



# Big Data Infrastructure

CS 489/698 Big Data Infrastructure (Winter 2017)

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

January 19, 2017

Jimmy Lin

David R. Cheriton School of Computer Science  
University of Waterloo

These slides are available at <http://lintool.github.io/bigdata-2017w/>



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



# An Apt Quote

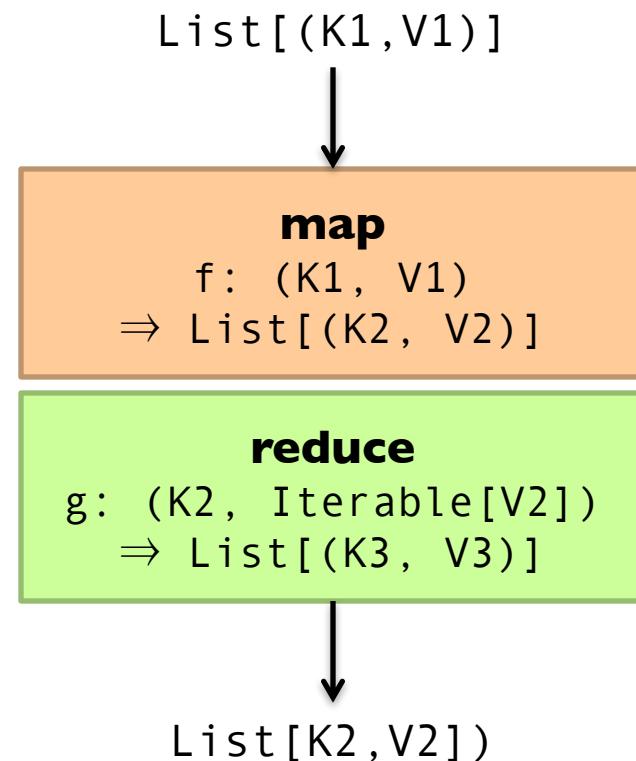
All problems in computer science can be solved by another level of indirection... Except for the problem of too many layers of indirection.

- David Wheeler

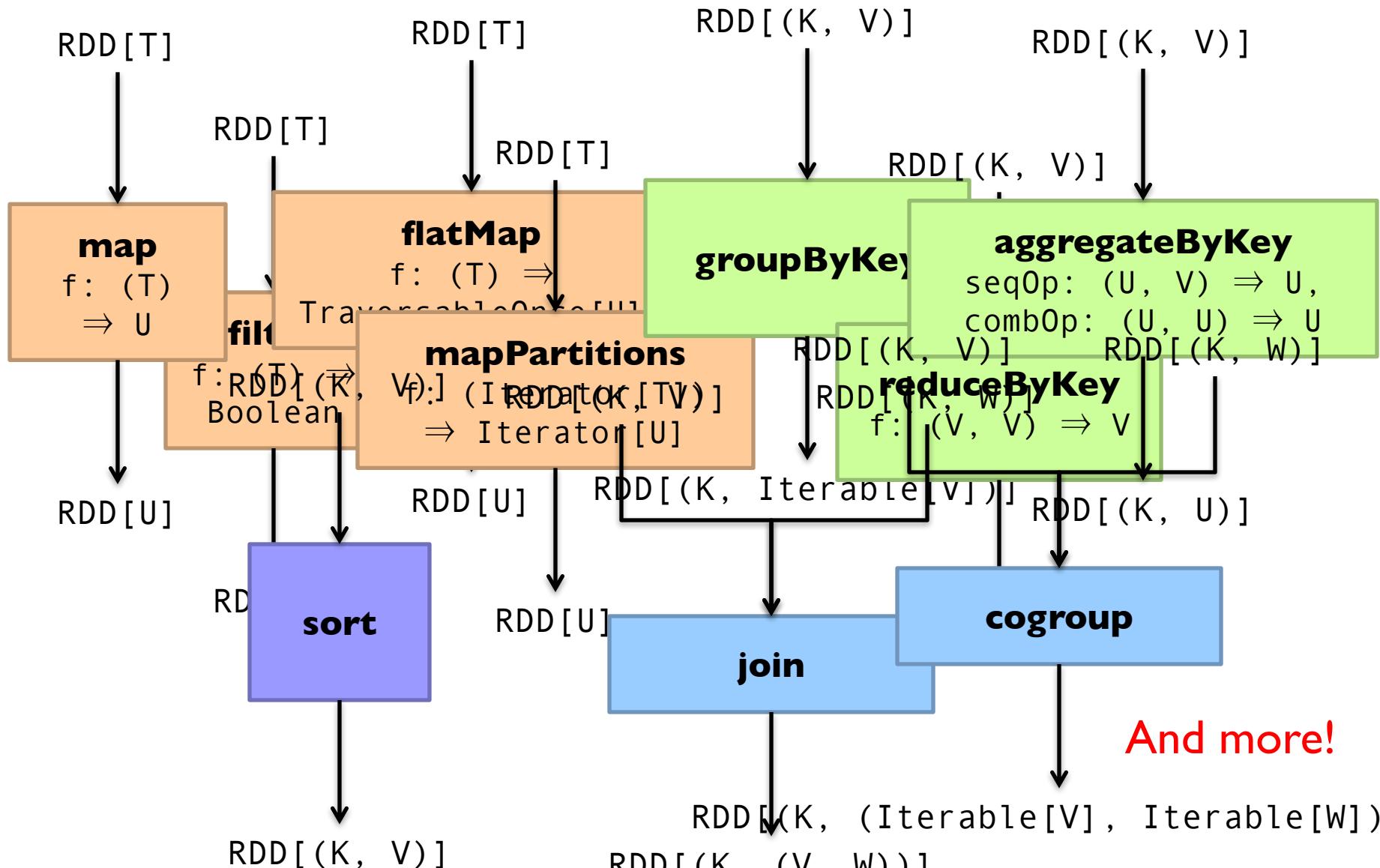
An aerial photograph of a large data center complex during sunset. The sky is a warm orange and yellow. In the foreground, there are several large white industrial buildings, some with flat roofs and others with green roofs. There are also numerous white shipping containers stacked in a row. A network of roads and parking lots connects the buildings. In the background, there are more industrial structures, a highway with traffic, and a vast, green, agricultural landscape stretching to the horizon.

The datacenter *is* the computer!  
What's the instruction set?  
What are the abstractions?

# MapReduce



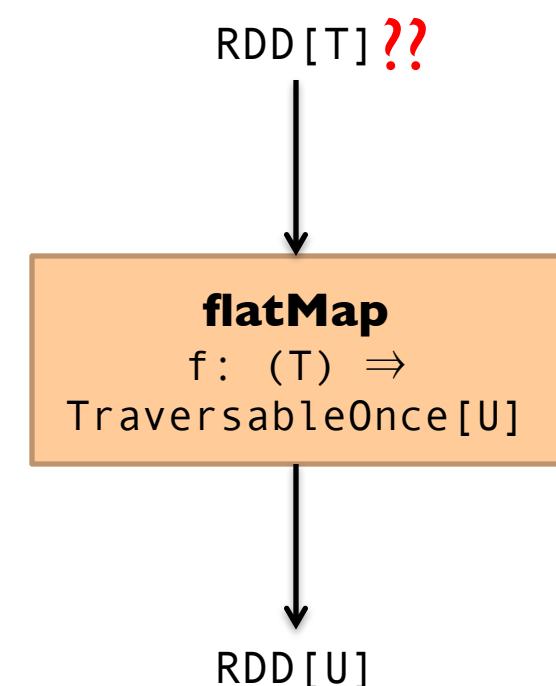
# Spark



# Spark Word Count

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

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



# What's an RDD?

## Resilient Distributed Dataset (RDD)

= immutable   = partitioned

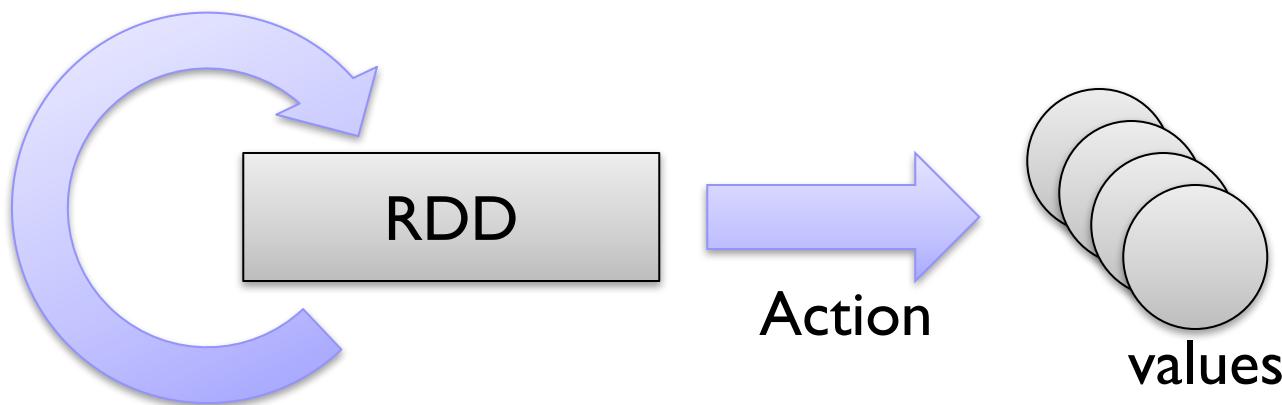
Wait, so how do you actually do anything?

