UNIVERSITY OF WATERLOO

Big Data Infrastructure

CS 489/698 Big Data Infrastructure (Winter 2016)

Week 3: From MapReduce to Spark (2/2)

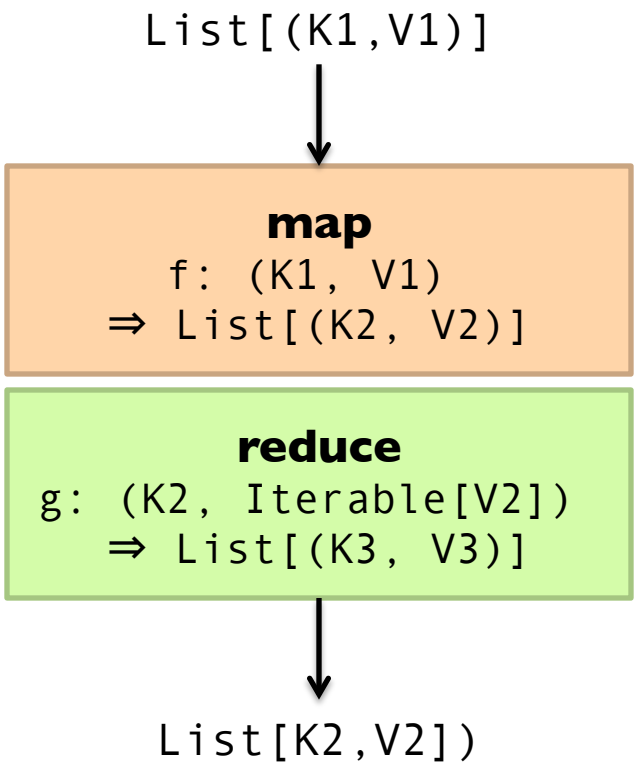
January 21, 2016

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

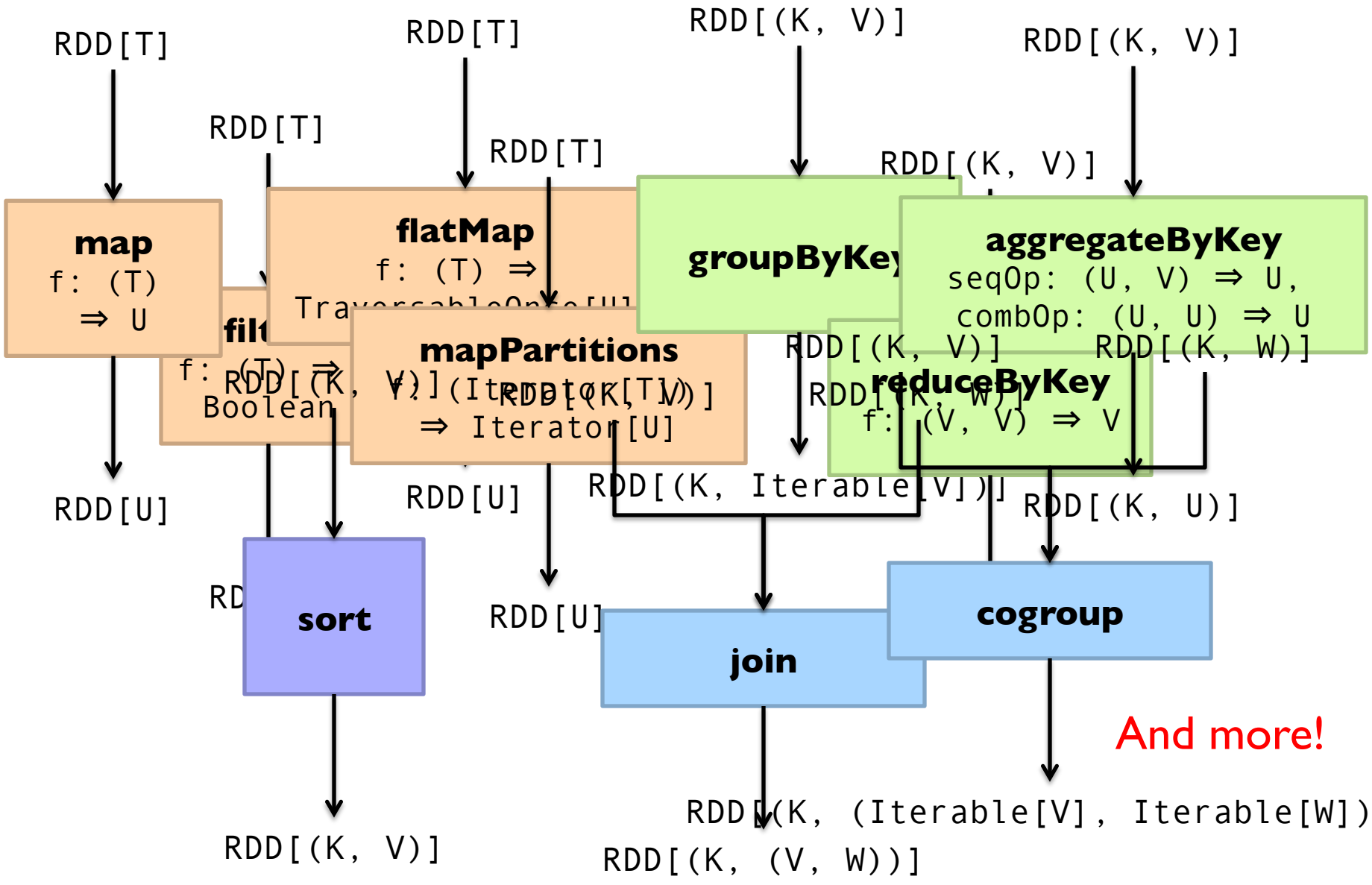
These slides are available at http://lintool.github.io/bigdata-2016w/


The datacenter *is* the computer!
What's the instruction set?
Source: Google

MapReduce


List[(K1,V1)]
map
f: (K1, V1)
⇒ List[(K2, V2)]
reduce
g: (K2, Iterable[V2])
⇒ List[(K3, V3)]
List[K2,V2])

Spark


RDD[T]
RDD[T]
RDD[(K, V)]
RDD[(K, V)]
RDD[T]
RDD[T]
RDD[(K, V)]
map
f: (T) ⇒ U
flatMap
f: (T) ⇒
groupByKey
aggregateByKey
seqOp: (U, V) ⇒ U,
combOp: (U, U) ⇒ U
filter
f: (T) ⇒ Boolean
mapPartitions
f: (Iterator[T]) ⇒ Iterator[U]
RDD[(K, V)]
RDD[(K, W)]
reduceByKey
f: (V, V) ⇒ V
RDD[U]
RDD[U]
RDD[(K, Iterable[V])]
RDD[(K, U)]
sort
RDD[U]
join
cogroup
And more!
RDD[(K, V)]
RDD[(K, (Iterable[V], Iterable[W]))]
RDD[(K, (V, W))]

What's an RDD?

Resilient Distributed Dataset (RDD)

= immutable = partitioned

Wait, so how do you actually do anything?

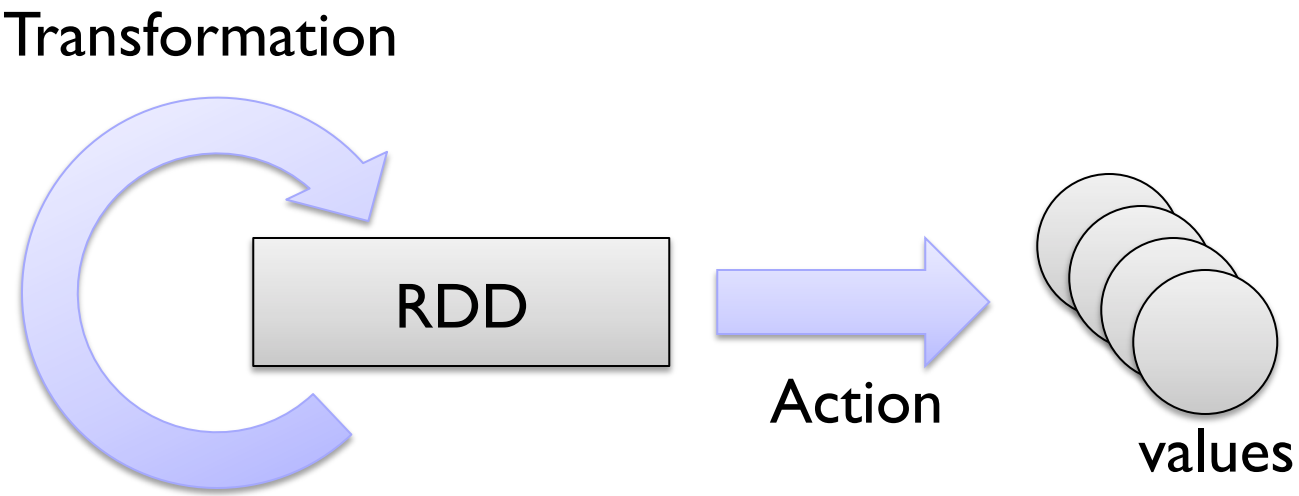
- Developers define *transformations* on RDDs
- Framework keeps track of lineage

