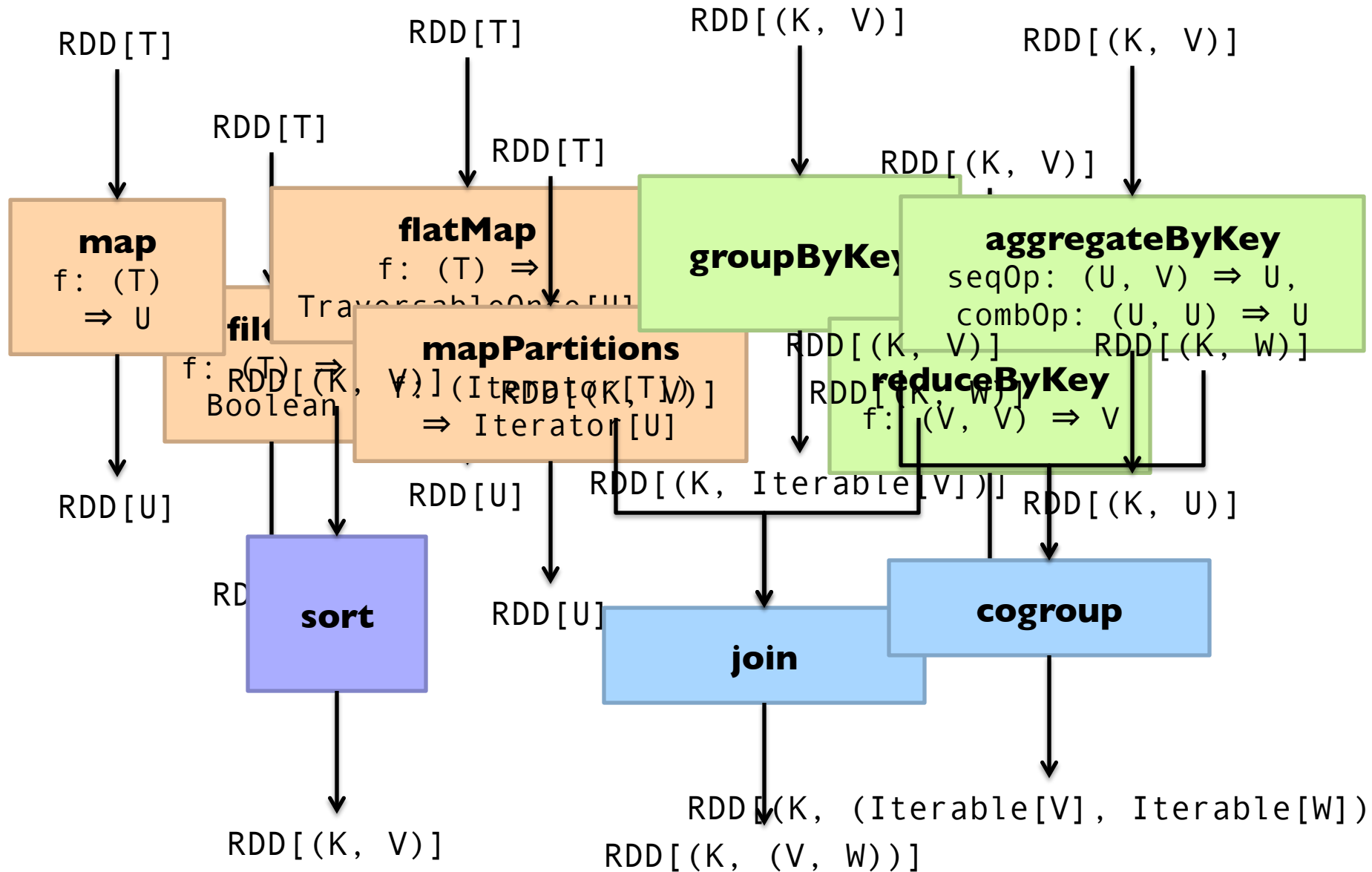
```
val textFile =  
  sc.textFile(args.input())
```

```
textFile  
  .flatMap(line => tokenize(line))  
  .map(word => (word, 1))  
  .reduceByKey(_ + _)  
  .saveAsTextFile(args.output())
```