Developers define *transformations* on RDDs

Framework keeps track of lineage

# 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((x, y) => x + y) ←
```

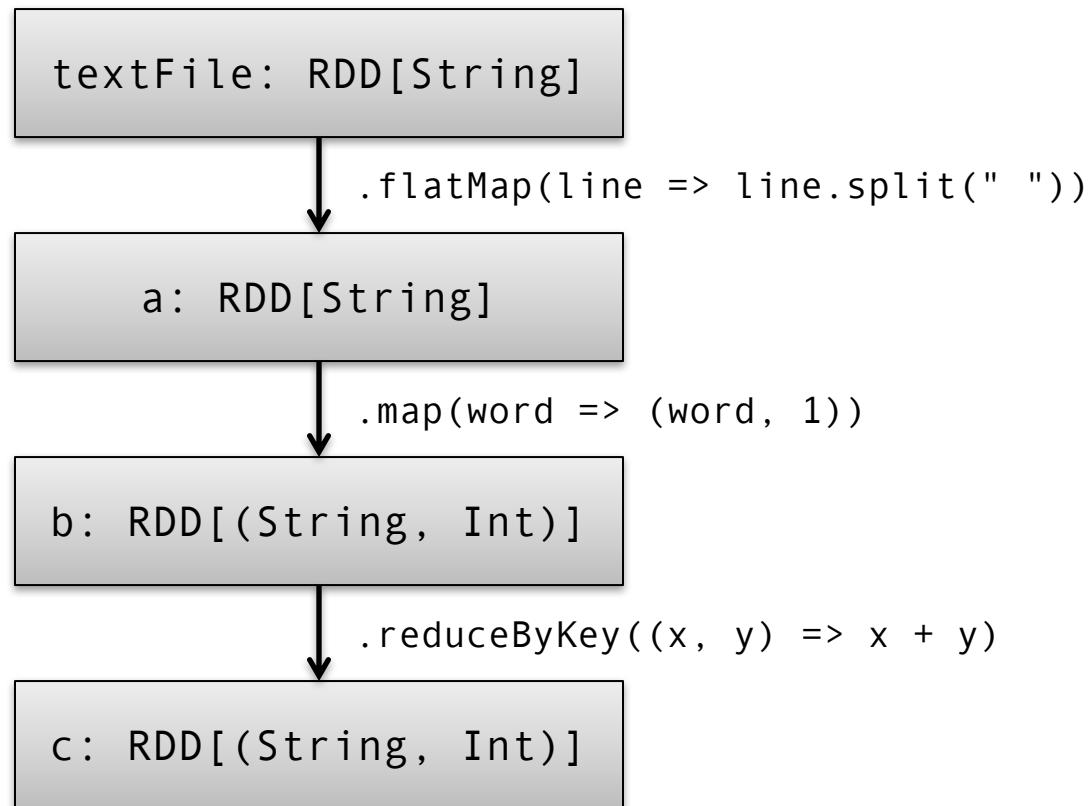
```
c.saveAsTextFile(args.output())
```

Action

Transformations

# RDDs and Lineage

On HDFS



Action!

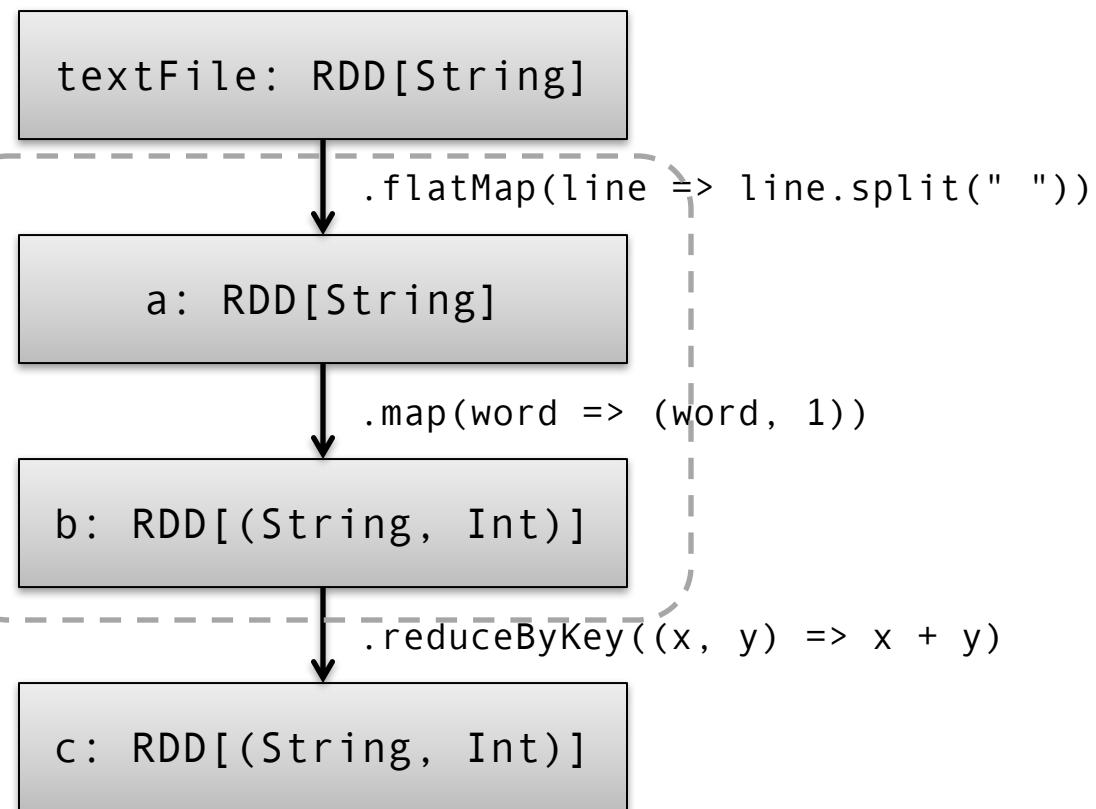
Remember,  
transformations are lazy!

# RDDs and Optimizations

Lazy evaluation creates optimization opportunities

On HDFS  
RDDs don't need  
to be materialized!

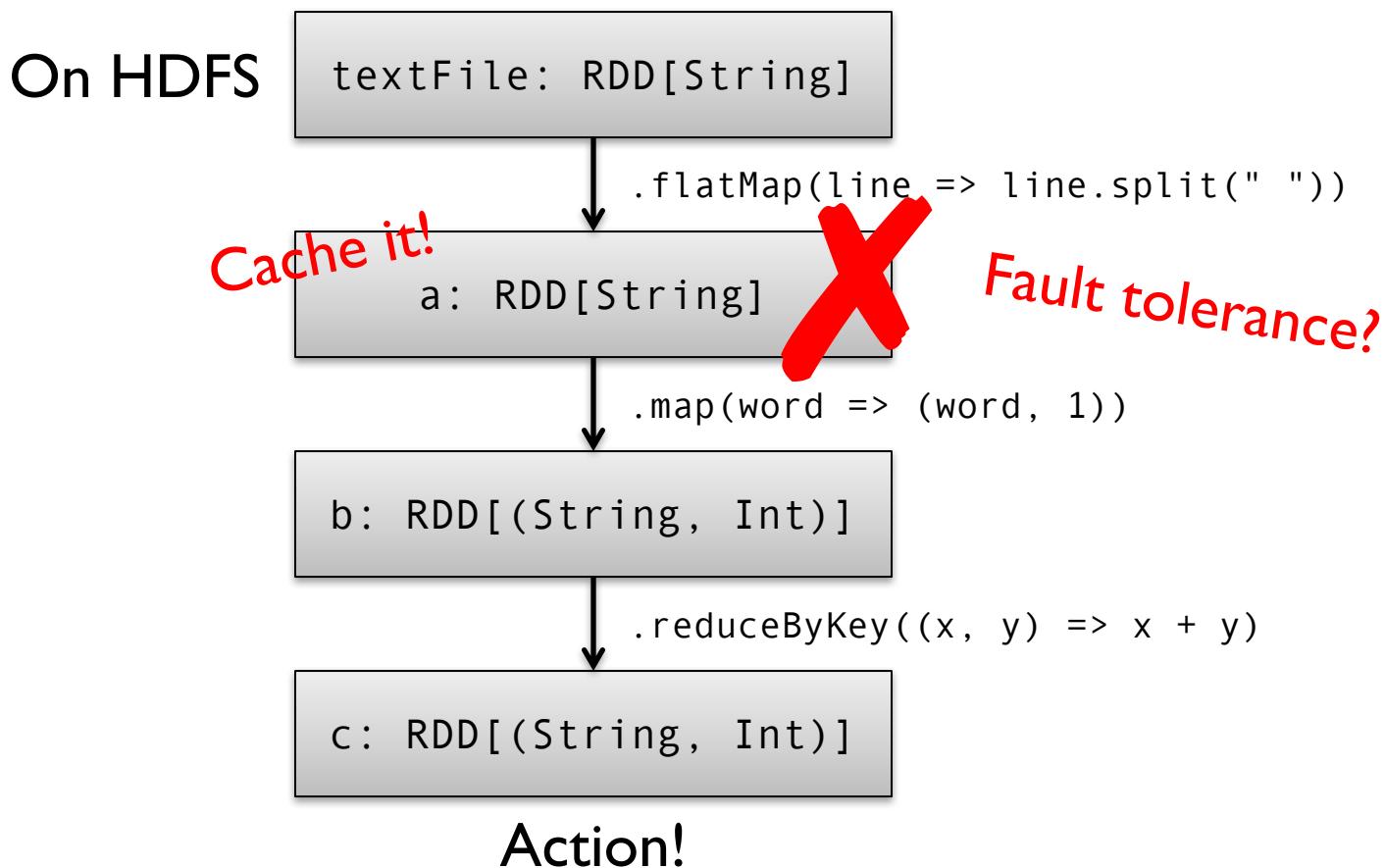