Spark Word Count

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

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

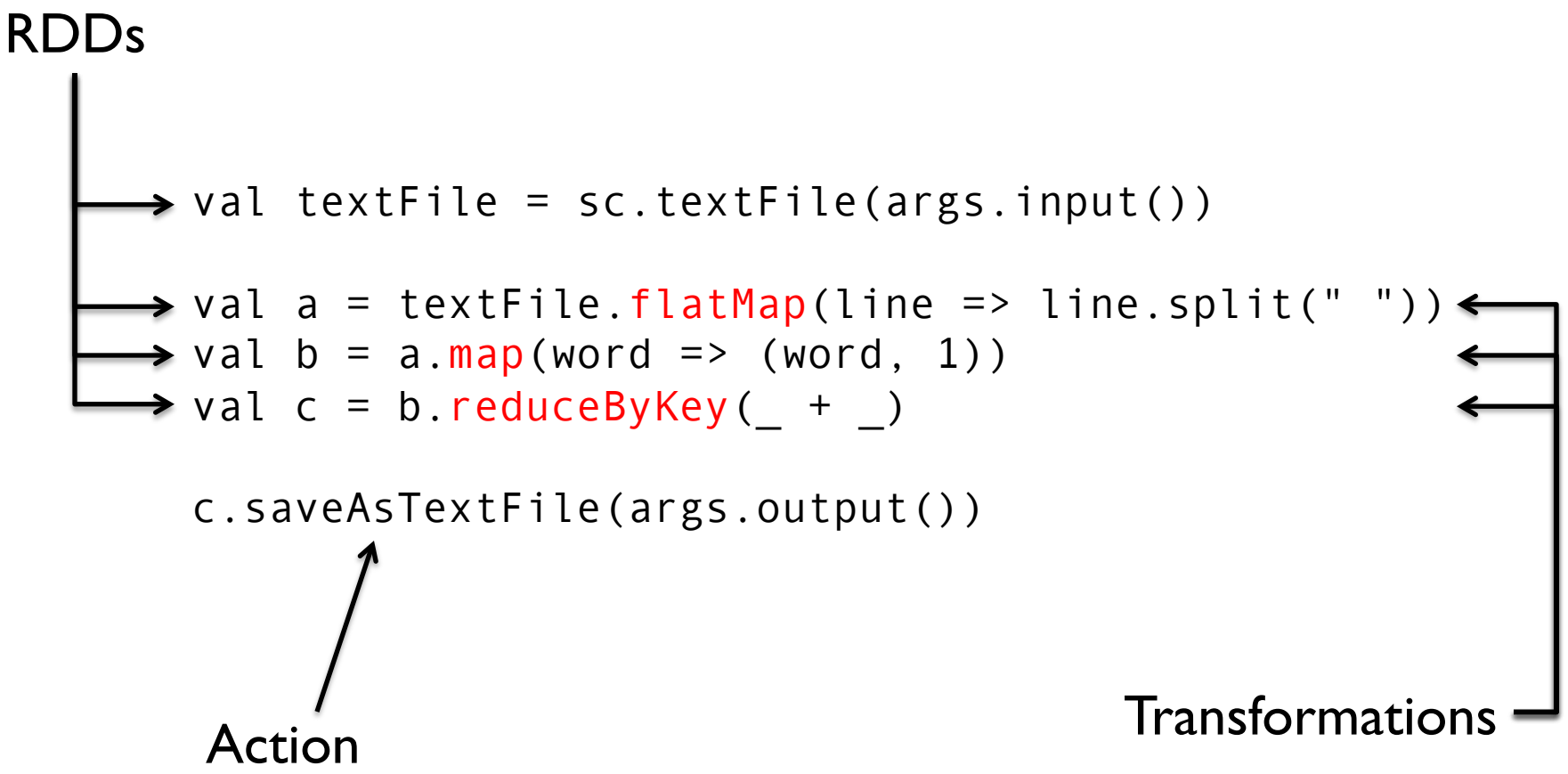
RDD Lifecycle



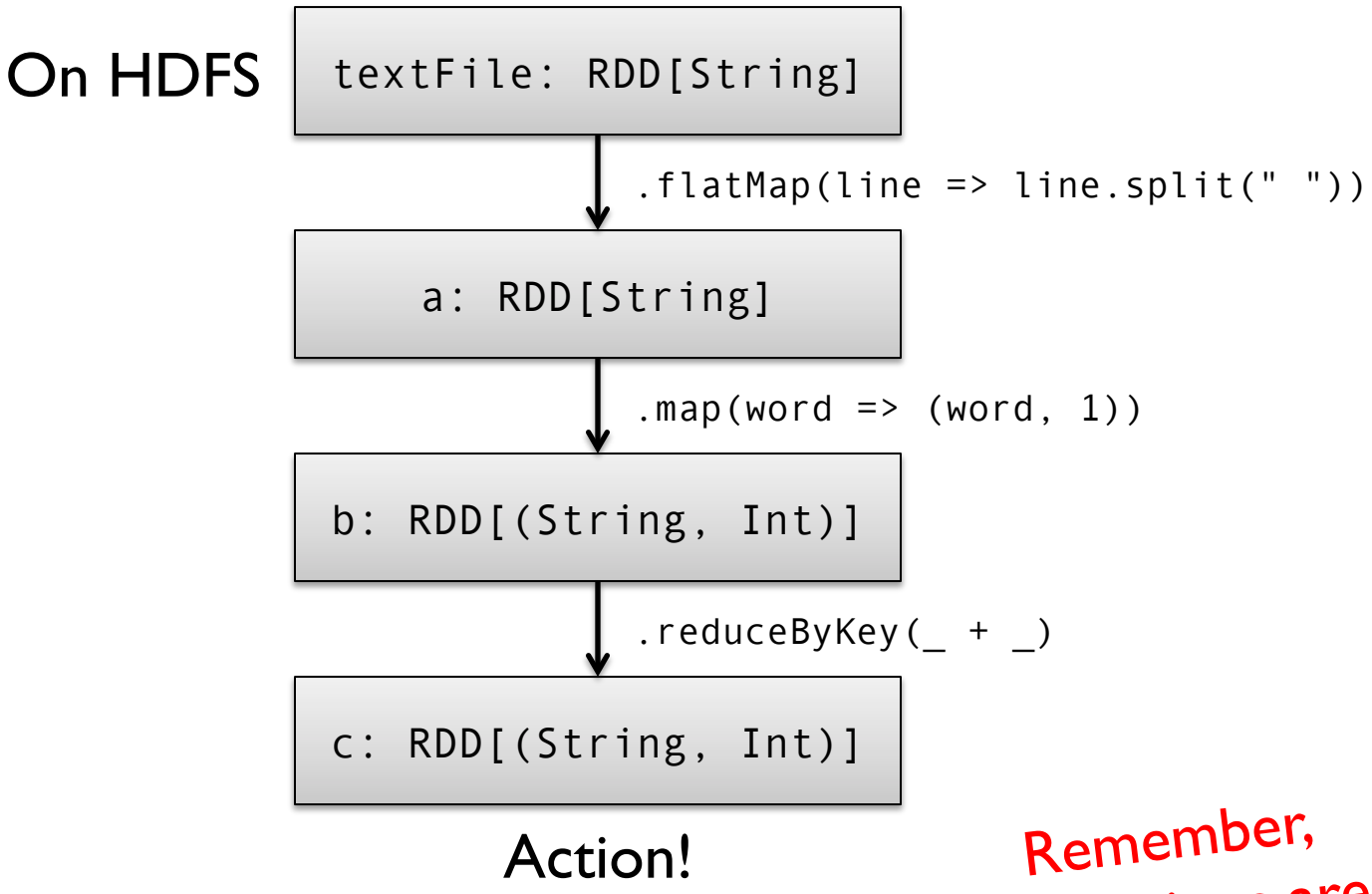
Transformations are lazy:
Framework keeps track of lineage