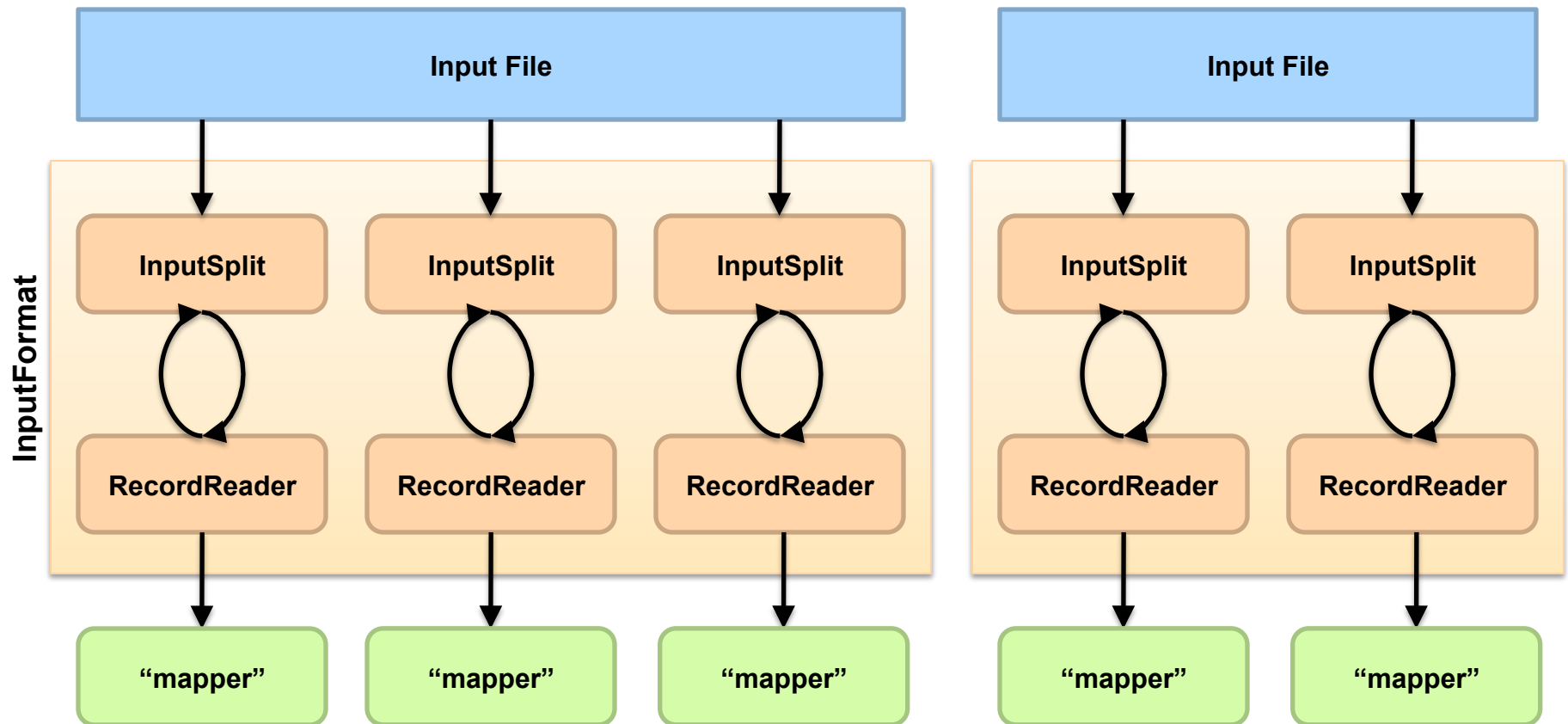
Note: you can run code “locally”,
integrate cluster-computed values!

Beware of the collect action!

Spark Transformations

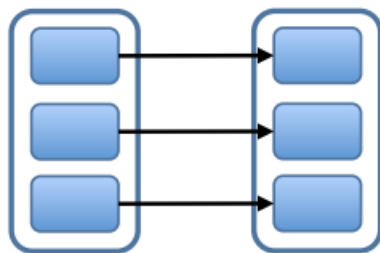


Starting Points

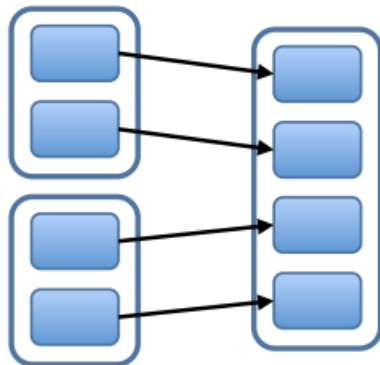


Physical Operators

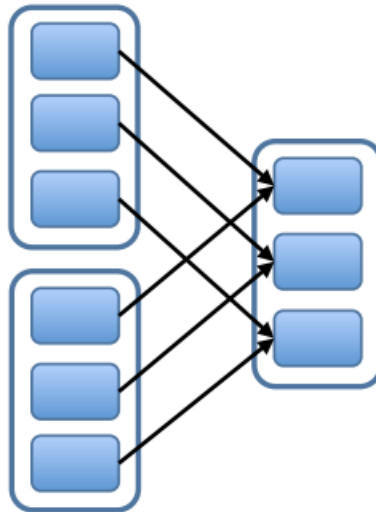
Narrow Dependencies:



map, filter

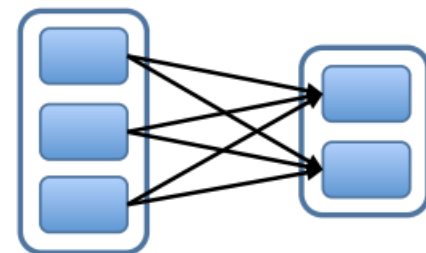


union

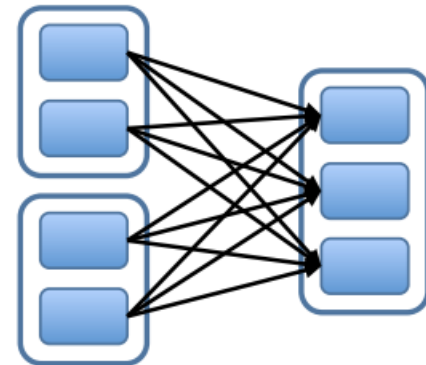


join with inputs
co-partitioned

Wide Dependencies:

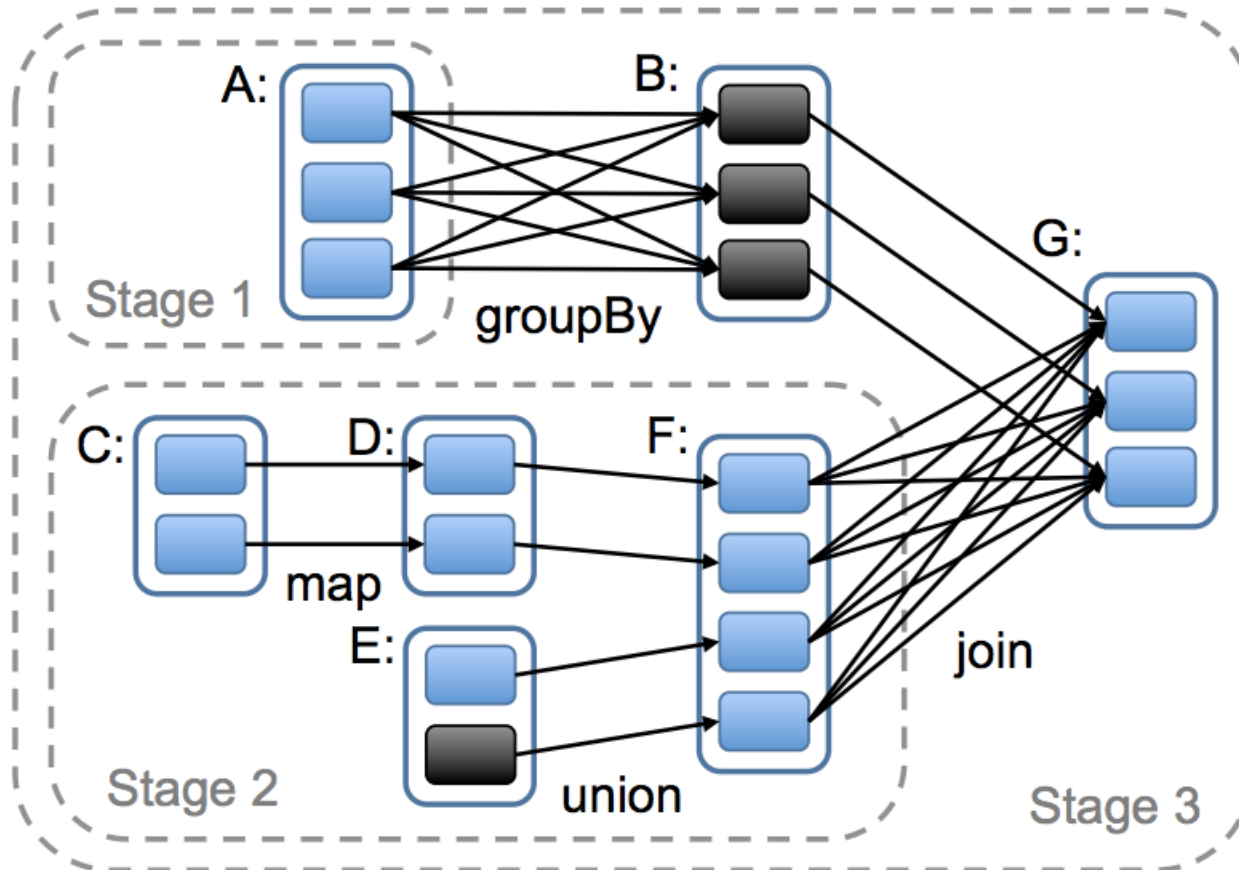


groupByKey

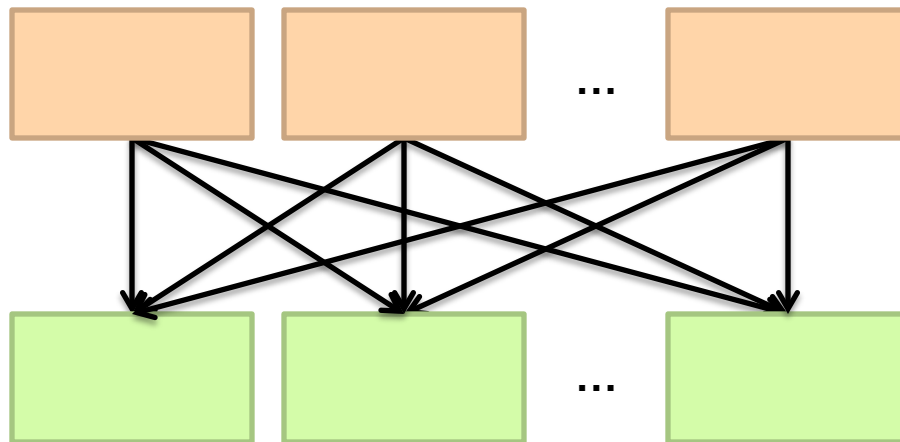