Want MM?



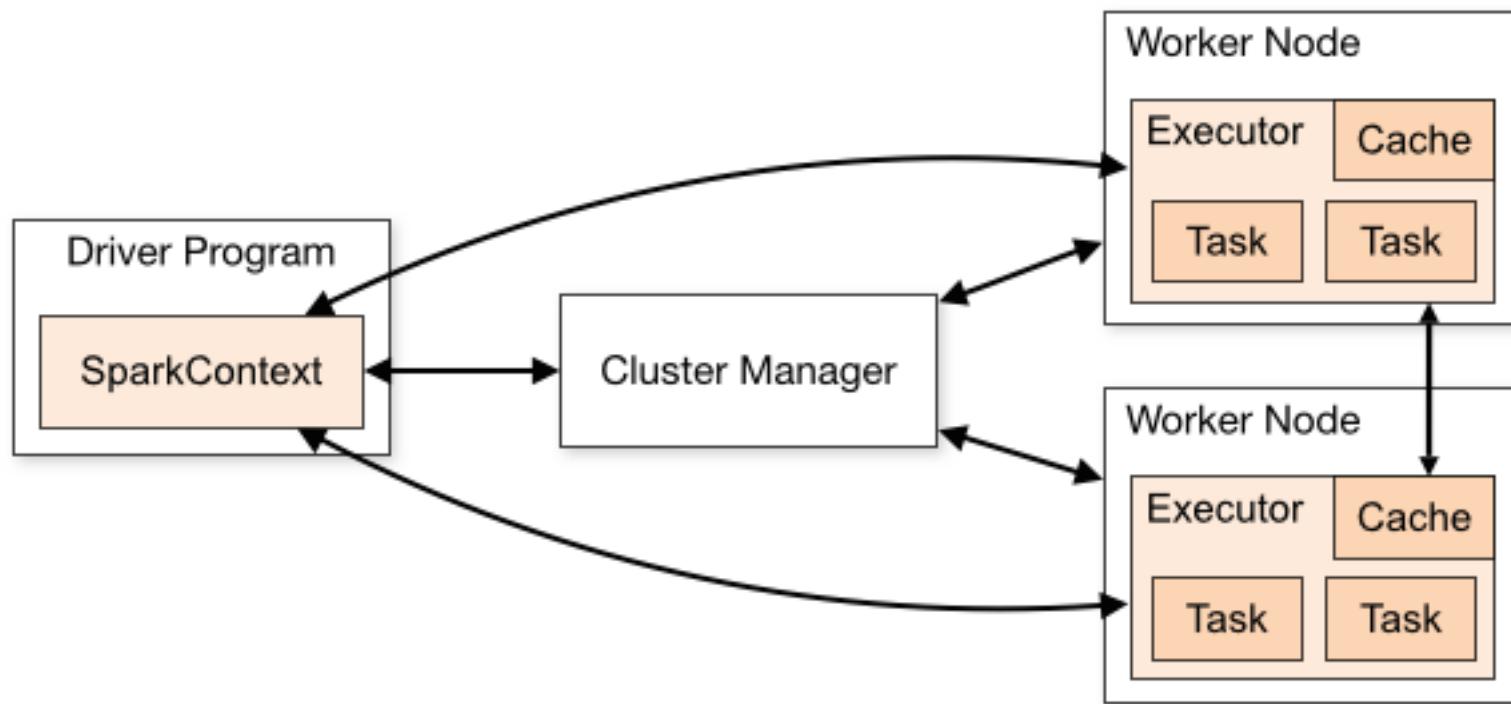
Action!

# RDDs and Caching

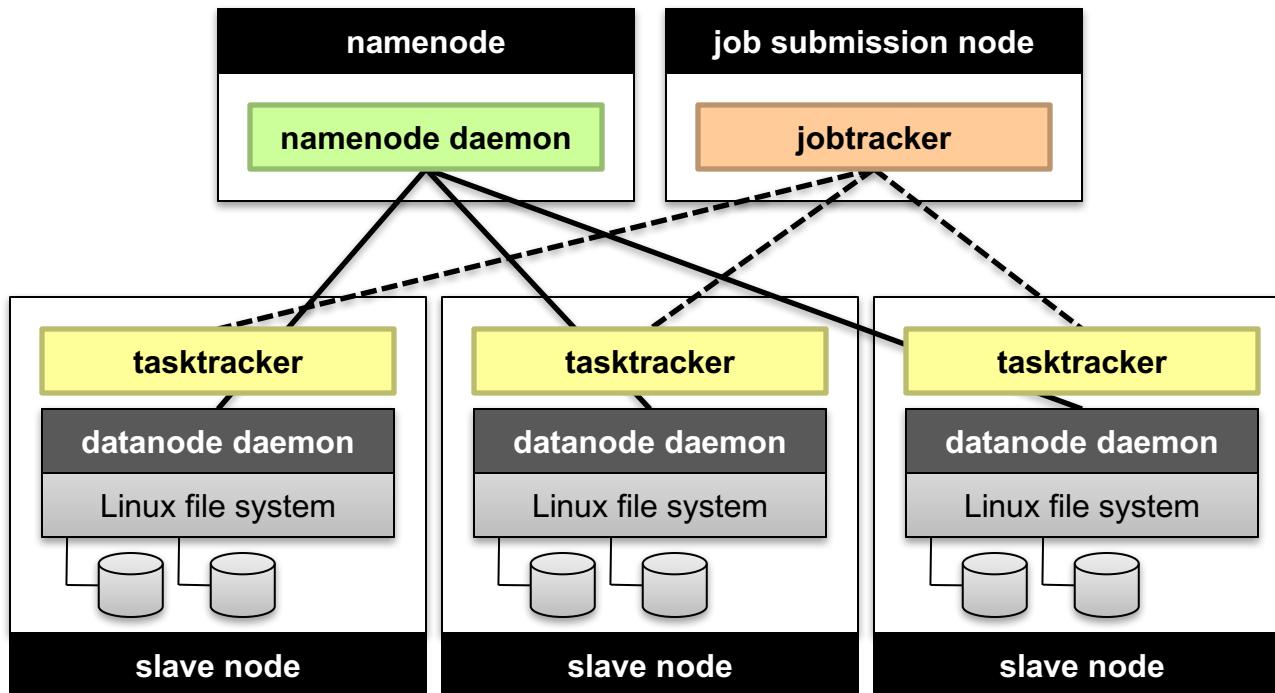
RDDs can be materialized in memory (and on disk)!



# Spark Architecture



# Compare



# An Apt Quote

All problems in computer science can be solved by another level of indirection... Except for the problem of too many layers of indirection.

- David Wheeler

# YARN

Hadoop's (original) limitations:

Can only run MapReduce

What if we want to run other distributed frameworks?

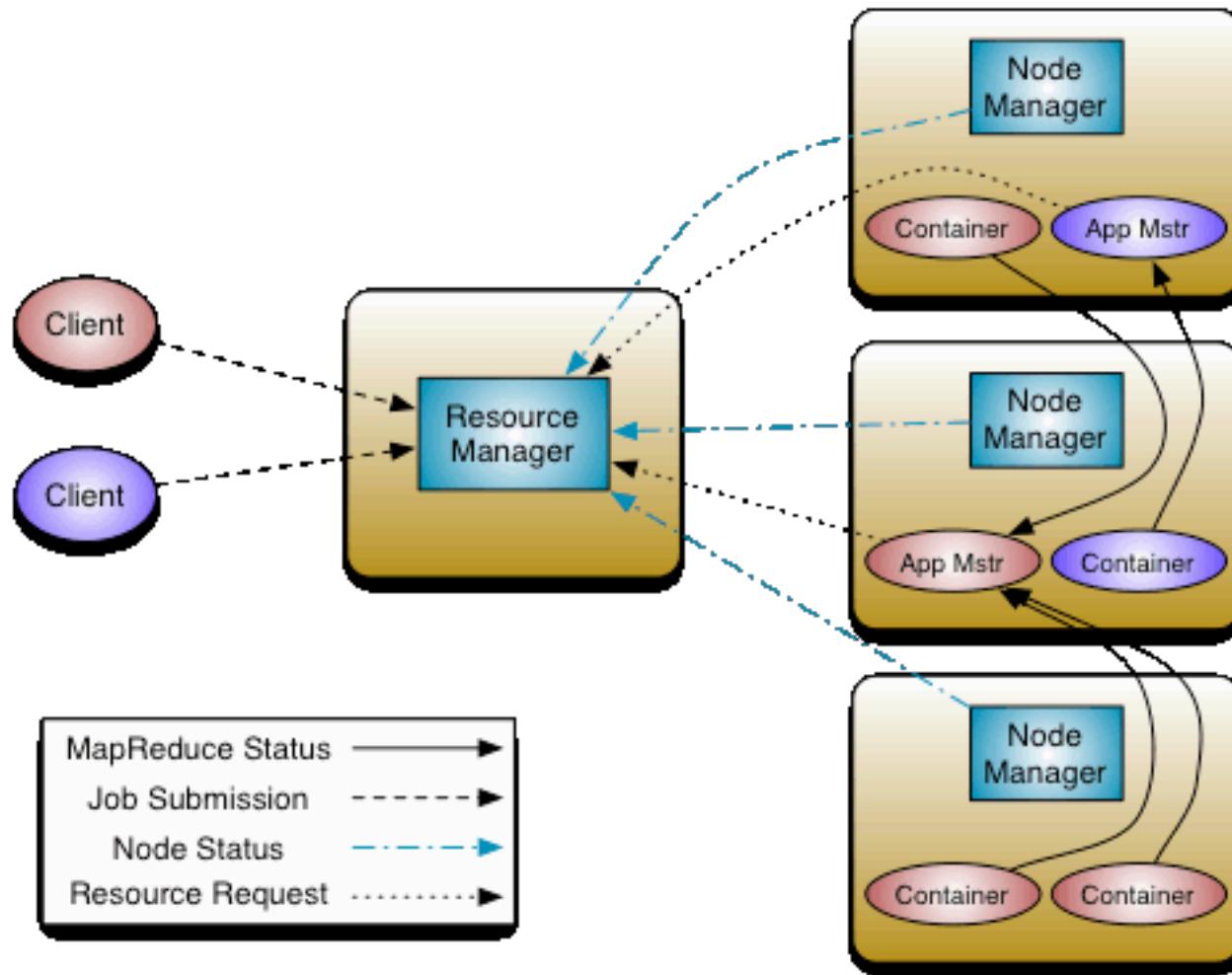