Actions trigger actual execution

Spark Word Count

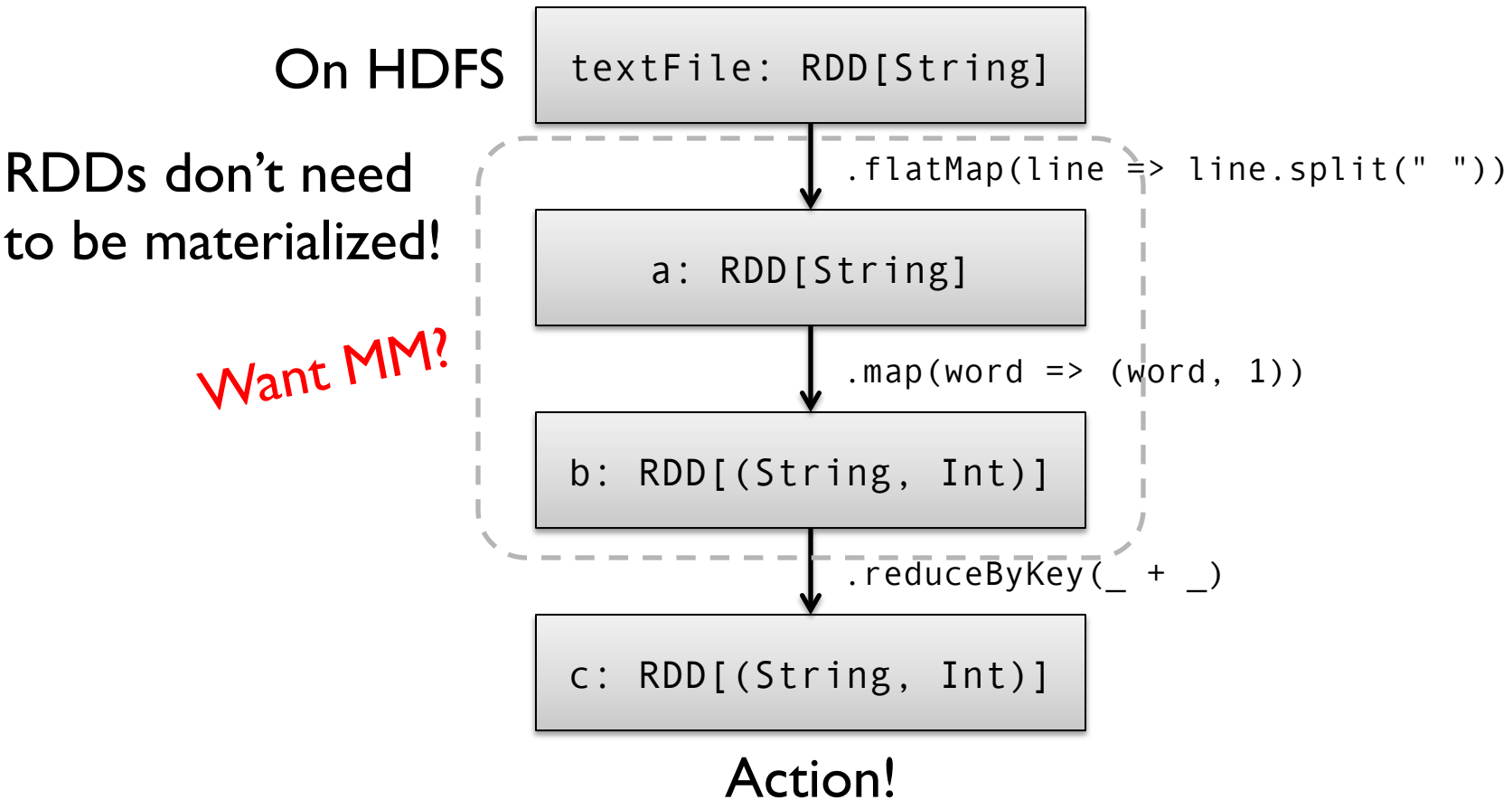


RDDs and Lineage



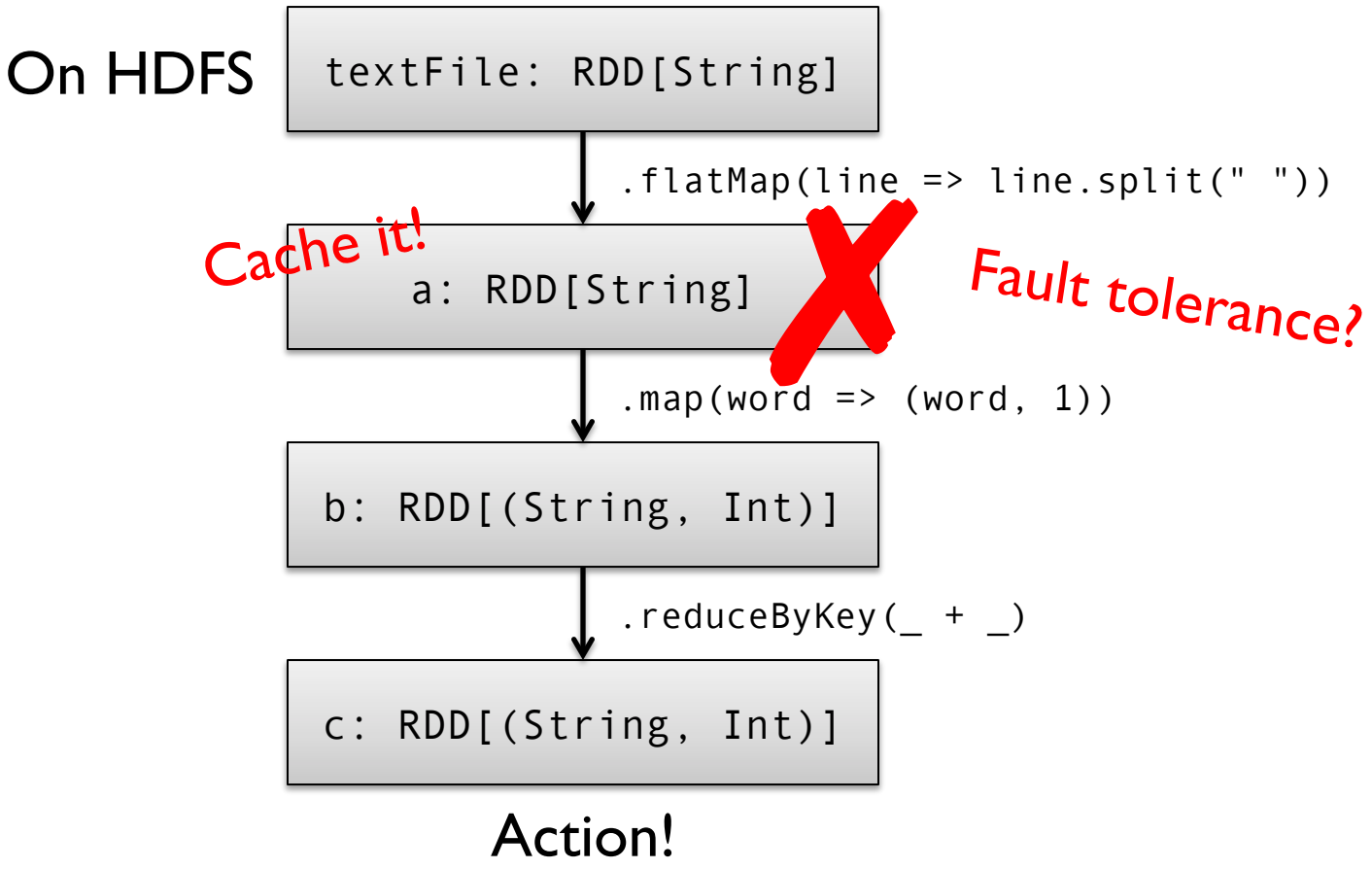
RDDs and Optimizations

Lazy evaluation creates optimization opportunities



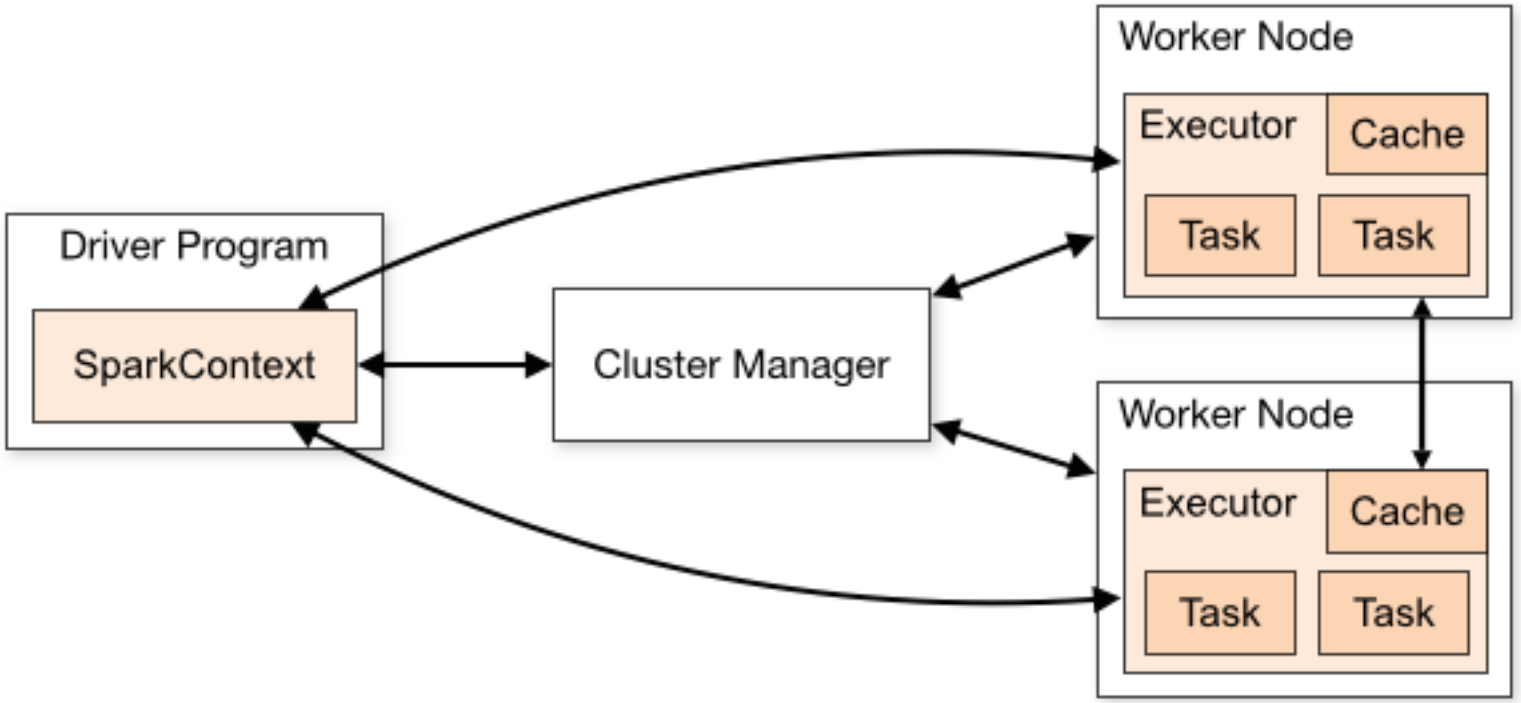
RDDs and Caching

RDDs can be materialized in memory!

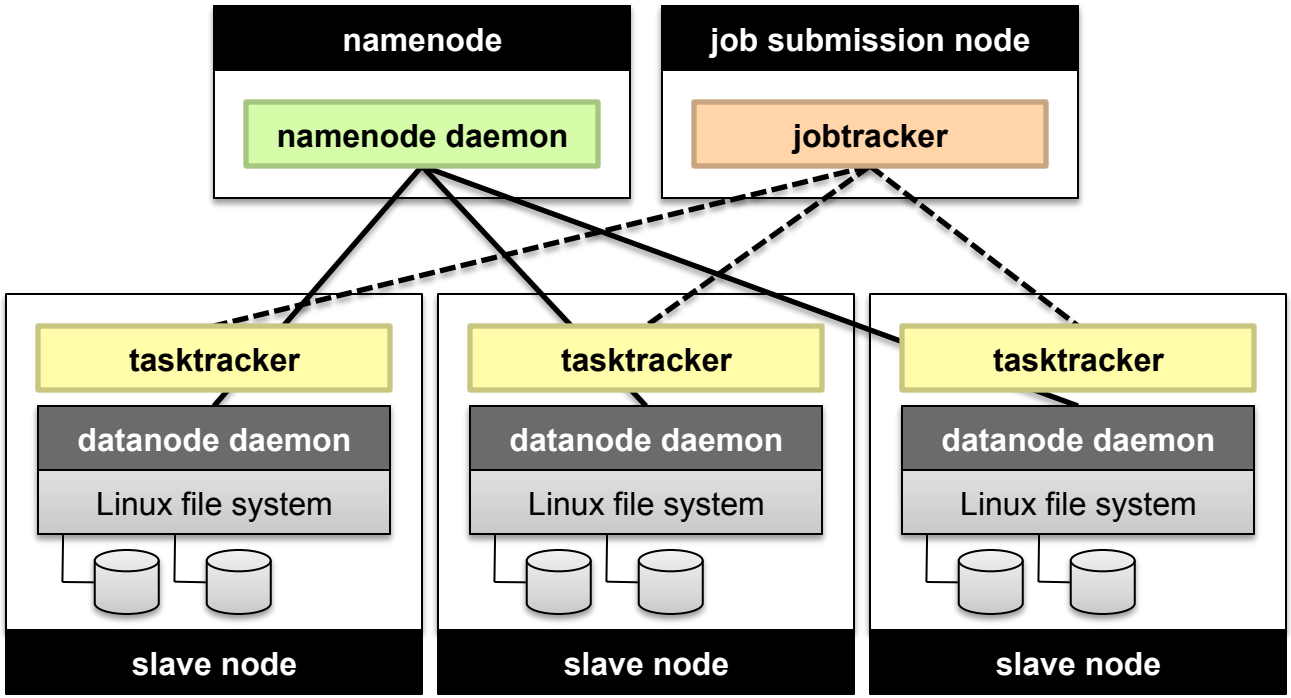


Spark works even if the RDDs are *partially* cached!