join with inputs not
co-partitioned

Execution Plan

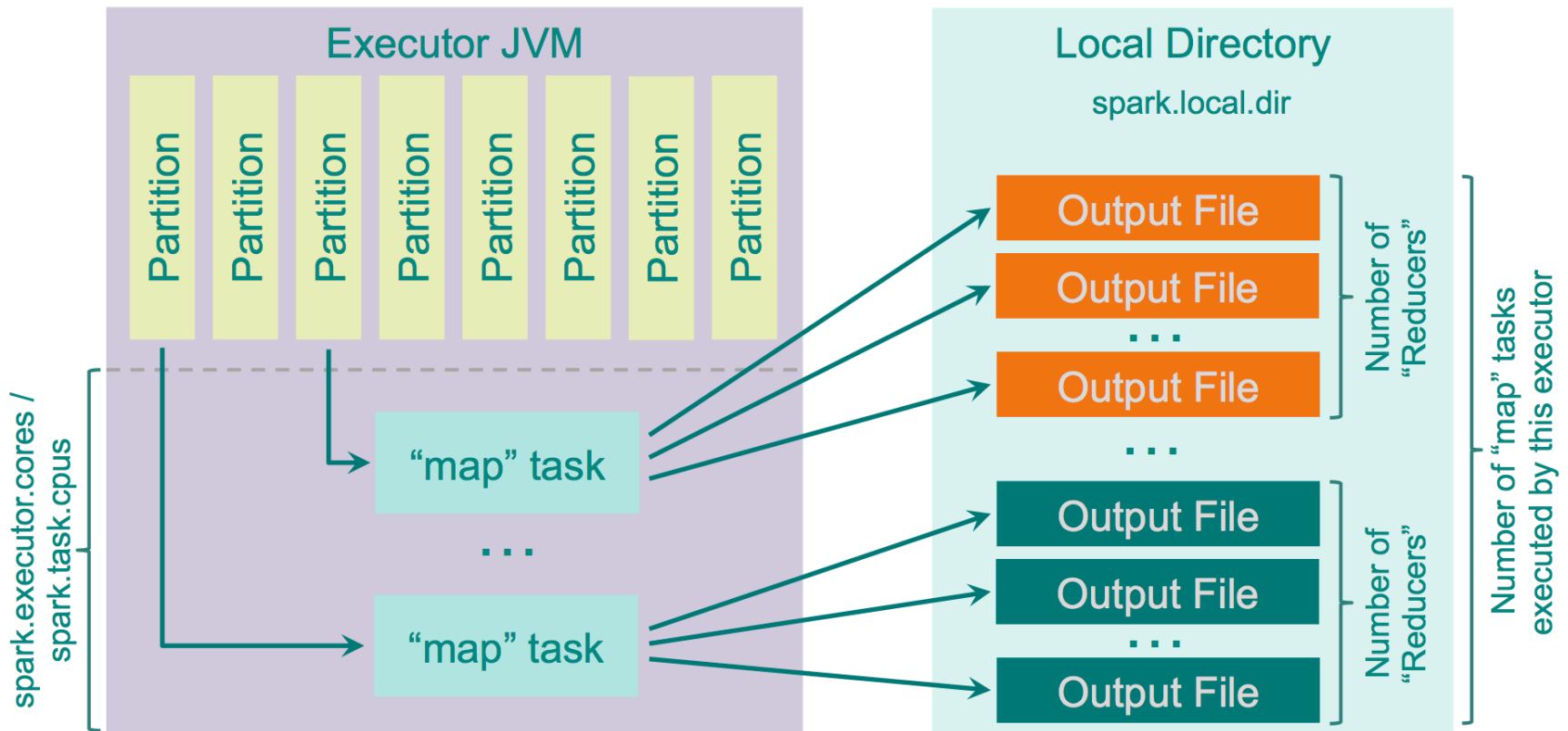


Can't avoid this!