YARN = Yet-Another-Resource-Negotiator

Provides API to develop any generic distributed application

Handles scheduling and resource request

MapReduce (MR2) is one such application in YARN

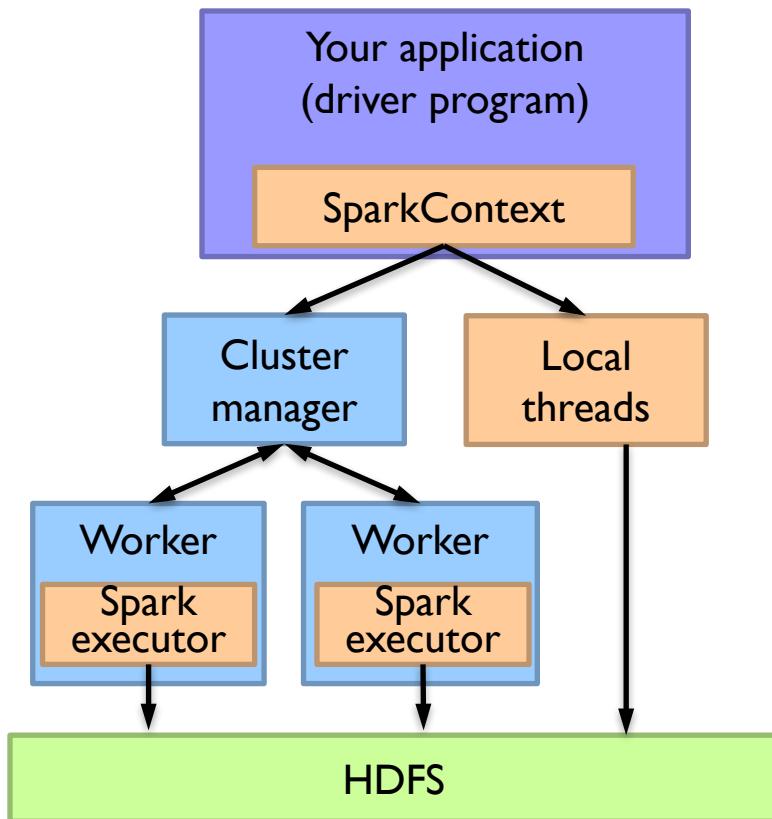
# YARN



# Spark Programs

Scala, Java, Python, R

spark-shell      spark-submit

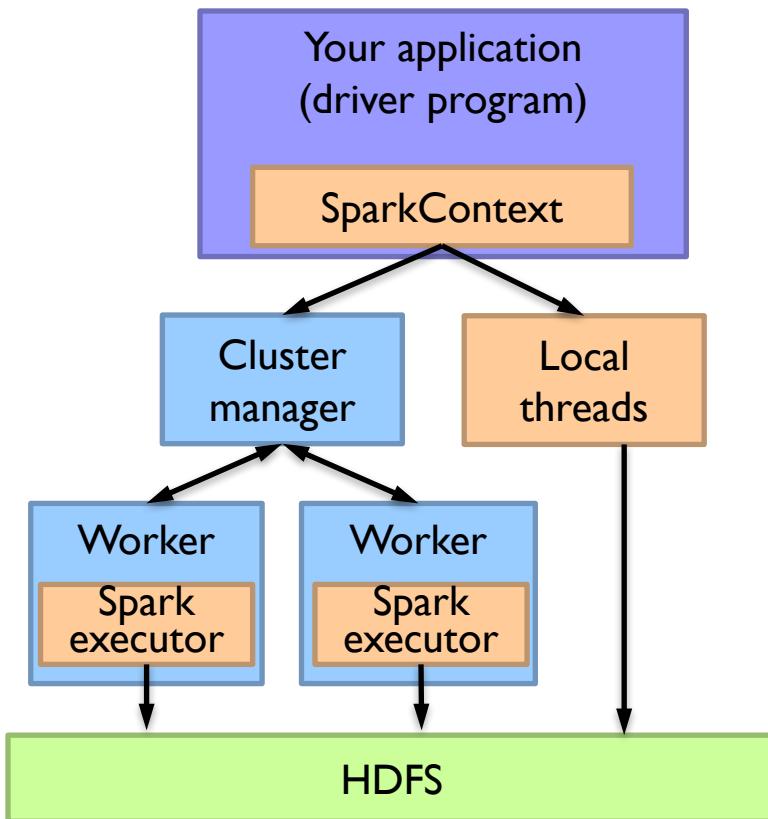


Spark context: tells the framework where to find the cluster

Use the Spark context to create RDDs

# Spark Driver

spark-shell      spark-submit

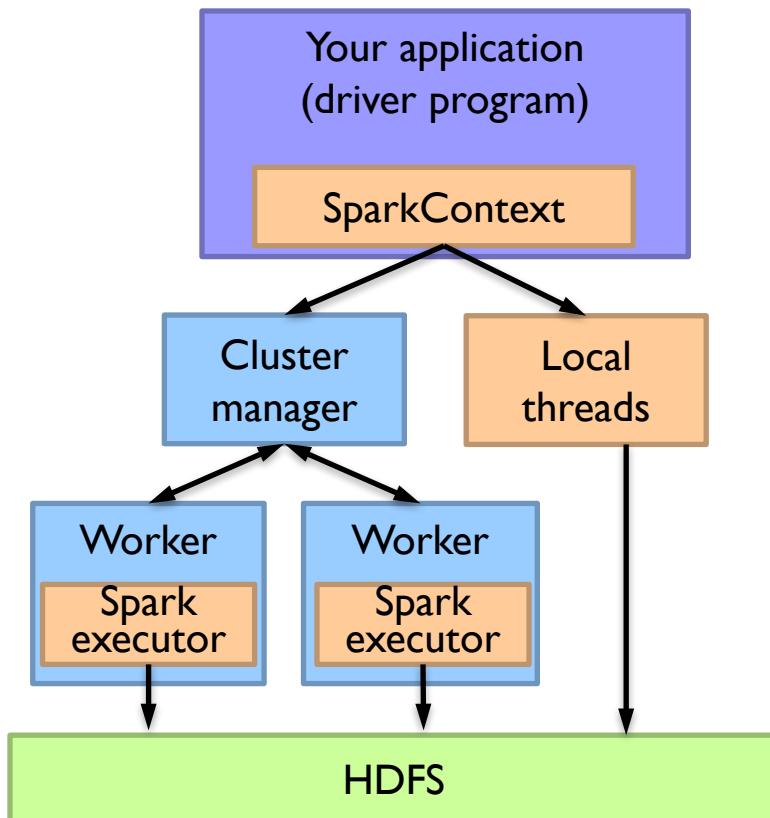