Spark Architecture

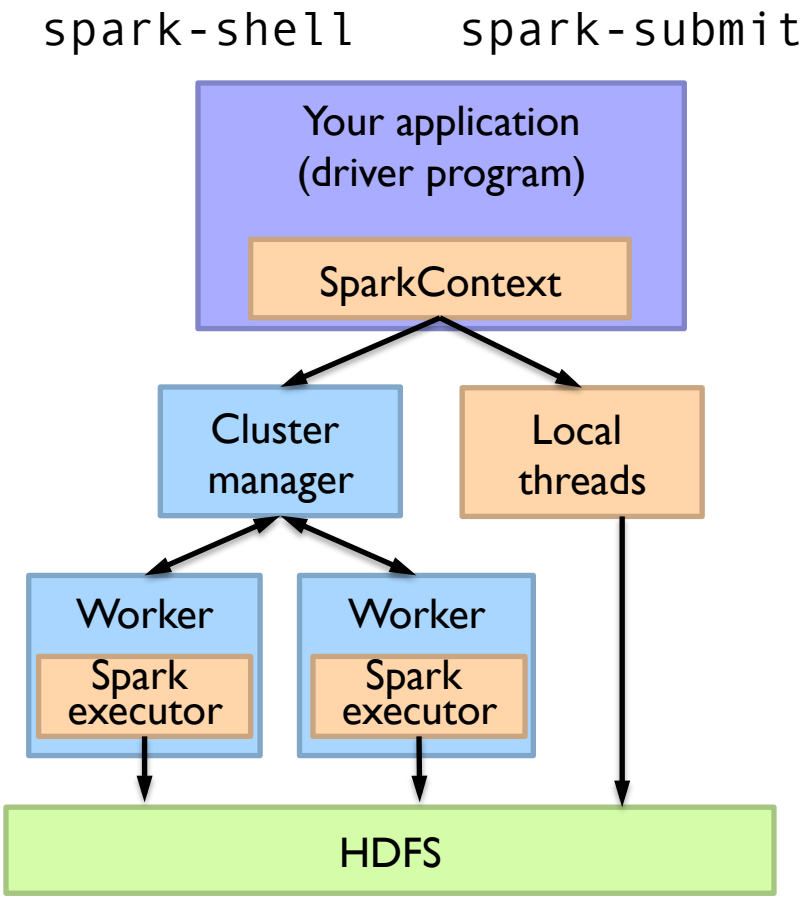


Compare


namenode
namenode daemon
job submission node
jobtracker
tasktracker
datanode daemon
Linux file system
slave node
tasktracker
datanode daemon
Linux file system
slave node
tasktracker
datanode daemon
Linux file system
slave node

Spark Programs

Scala, Java, Python, R

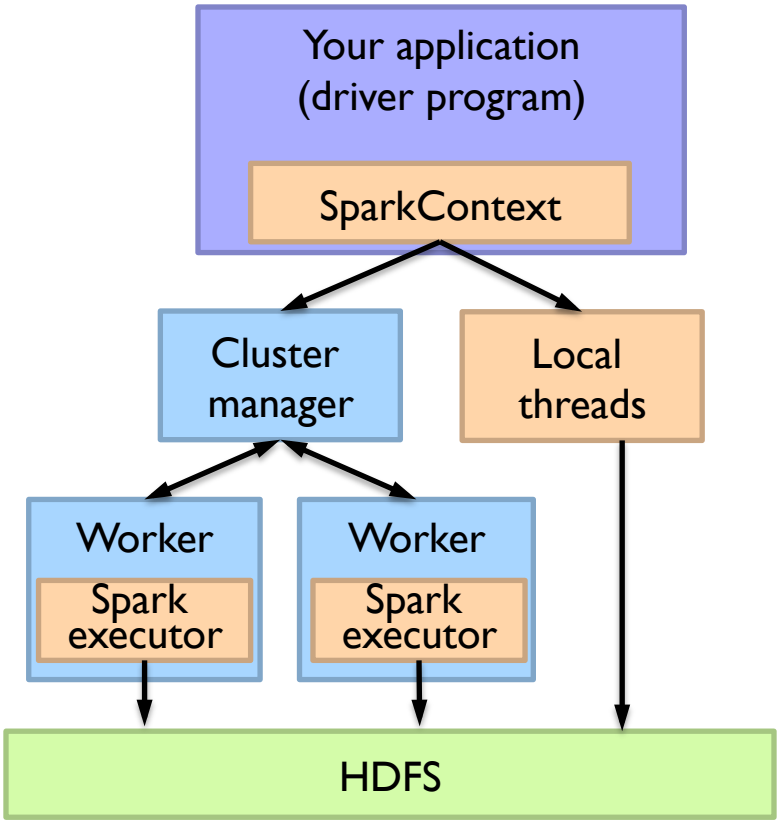


Spark context: tells the framework where to find the cluster

Use the Spark context to create RDDs

Spark Driver

`spark-shell` `spark-submit`



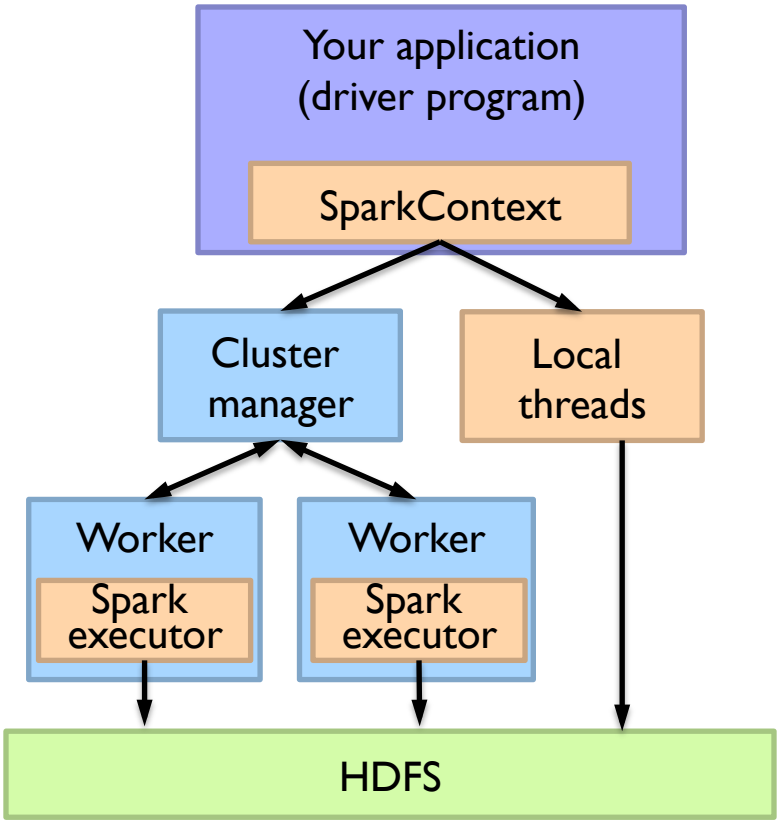

```
val textFile =
  sc.textFile(args.input())

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`



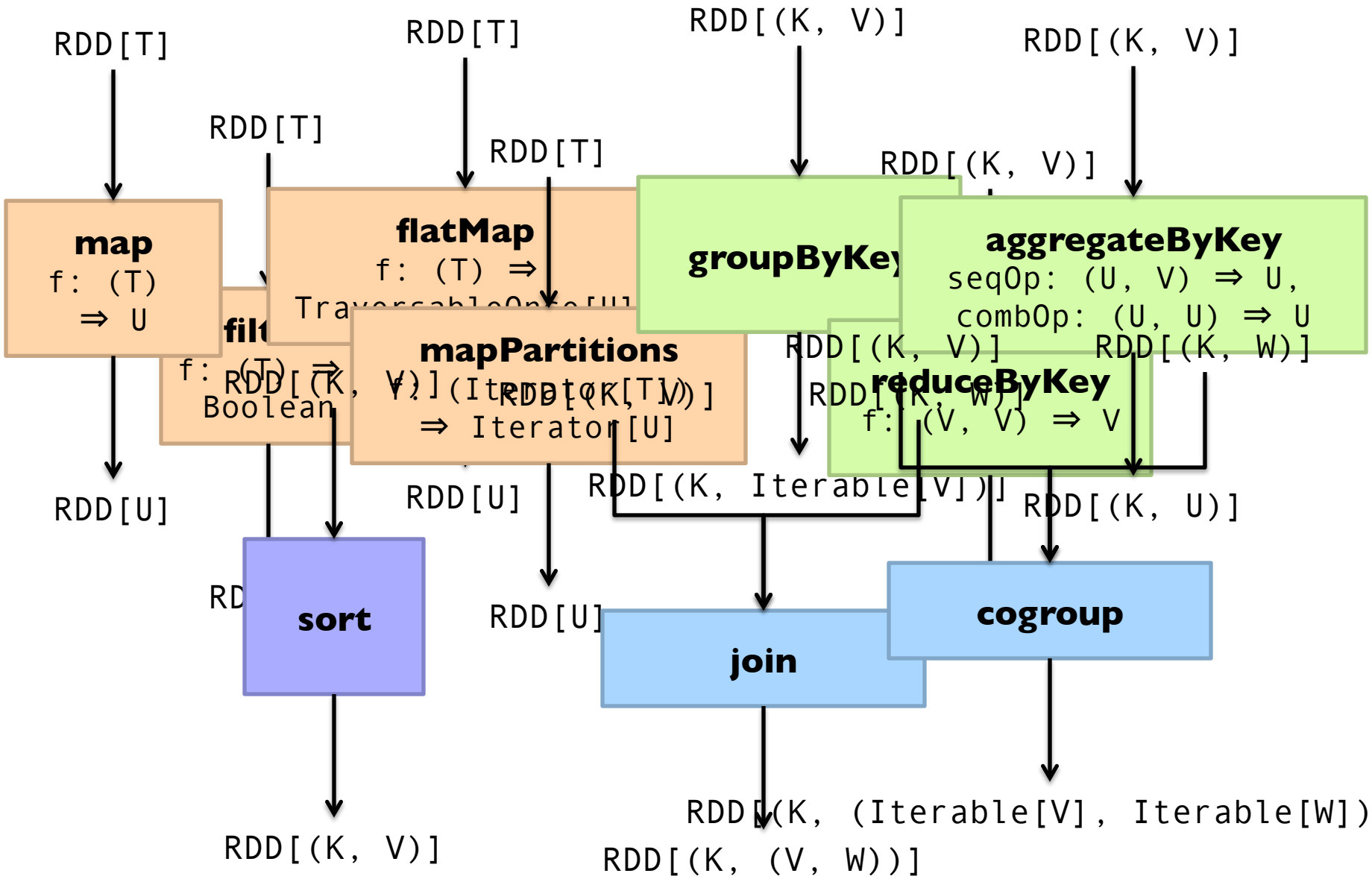