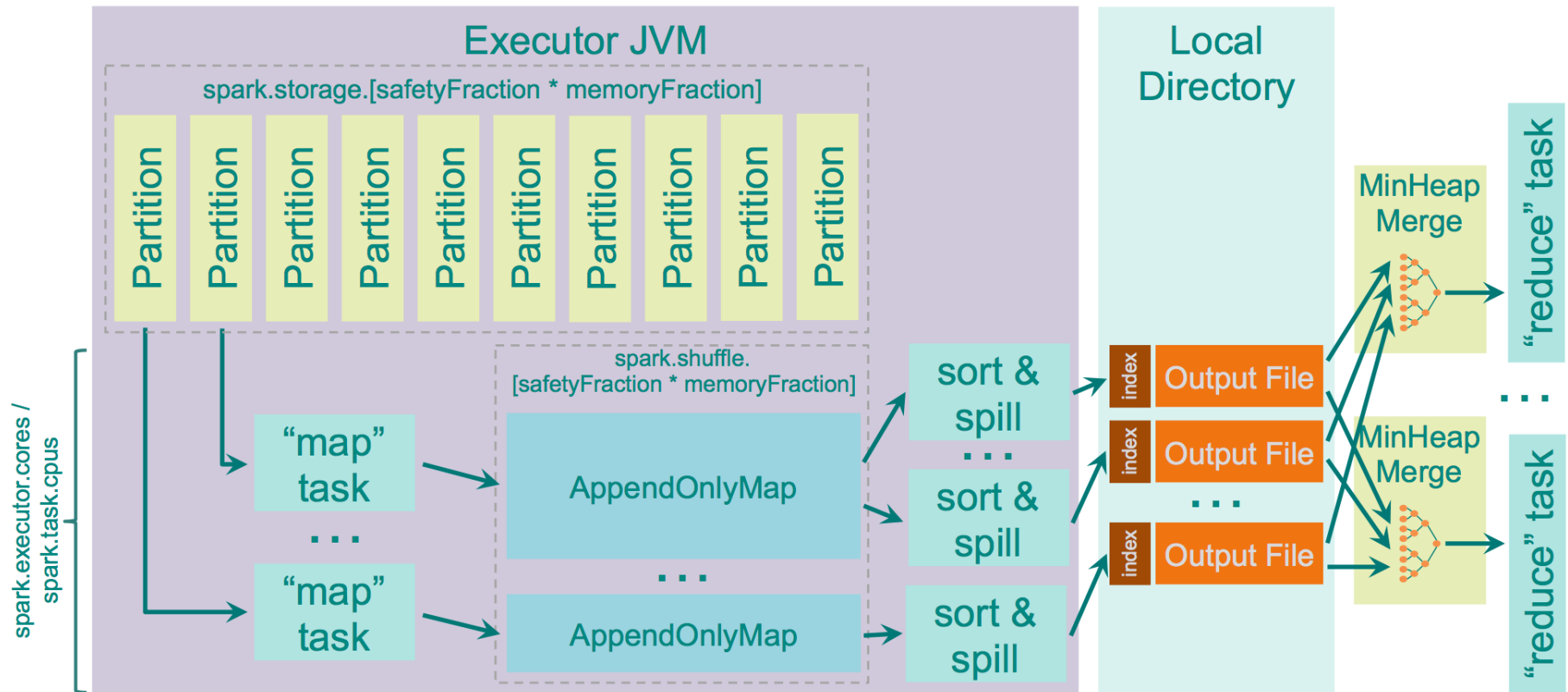
Spark Shuffle Implementations

Hash shuffle

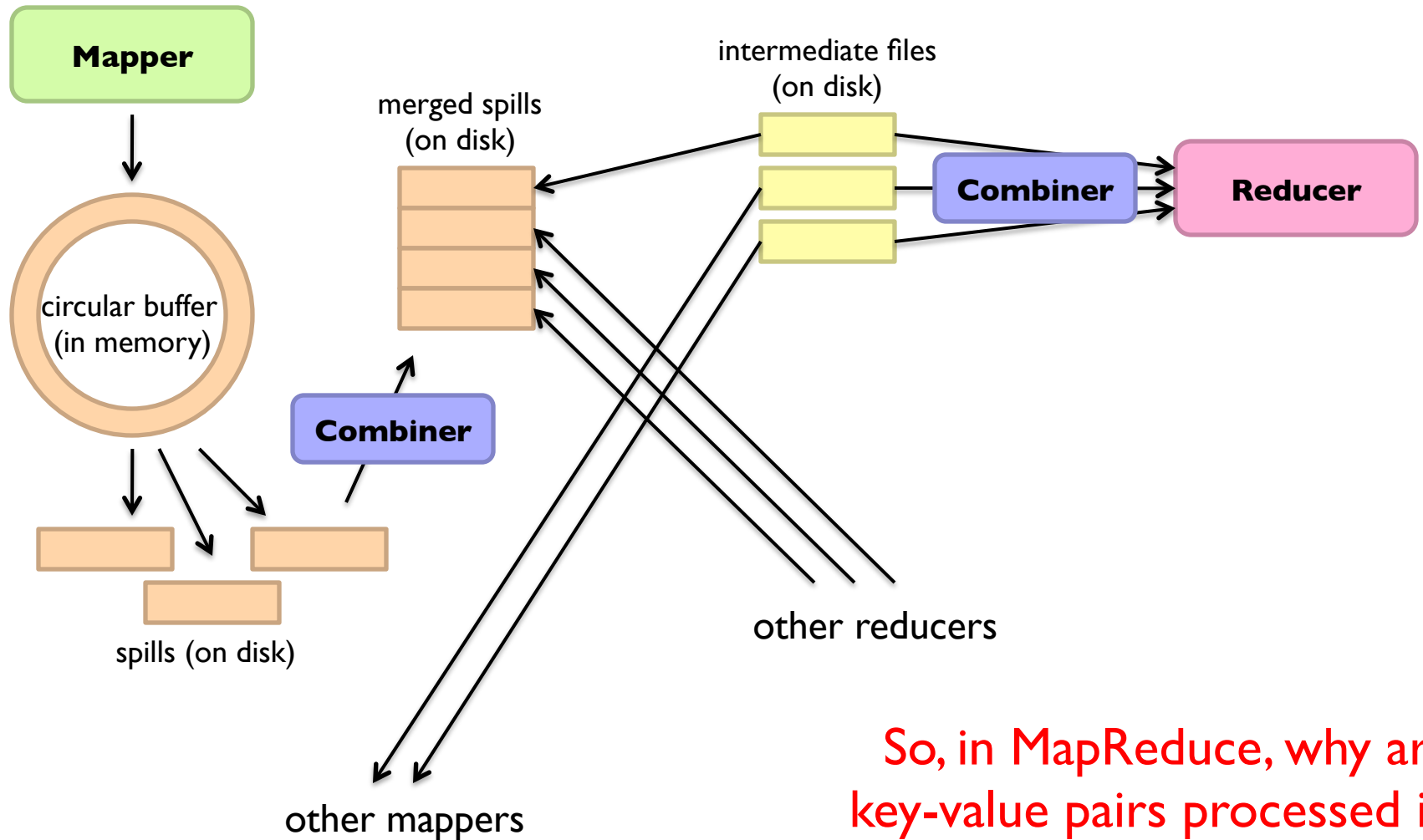


Spark Shuffle Implementations

Sort shuffle

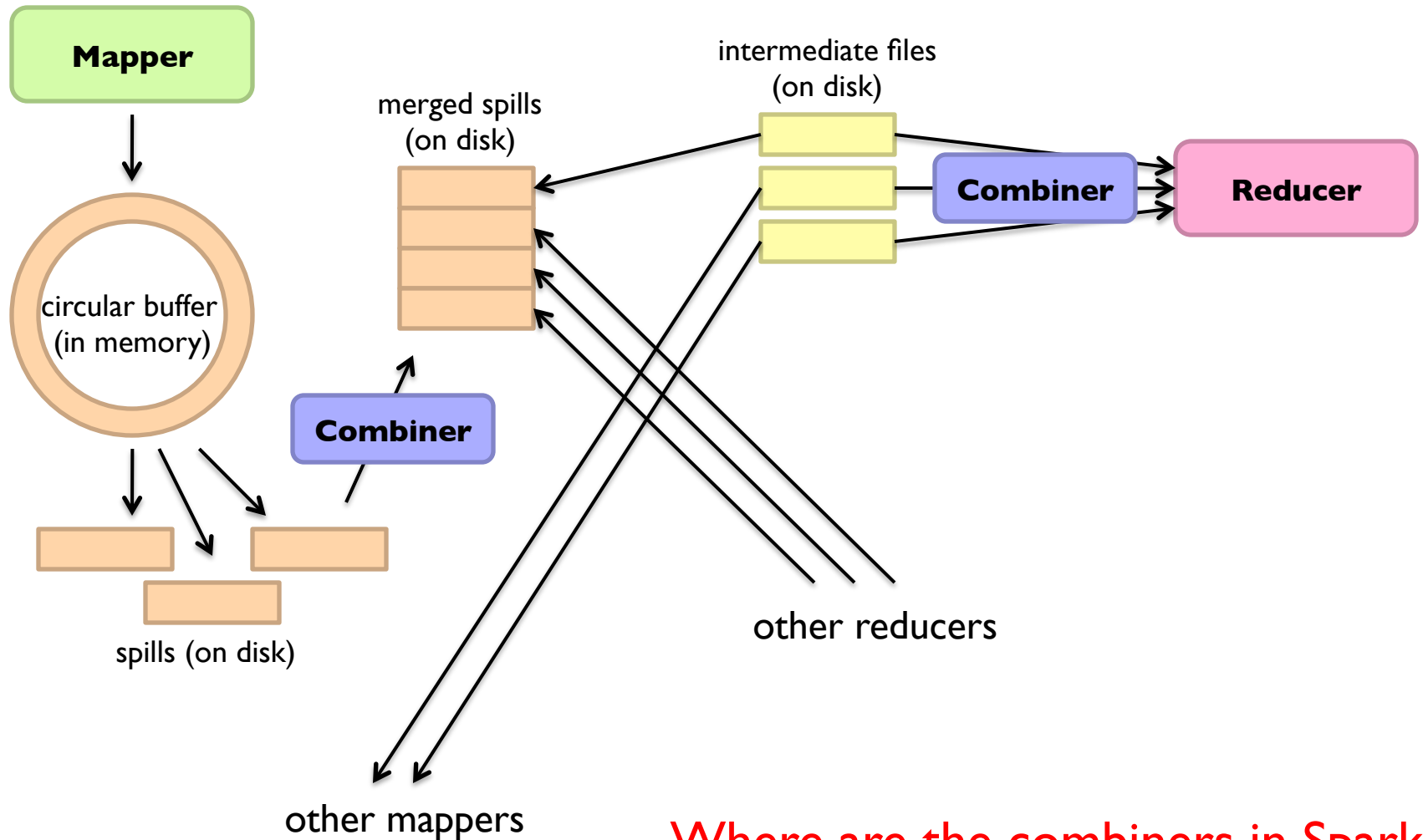


Remember this?