```
val textFile =  
  sc.textFile(args.input())  
  
textFile  
  .flatMap(line => tokenize(line))  
  .map(word => (word, 1))  
  .reduceByKey((x, y) => x + y)  
  .saveAsTextFile(args.output())
```

What's happening  
to the functions?

# Spark Driver

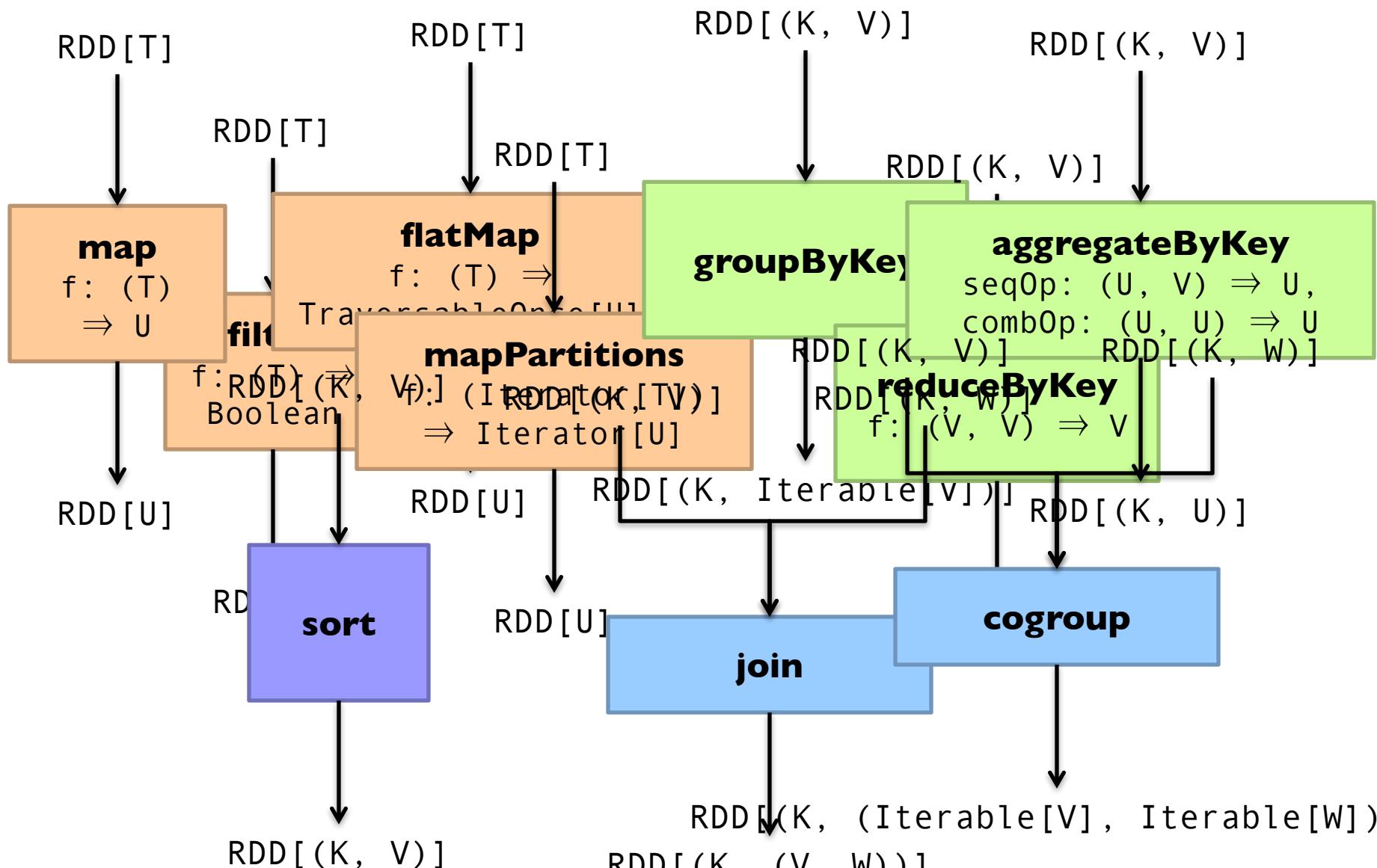
spark-shell      spark-submit



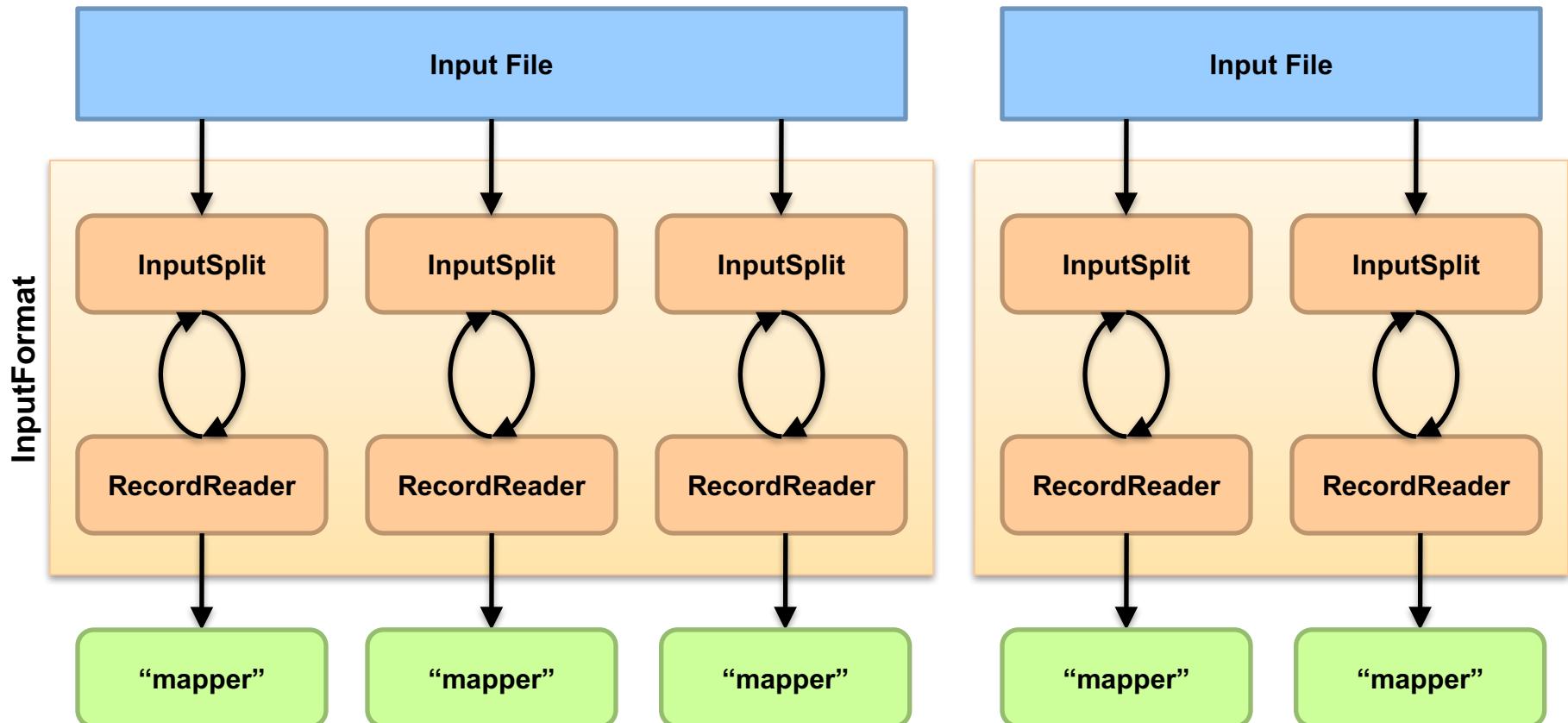
```
val textFile =  
  sc.textFile(args.input())  
  