```
val textFile =
  sc.textFile(args.input())

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

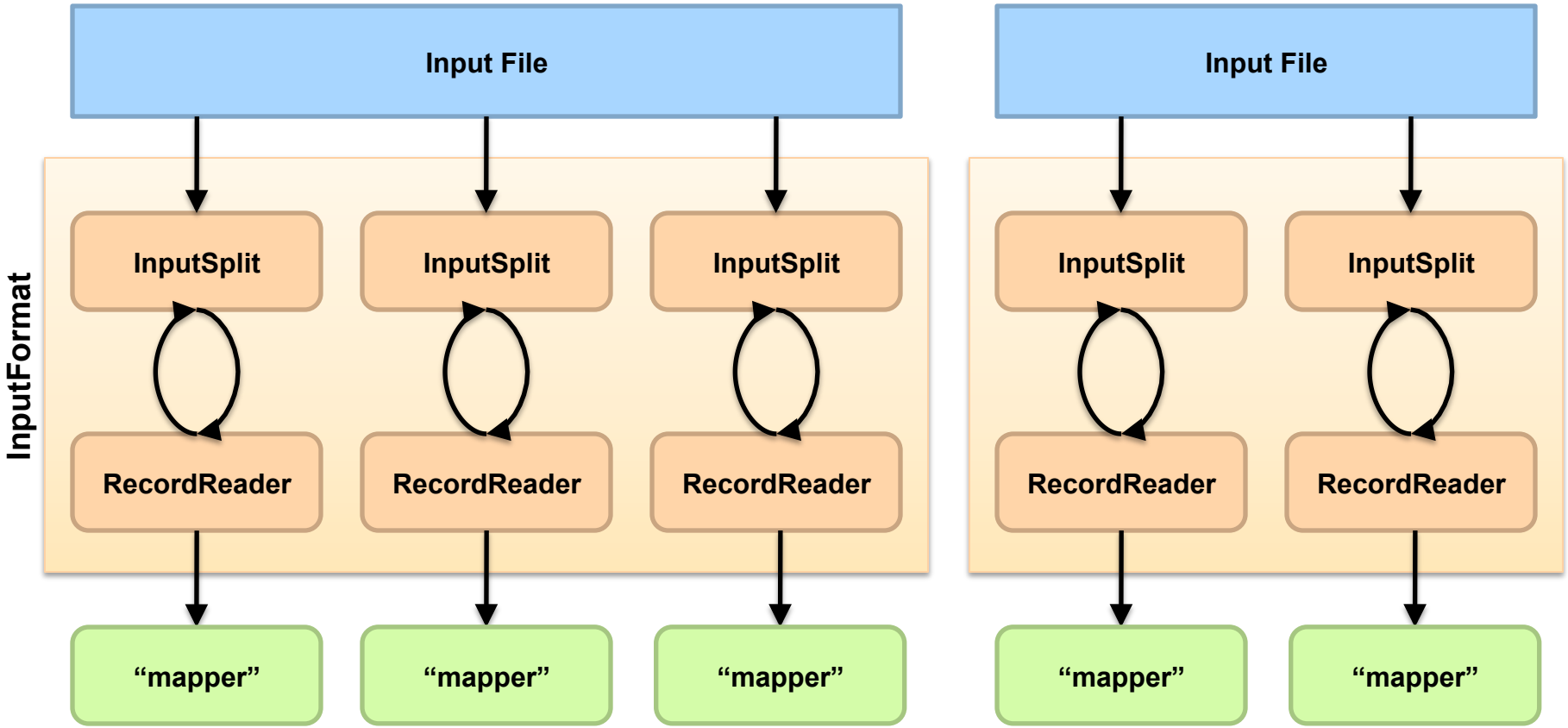
Note: you can run code "locally", integrate cluster-computed values!

Beware of the collect action!