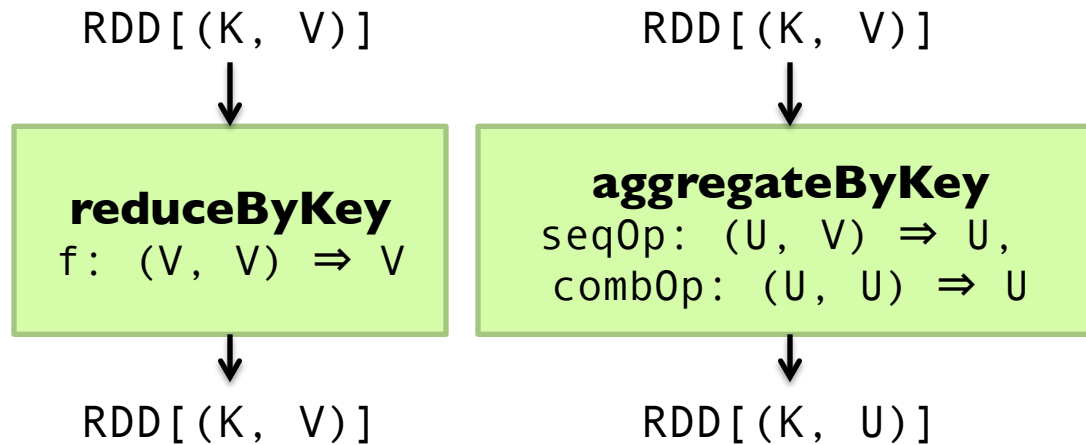
So, in MapReduce, why are key-value pairs processed in sorted order in the reducer?

Remember this?

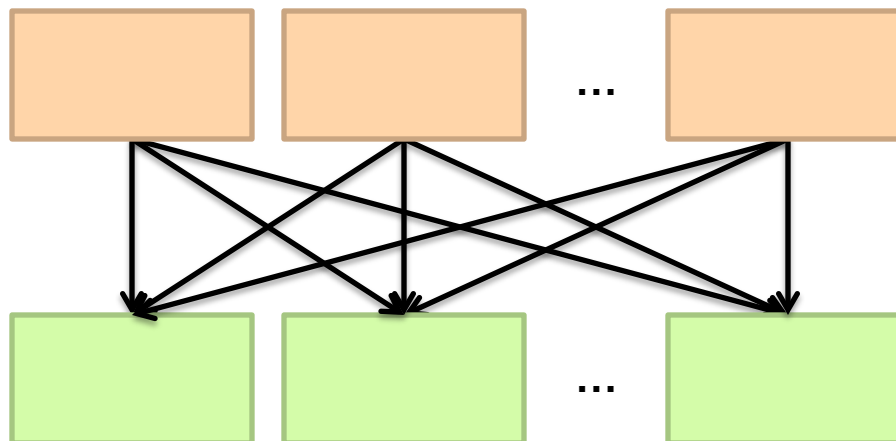


Where are the combiners in Spark?