textFile  
  .flatMap(line => tokenize(line))  
  .map(word => (word, 1))  
  .reduceByKey((x, y) => x + y)  
  .saveAsTextFile(args.output())
```

Note: you can run code “locally”,  
integrate cluster-computed values!  
Beware of the collect action!

# Spark Transformations

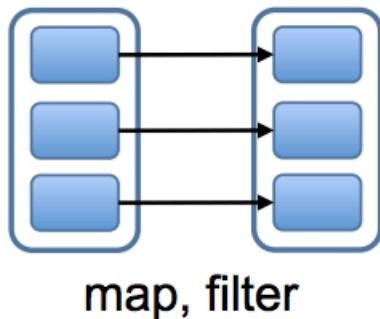


# Starting Points

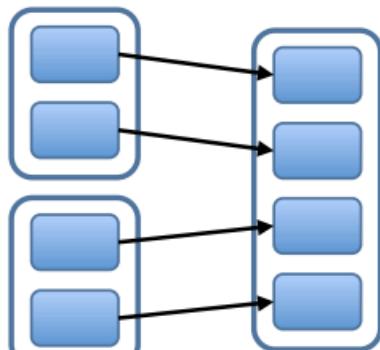


# Physical Operators

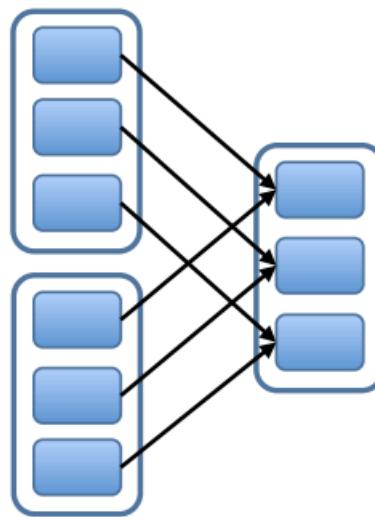