Spark Transformations

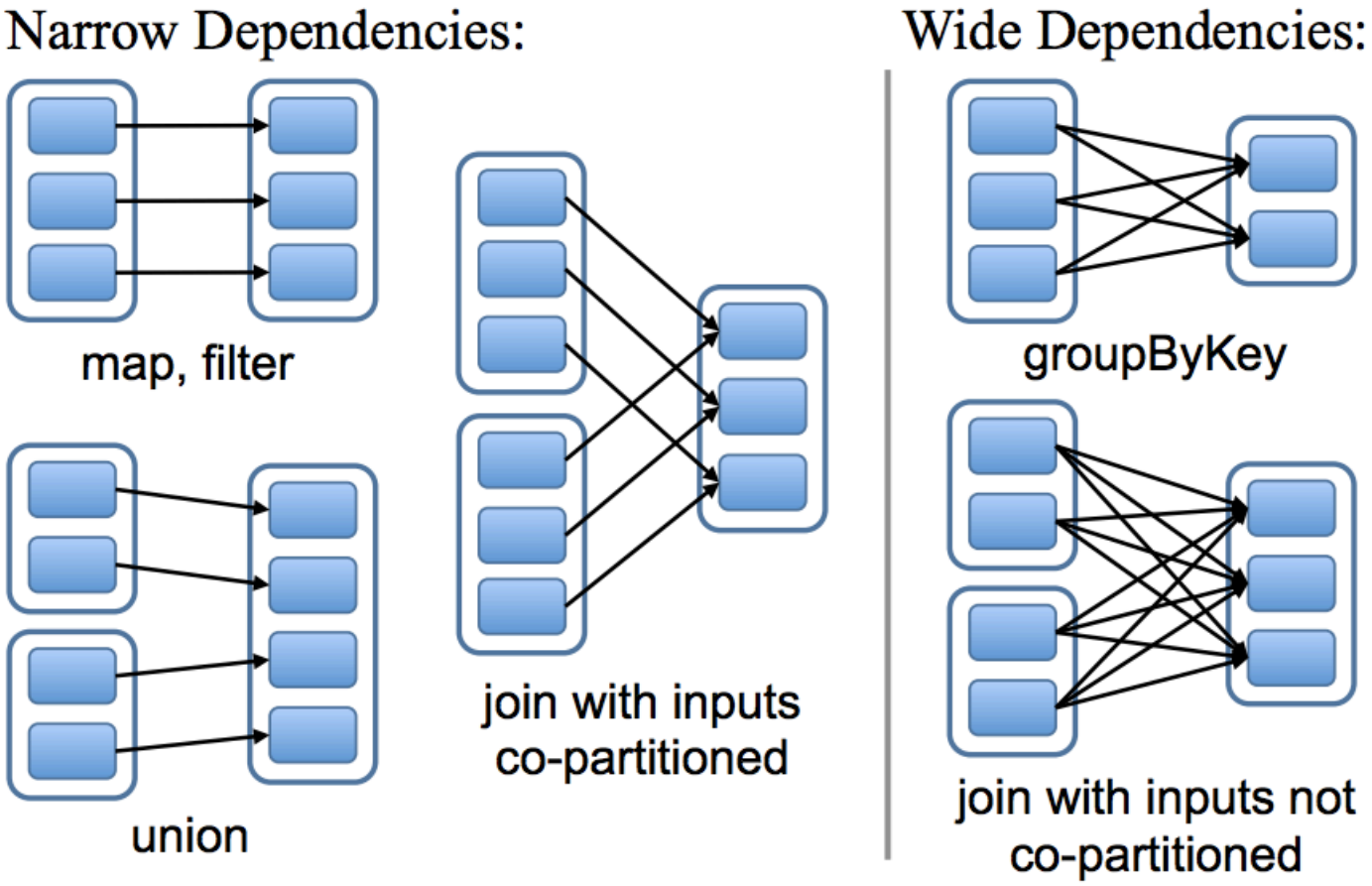


Starting Points

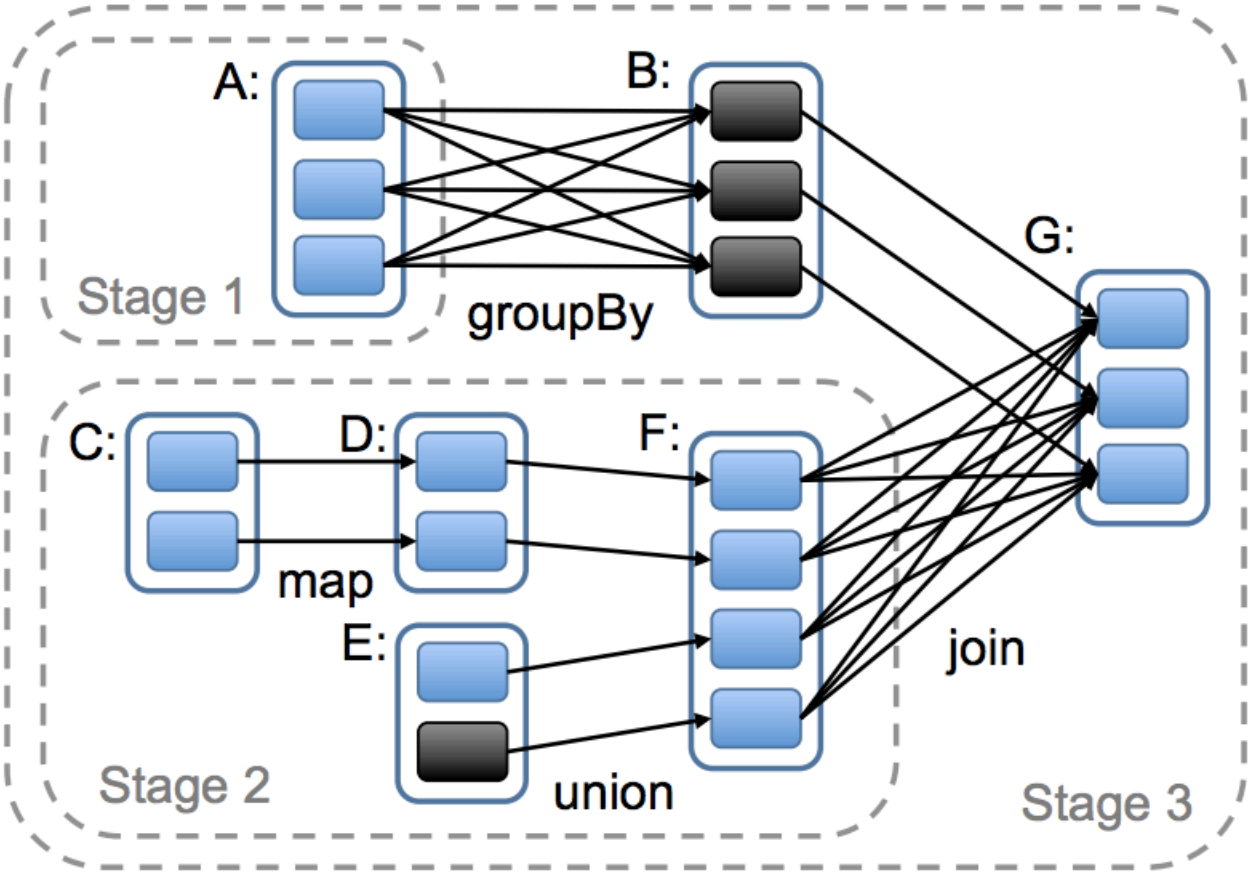


Source: redrawn from a slide by Cloduera, cc-licensed

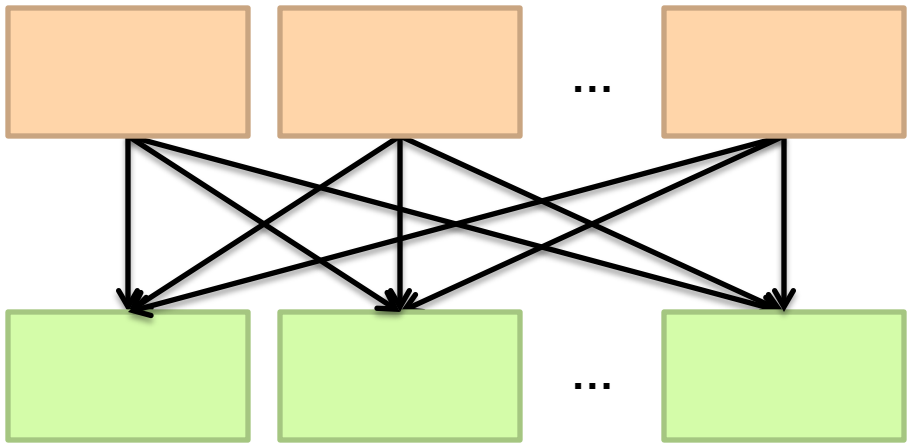
Physical Operators



Execution Plan

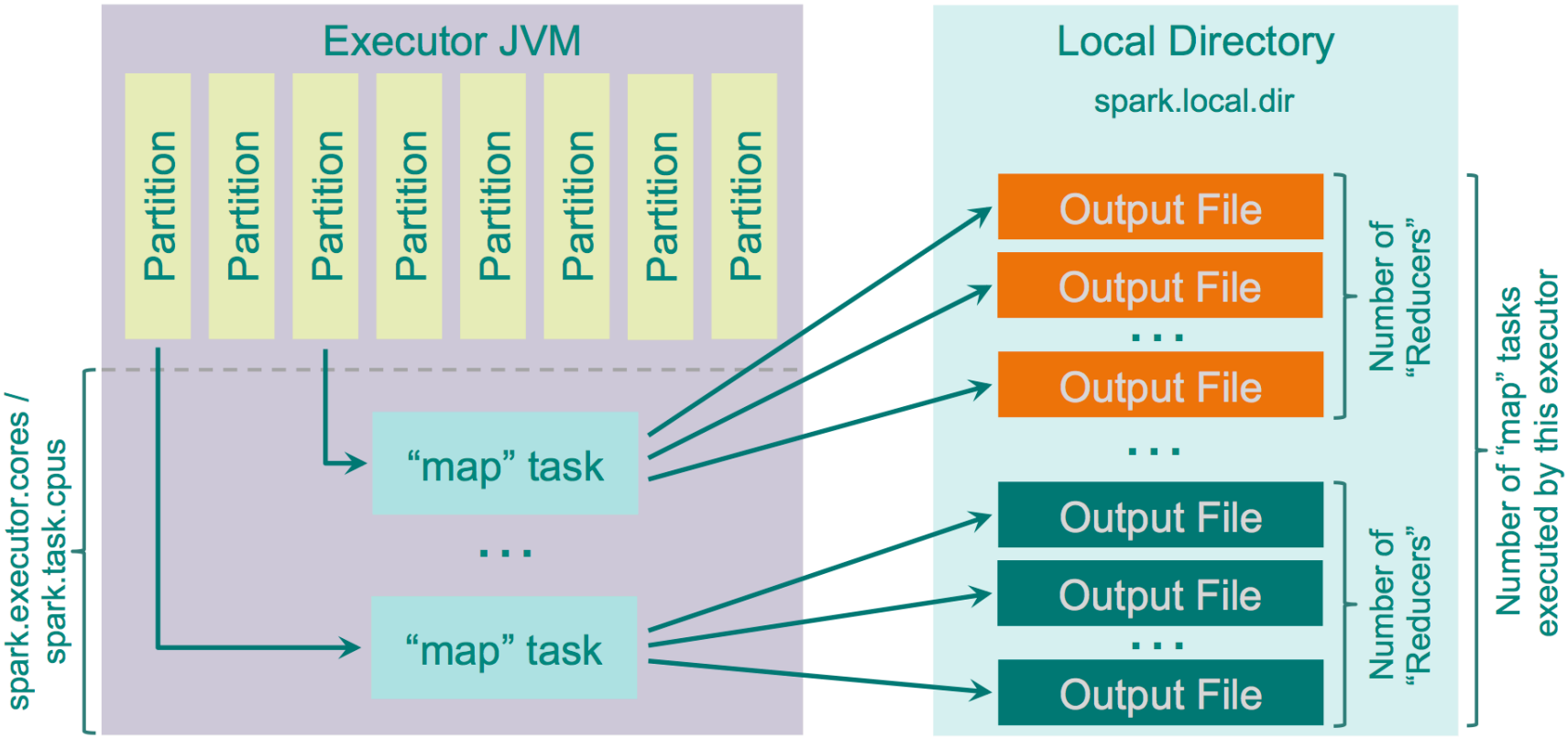


Can’t avoid this!