Reduce-like Operations



How can we optimize?
What happened to combiners?



Spark #wins

Richer operators

RDD abstraction supports
optimizations (pipelining, caching, etc.)

Scala, Java, Python, R, bindings



Algorithm design, redux



Two superpowers:

Associativity
Commutativity
(sorting)

What follows... very basic category theory...

The Power of Associativity

You can put parenthesis where ever you want!

$$\left(v_1 \oplus v_2 \oplus v_3 \right) \oplus \left(v_4 \oplus v_5 \oplus v_6 \oplus v_7 \right) \oplus \left(v_8 \oplus v_9 \right)$$

$$\left(v_1 \oplus v_2 \right) \oplus \left(v_3 \oplus v_4 \oplus v_5 \right) \oplus \left(v_6 \oplus v_7 \oplus v_8 \oplus v_9 \right)$$

$$\left(v_1 \oplus v_2 \oplus \left(v_3 \oplus v_4 \oplus v_5 \right) \right) \oplus \left(v_6 \oplus v_7 \oplus v_8 \oplus v_9 \right)$$

The Power of Commutativity

You can swap order of operands however you want!

$$(v_1 \oplus v_2 \oplus v_3) \oplus (v_4 \oplus v_5 \oplus v_6 \oplus v_7) \oplus (v_8 \oplus v_9)$$

$$(v_4 \oplus v_5 \oplus v_6 \oplus v_7) \oplus (v_1 \oplus v_2 \oplus v_3) \oplus (v_8 \oplus v_9)$$

$$(v_8 \oplus v_9) \oplus (v_4 \oplus v_5 \oplus v_6 \oplus v_7) \oplus (v_1 \oplus v_2 \oplus v_3)$$

Implications for distributed processing?

You don't know when the tasks begin

You don't know when the tasks end

You don't know when the tasks interrupt each other

You don't know when intermediate data arrive

...

It's okay!

Fancy Labels for Simple Concepts...

Semigroup = $(M, \oplus)$

$$\oplus : M \times M \rightarrow M, \text{ s.t., } \forall m_1, m_2, m_3 \in M$$

$$(m_1 \oplus m_2) \oplus m_3 = m_1 \oplus (m_2 \oplus m_3)$$

Monoid = Semigroup + identity

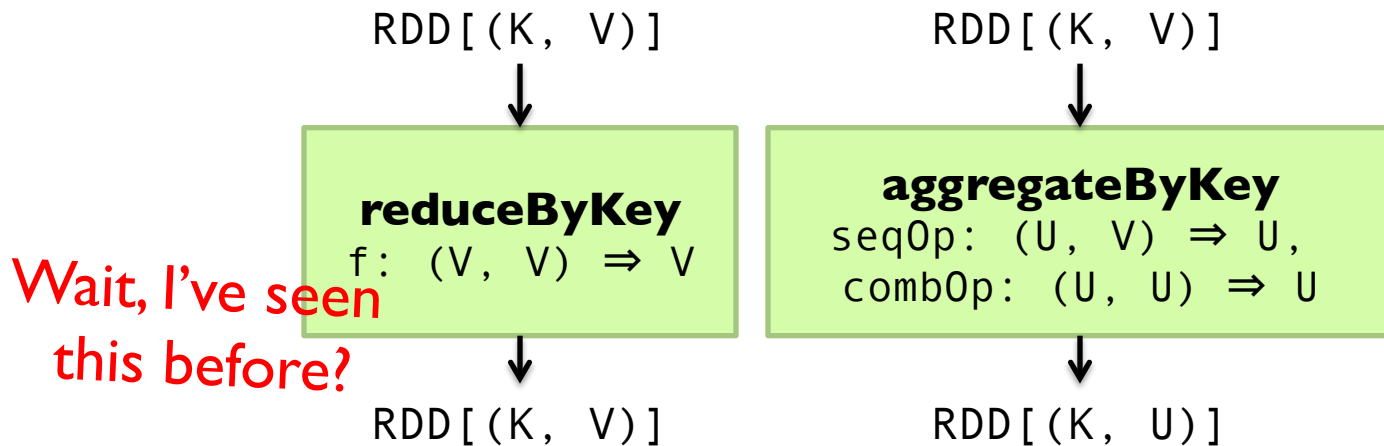
$$\varepsilon \text{ s.t., } \varepsilon \oplus m = m \oplus \varepsilon = m, \forall m \in M$$

Commutative Monoid = Monoid + commutativity

$$\forall m_1, m_2 \in M, m_1 \oplus m_2 = m_2 \oplus m_1$$

A few examples?

Back to these...