Narrow Dependencies:



map, filter

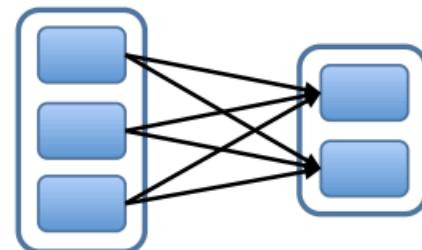


union

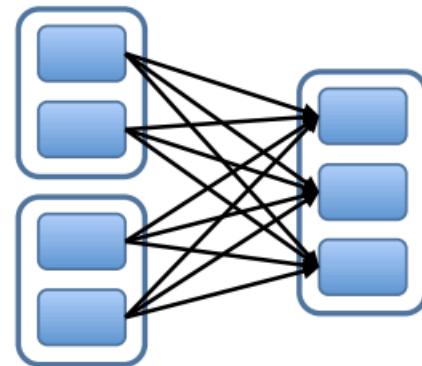


join with inputs  
co-partitioned

Wide Dependencies:

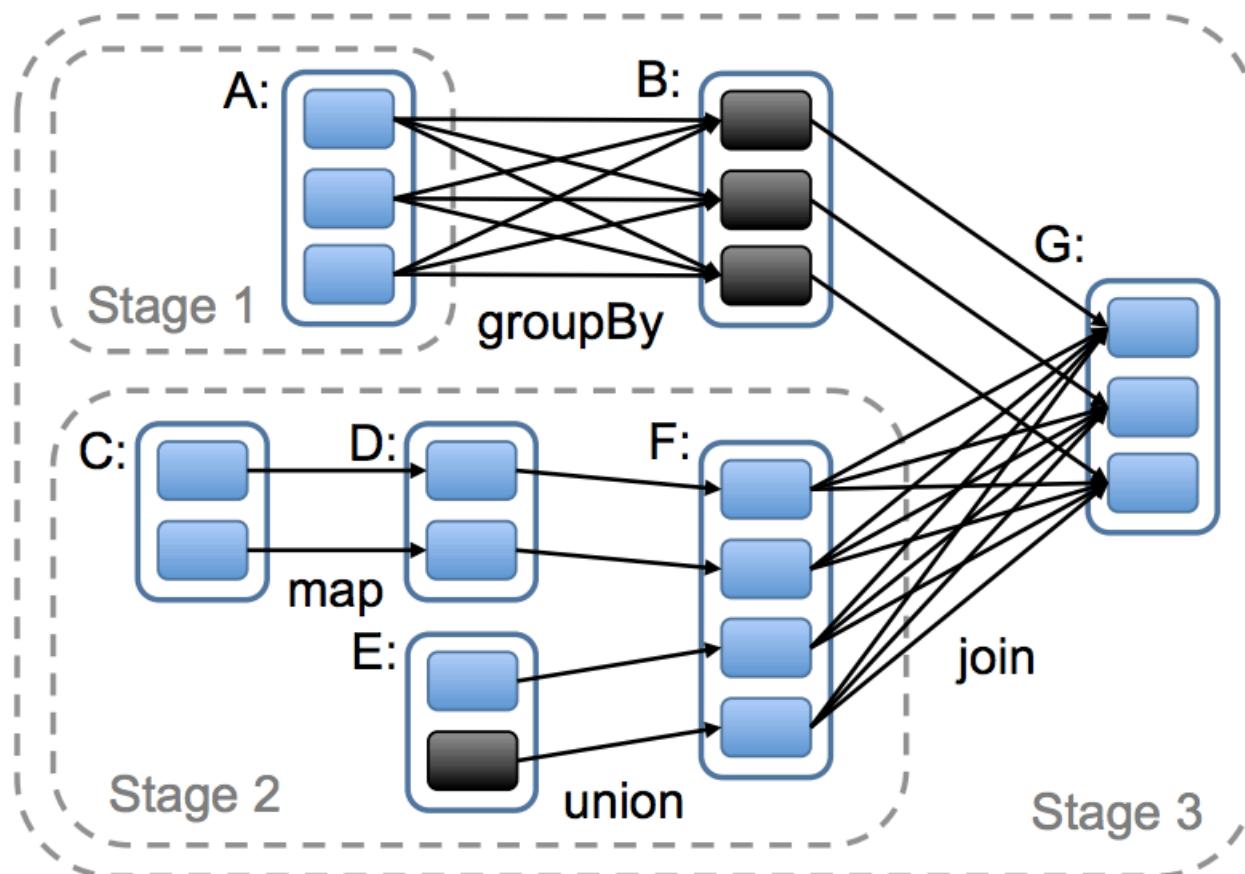


groupByKey



join with inputs not  
co-partitioned

# Execution Plan

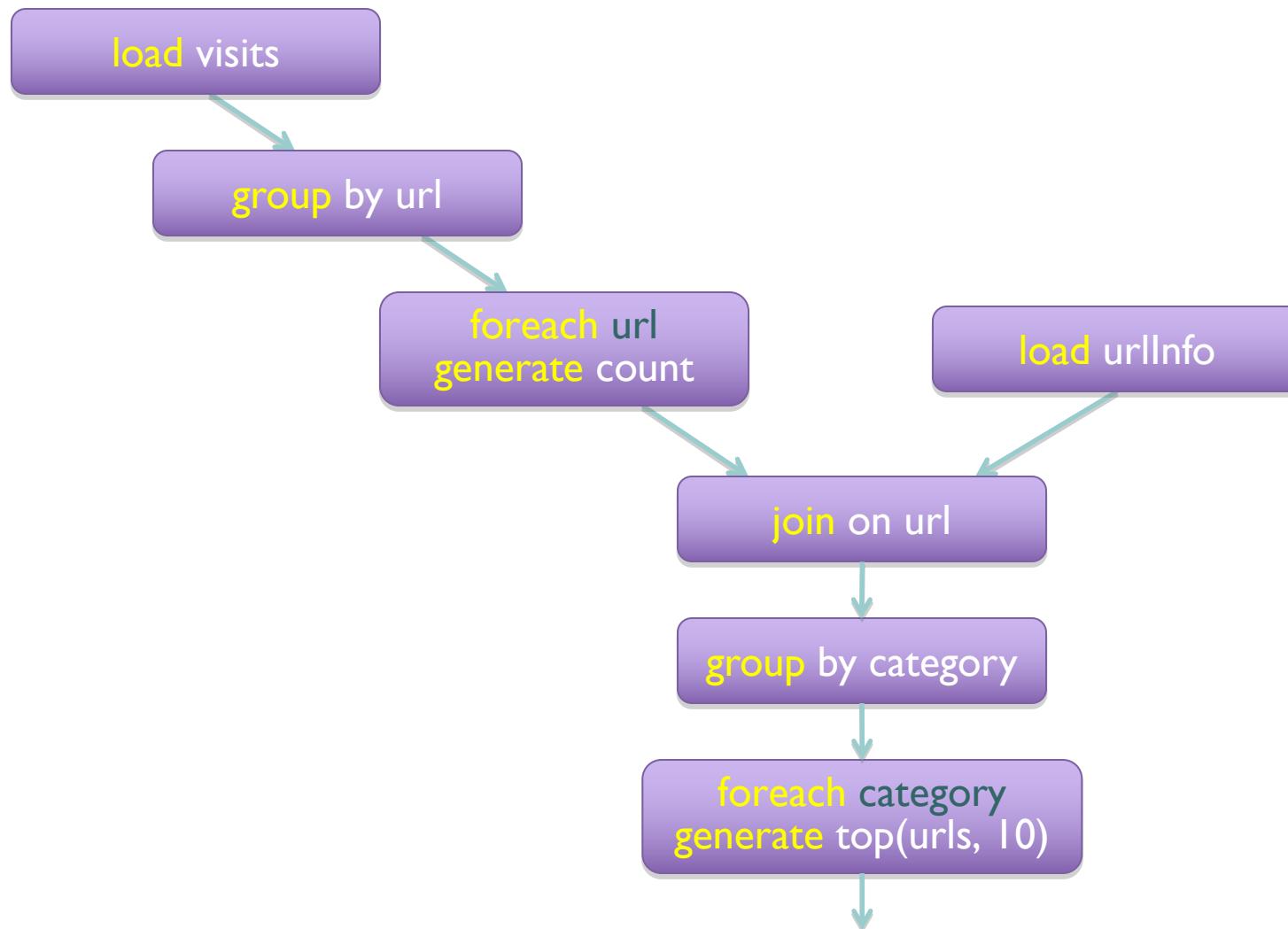


Wait, where have we seen this before?

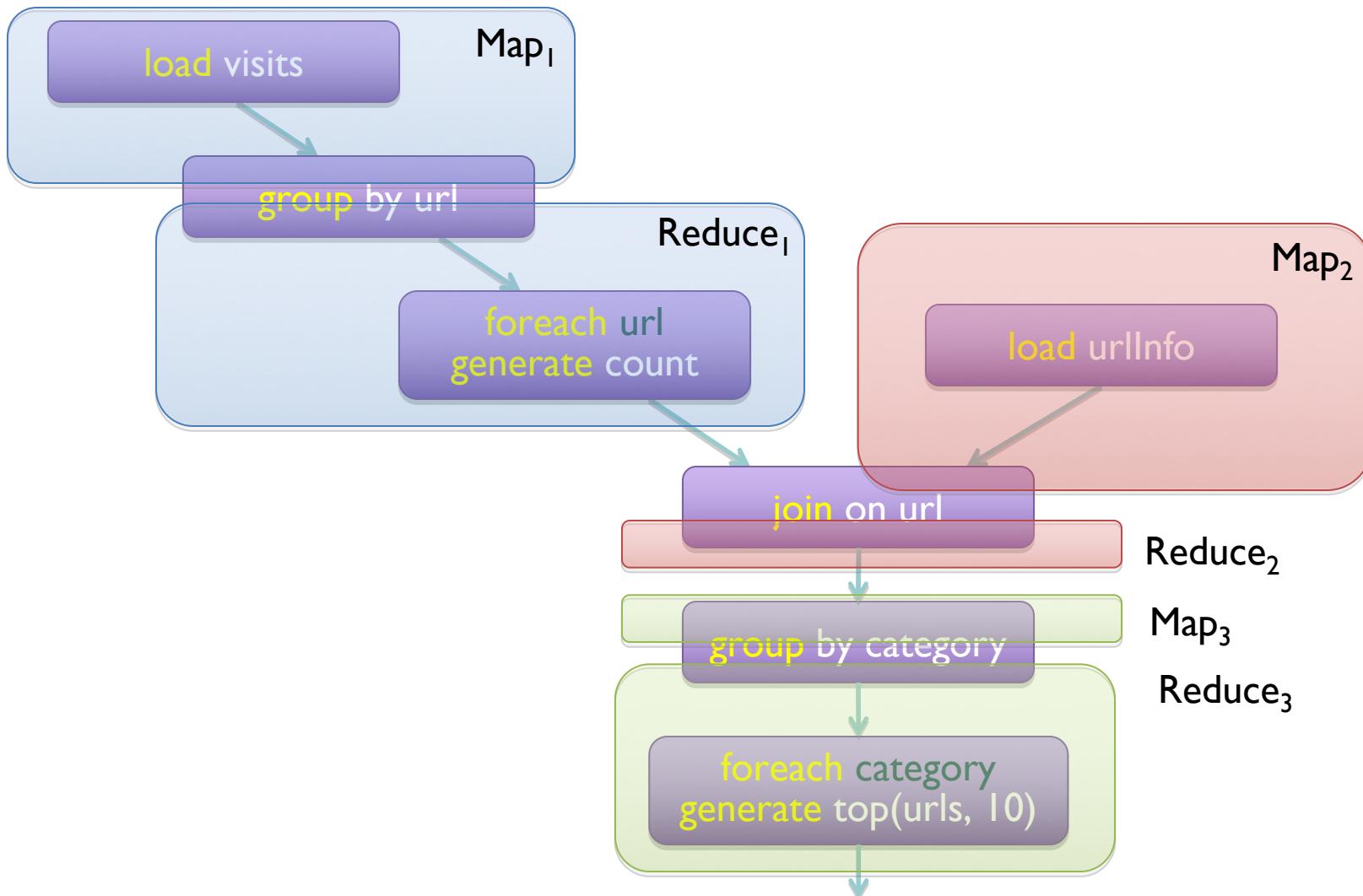
# Pig: Example Script