Spark Shuffle Implementations

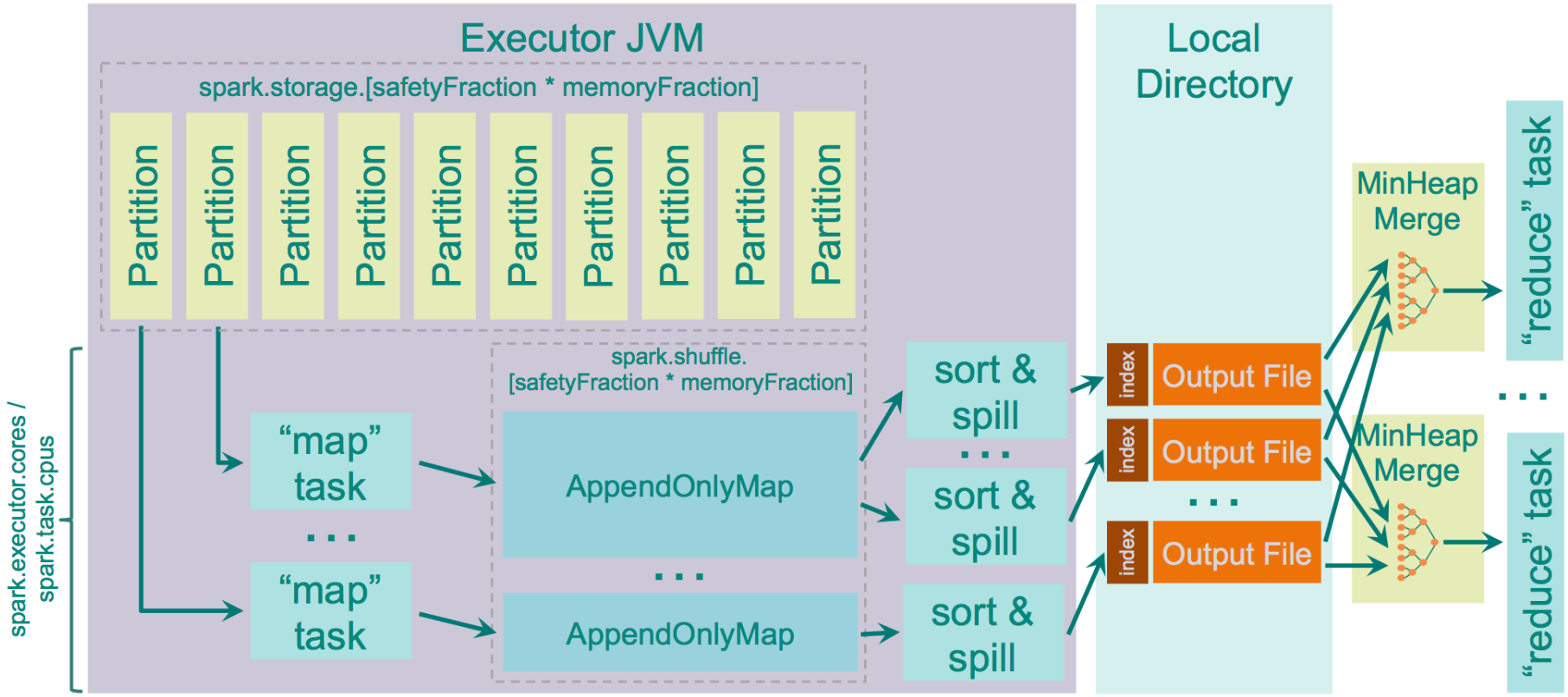
Hash shuffle



Source: http://0x0fff.com/spark-architecture-shuffle/

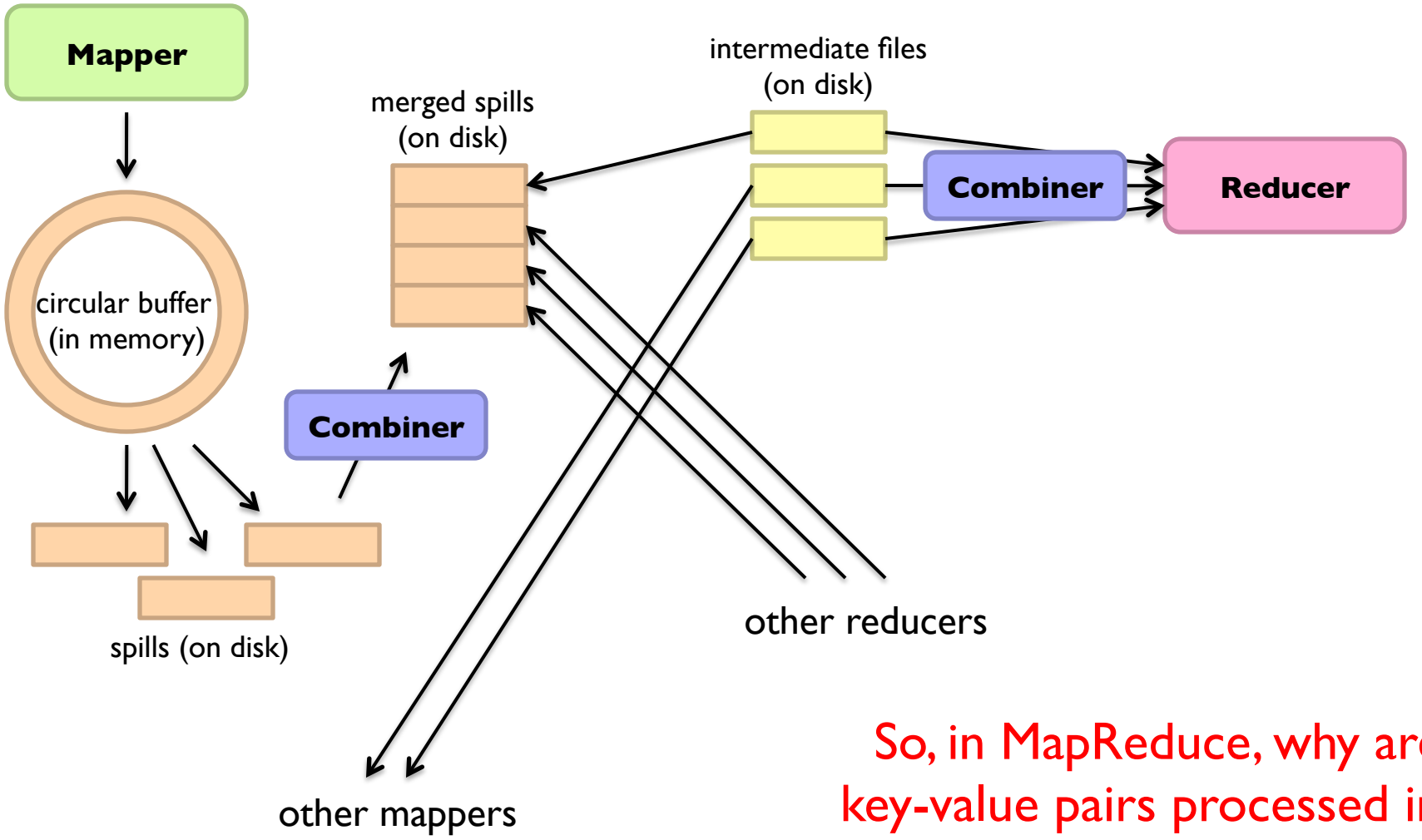
Spark Shuffle Implementations

Sort shuffle



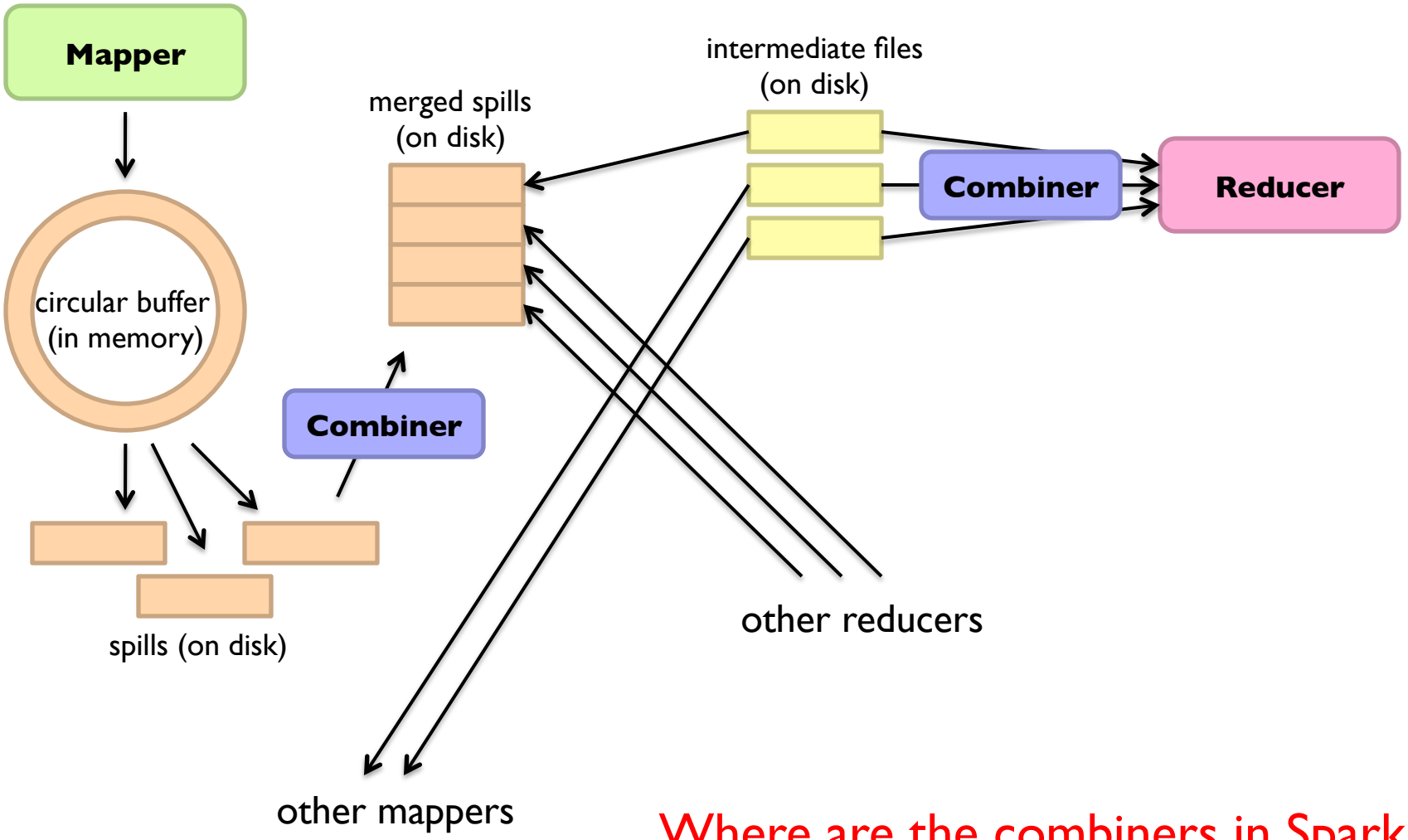
Source: http://0x0fff.com/spark-architecture-shuffle/

Remember this?



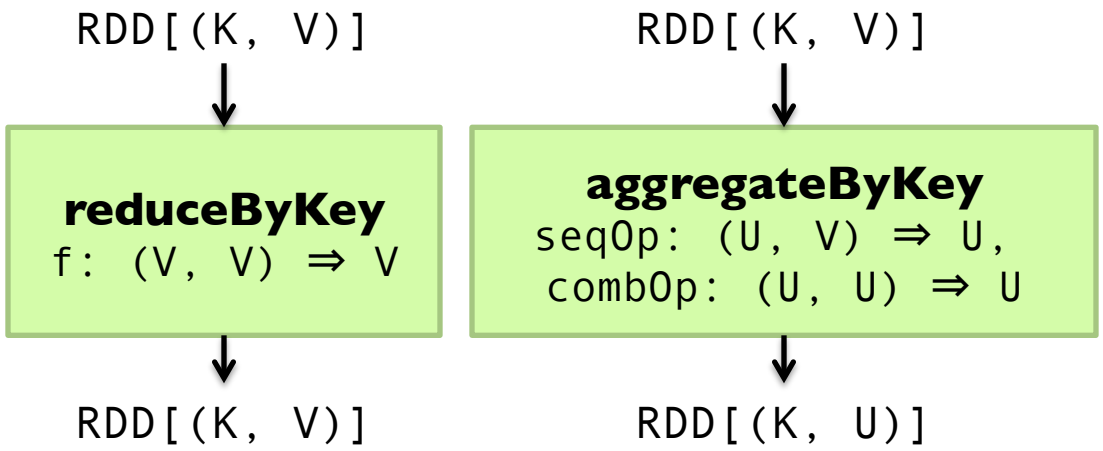
So, in MapReduce, why are key-value pairs processed in sorted order in the reducer?

Remember this?



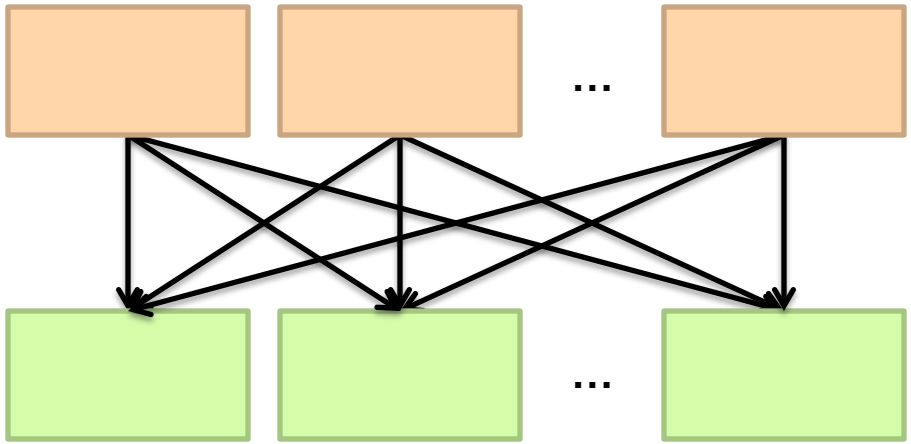
Where are the combiners in Spark?