Computing the mean v l (again)

```
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  $r_{avg}$ )
```

Computing the mean v2 (again)

```
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  $r_{avg}$ )
```

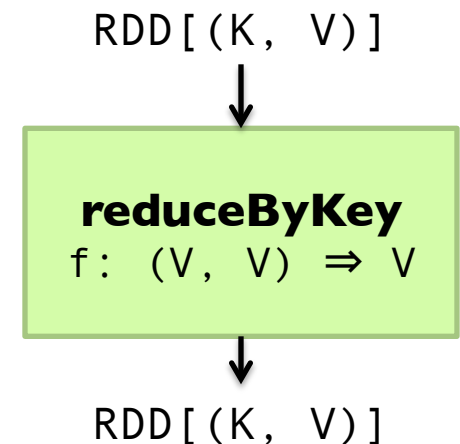
Computing the mean v3 (again)

```
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 [(s1, c1), (s2, c2) ...])
3:     sum ← 0
4:     cnt ← 0
5:     for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] 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 [(s1, c1), (s2, c2) ...])
3:     sum ← 0
4:     cnt ← 0
5:     for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:       sum ← sum + s
7:       cnt ← cnt + c
8:     ravg ← sum / cnt
9:     EMIT(string t, pair (ravg, cnt))
```

Wait, I've seen
this before?

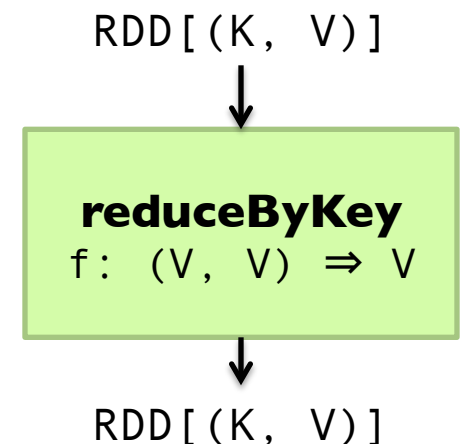


Co-occurrence Matrix: Stripes

```
1: class MAPPER
2:   method MAP(docid  $a$ , doc  $d$ )
3:     for all term  $w \in \text{doc } d$  do
4:        $H \leftarrow \text{new ASSOCIATIVEARRAY}$ 
5:       for all term  $u \in \text{NEIGHBORS}(w)$  do
6:          $H\{u\} \leftarrow H\{u\} + 1$  ▷ Tally words co-occurring with  $w$ 
7:       EMIT(Term  $w$ , Stripe  $H$ )
```

```
1: class REDUCER
2:   method REDUCE(term  $w$ , stripes  $[H_1, H_2, H_3, \dots]$ )
3:      $H_f \leftarrow \text{new ASSOCIATIVEARRAY}$ 
4:     for all stripe  $H \in \text{stripes } [H_1, H_2, H_3, \dots]$  do
5:       SUM( $H_f, H$ )
6:     EMIT(term  $w$ , stripe  $H_f$ )
```

Wait, I've seen this before?



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

What about this?

Commutative monoids!

f(B|A): “Pairs”

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

(a, *) → 32

Reducer holds this value in memory

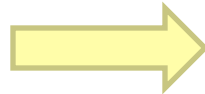
(a, b₁) → 3

(a, b₂) → 12

(a, b₃) → 7

(a, b₄) → 1

...



(a, b₁) → 3 / 32

(a, b₂) → 12 / 32

(a, b₃) → 7 / 32

(a, b₄) → 1 / 32

...

○ For this to work:

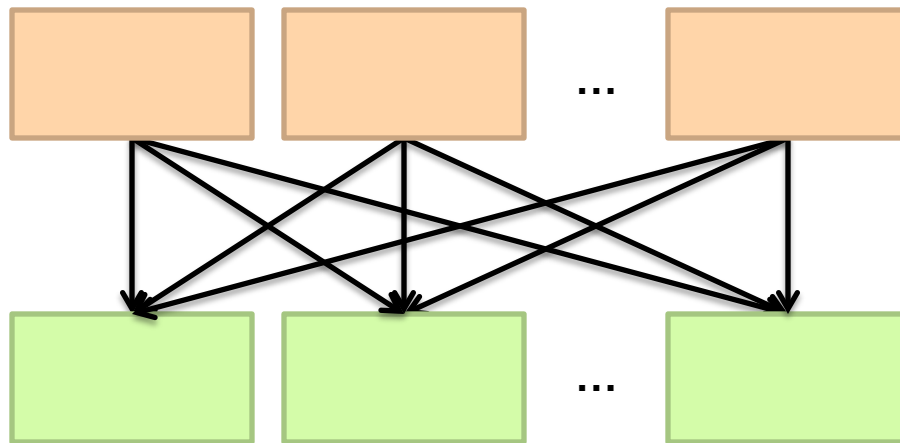
- Must emit extra (a, *) for every b_n in mapper
- Must make sure all a's get sent to same reducer (use partitioner)
- Must make sure (a, *) comes first (define sort order)
- Must hold state in reducer across different key-value pairs



Two superpowers:

Associativity
Commutativity
(sorting)

Because you can't avoid this...



And sort-based shuffling is pretty efficient!

An aerial photograph of a large industrial facility, likely a datacenter, situated in a rural area. The facility consists of several large, white, rectangular buildings with flat roofs, arranged in a grid-like pattern. In the foreground, there is a large parking lot filled with many white semi-trailers. To the right, a large building with a complex roof structure, possibly a cooling or power plant, is visible. The surrounding landscape is a mix of green fields and brown, tilled soil. In the background, a line of trees and distant hills are visible under a sky with a bright, low sun, creating a warm, orange glow.

The datacenter *is* the computer!

What's the instruction set?

Algorithm design in a nutshell...



Exploit associativity and commutativity
via commutative monoids (if you can)

Exploit framework-based sorting to
sequence computations (if you can't)



Questions?