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

Source: Wikipedia (Mahout)

Two superpowers:

Associativity
Commutativity
(sorting)

What follows… very basic category theory…

The Power of Associativity

You can put parenthesis where ever you want!

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

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

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

Credit to Oscar Boykin for the idea behind these slides

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 , $\oplus$)

$\oplus : M \times M \rightarrow M$, s.t., $\forall\, m_1, m_2, m_3 \ni M$

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

Monoid = Semigroup + identity

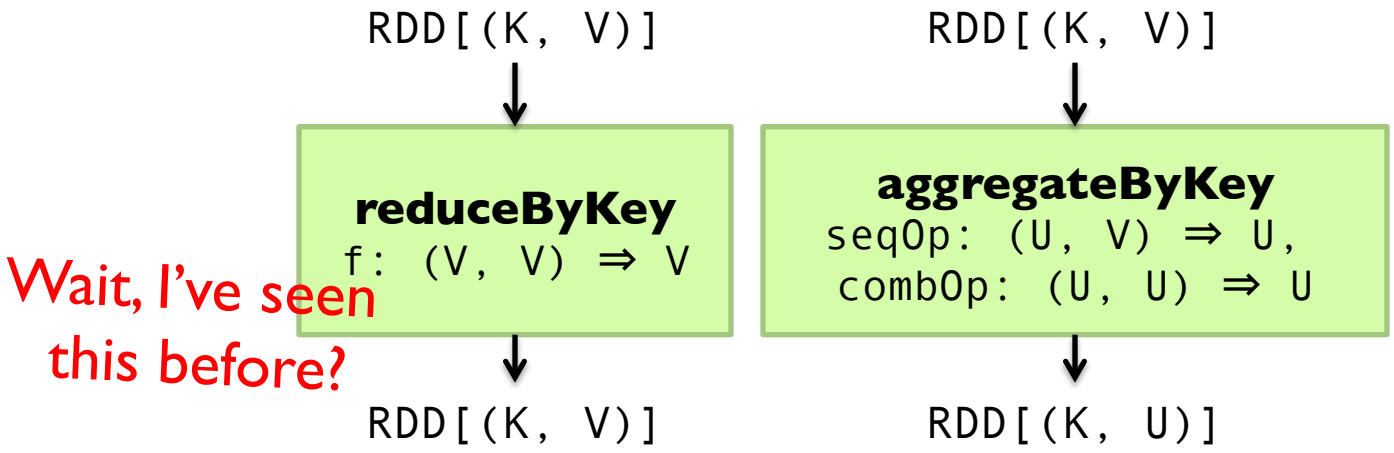
ε s.t., $\varepsilon \oplus m = m \oplus \varepsilon = m$, $\forall\, m \ni M$

Commutative Monoid = Monoid + commutativity

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

A few examples?

Back to these…



Computing the mean v1 (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, ...])
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)
```

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, ...])
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)
```

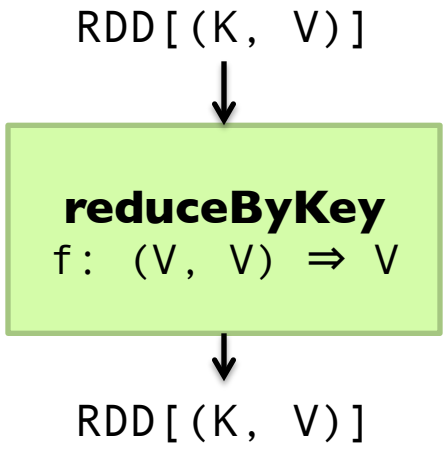
Computing the mean v3 (again)

```
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 [(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))
```

Wait, I've seen this before?

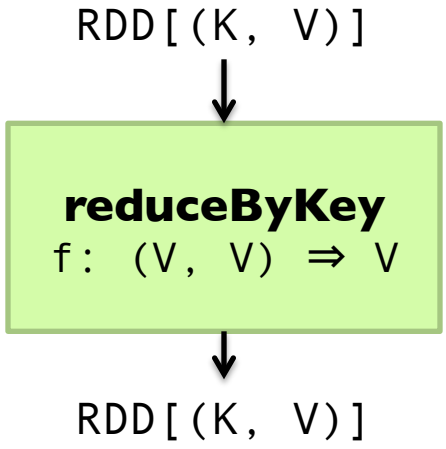


Co-occurrence Matrix: Stripes

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ 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)
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

What about this?

- 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

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, b_1) → 3
(a, b_2) → 12
(a, b_3) → 7
(a, b_4) → 1
…

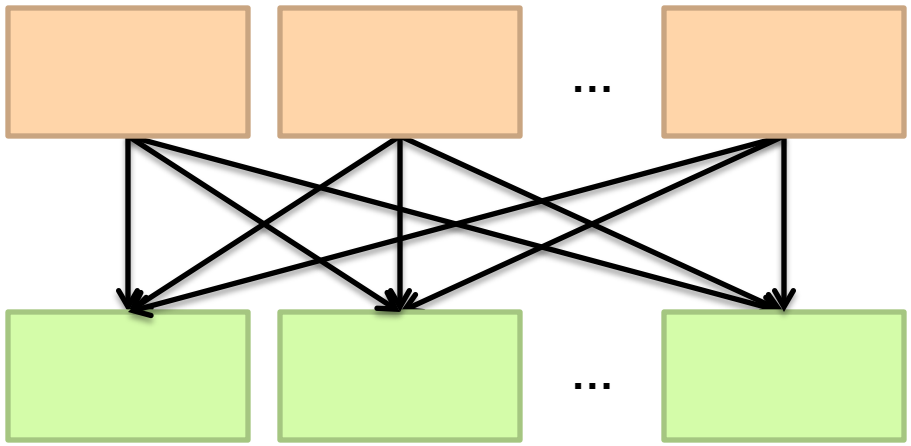
(a, b_1) → 3 / 32
(a, b_2) → 12 / 32
(a, b_3) → 7 / 32
(a, b_4) → 1 / 32
…

- 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!

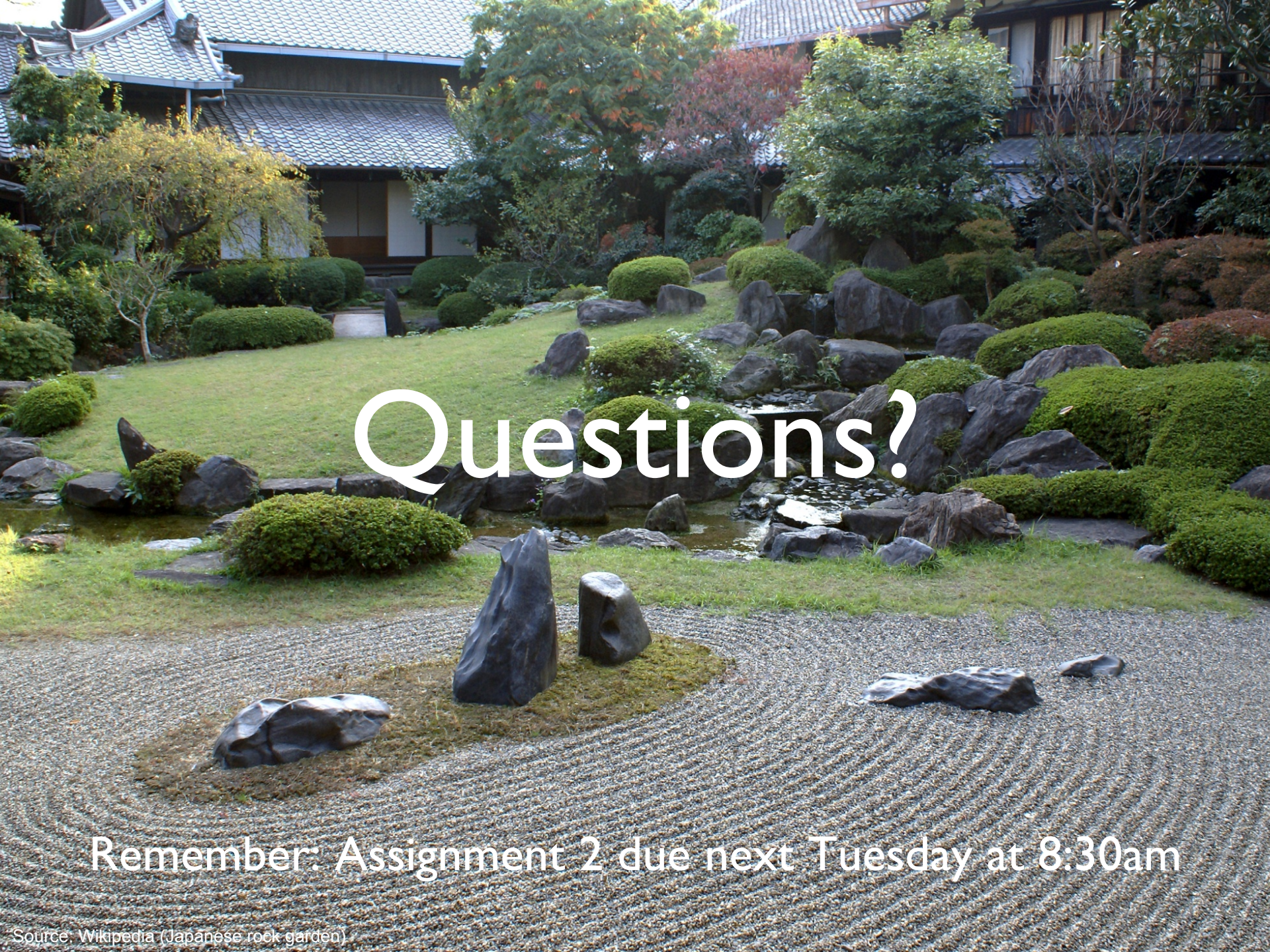

The datacenter *is* the computer!
What's the instruction set?
Source: Google

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)

Source: Wikipedia (Walnut)



Source: Wikipedia (Japanese rock garden)