```
visits = load '/data/visits' as (user, url, time);  
gVisits = group visits by url;  
visitCounts = foreach gVisits generate url, count(visits);  
urlInfo = load '/data/urlInfo' as (url, category, pRank);  
visitCounts = join visitCounts by url, urlInfo by url;  
gCategories = group visitCounts by category;  
topUrls = foreach gCategories generate top(visitCounts,10);  
  
store topUrls into '/data/topUrls';
```

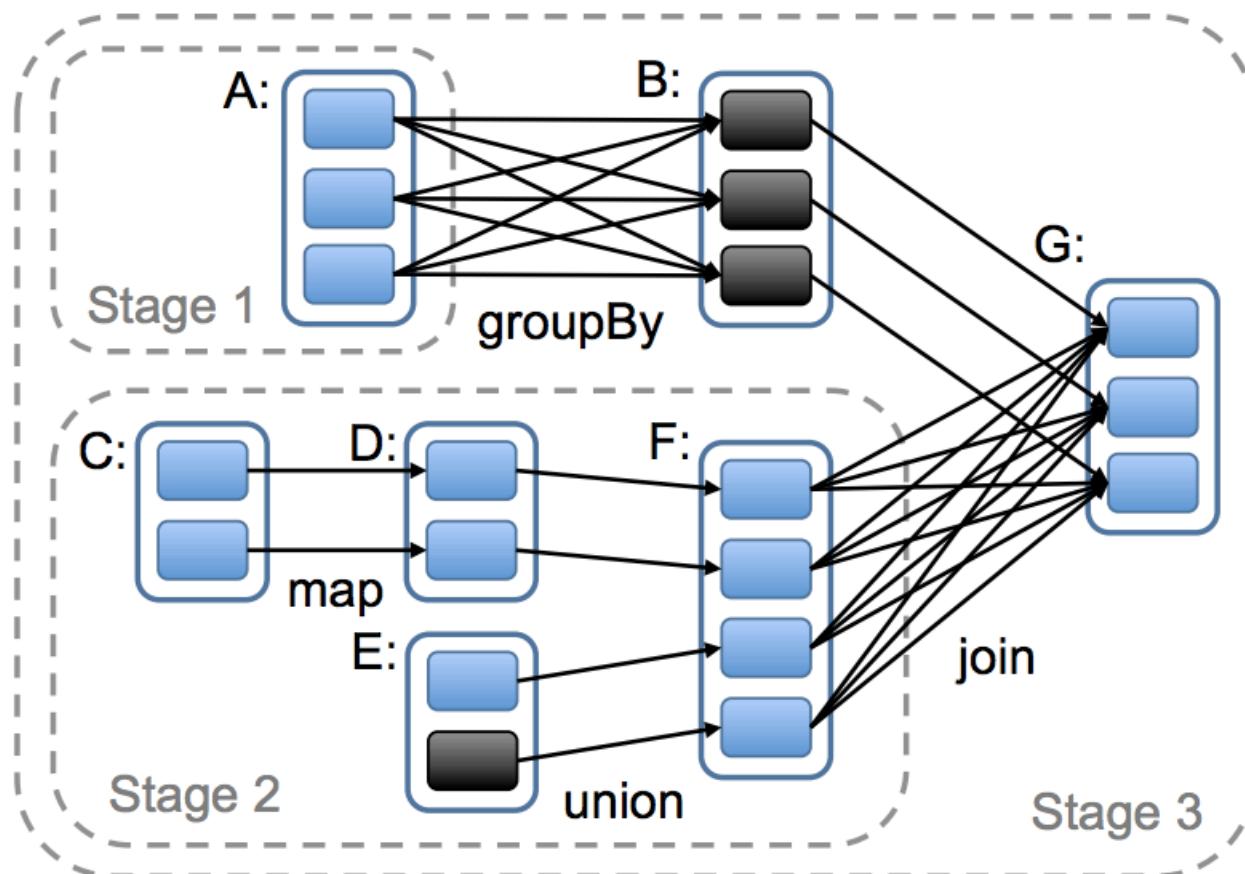
# Pig Query Plan



# Pig: MapReduce Execution

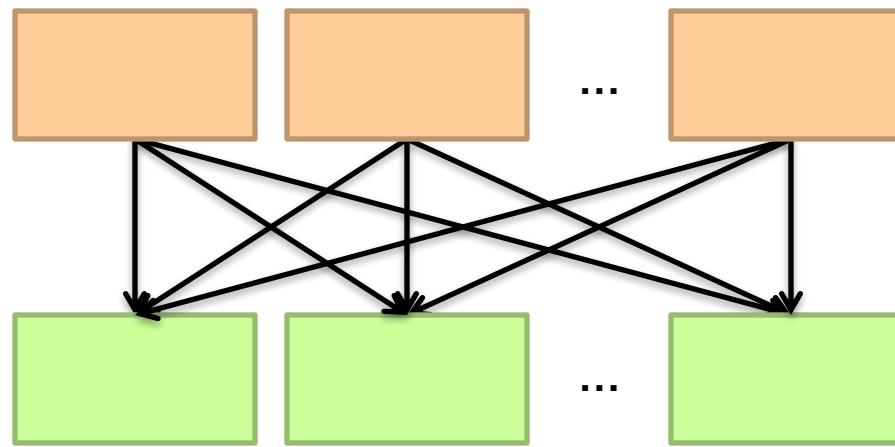


# Execution Plan



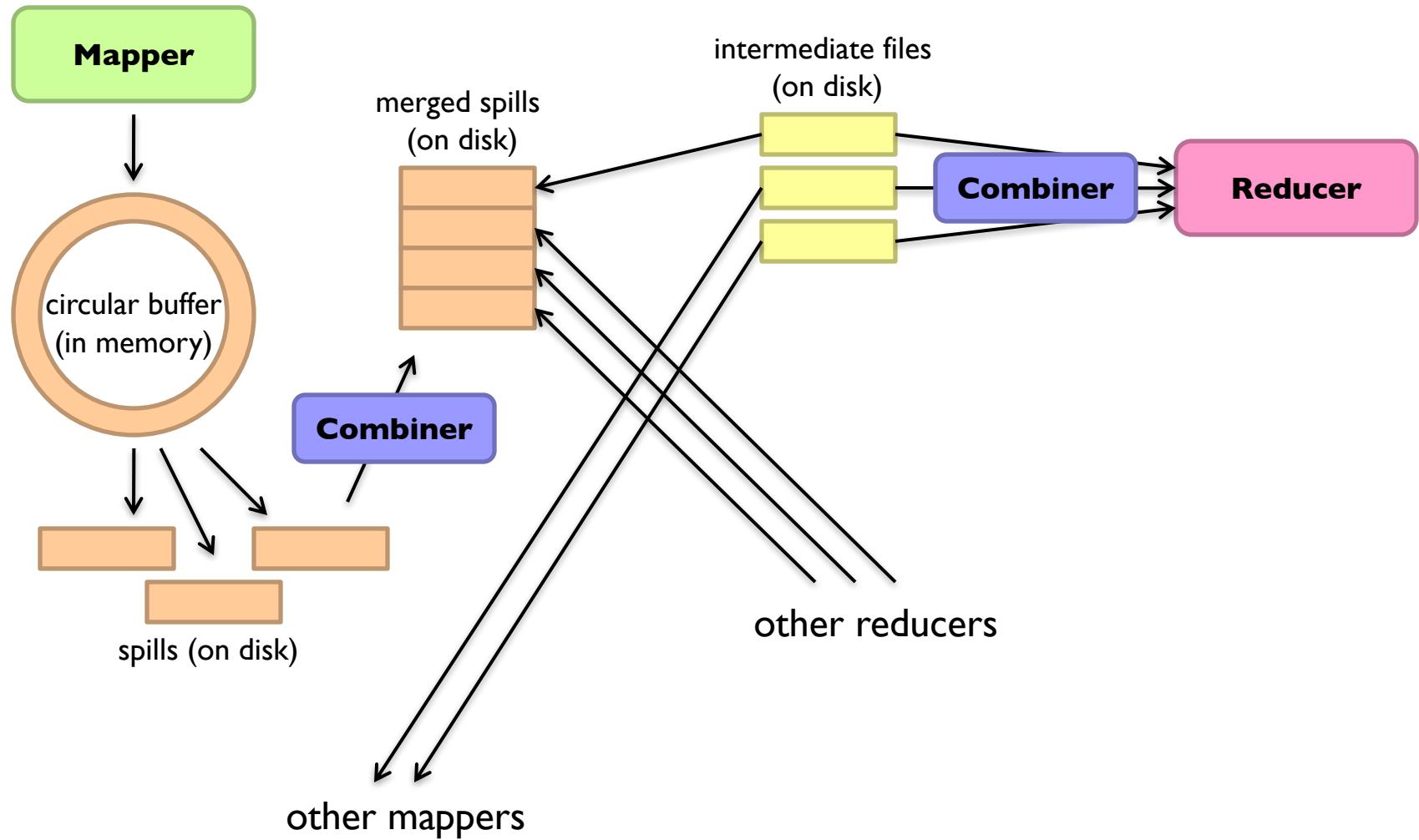
Kinda like a sequence of MapReduce jobs?

# Can't avoid this!



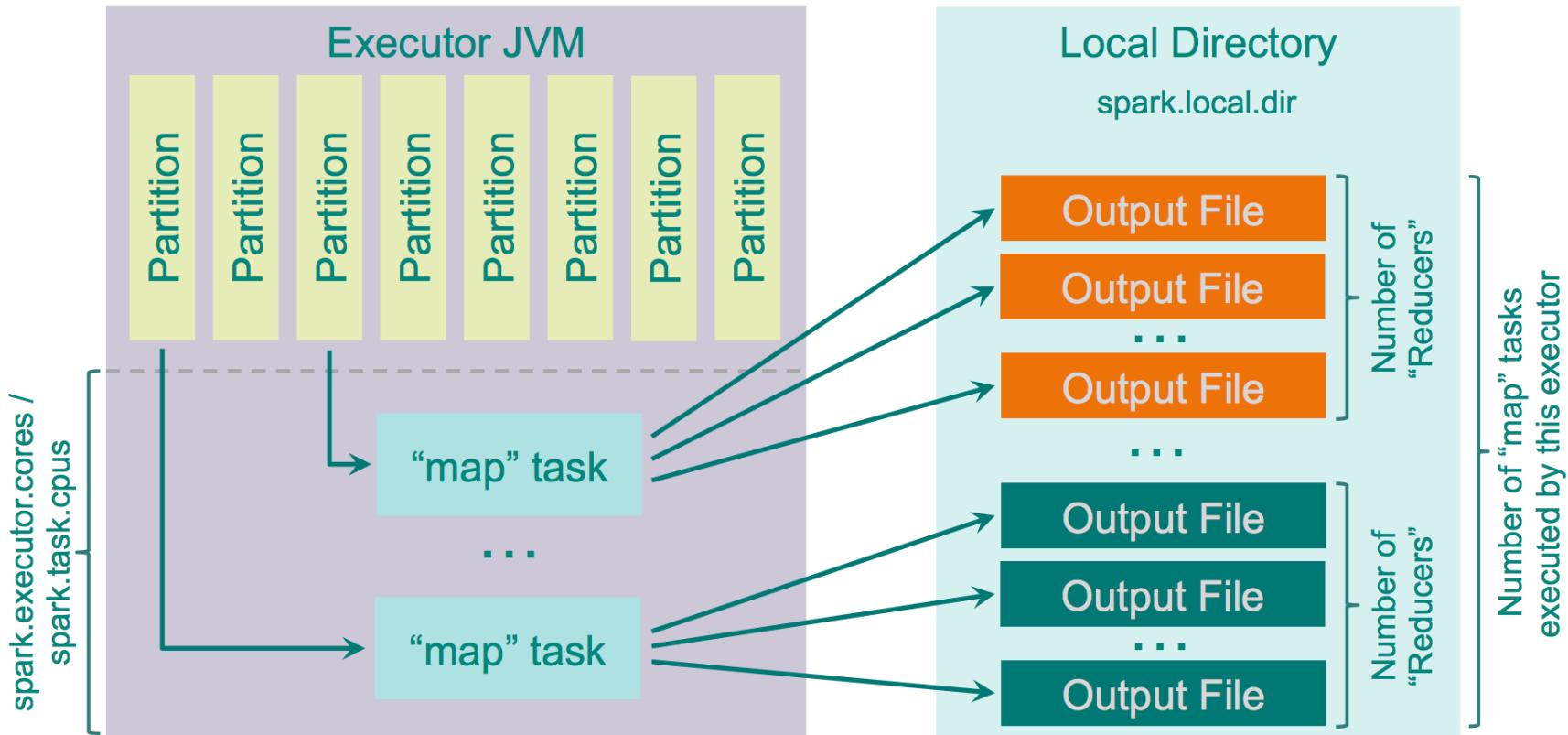
But, what's the major difference?

# Remember this?



# Spark Shuffle Implementations

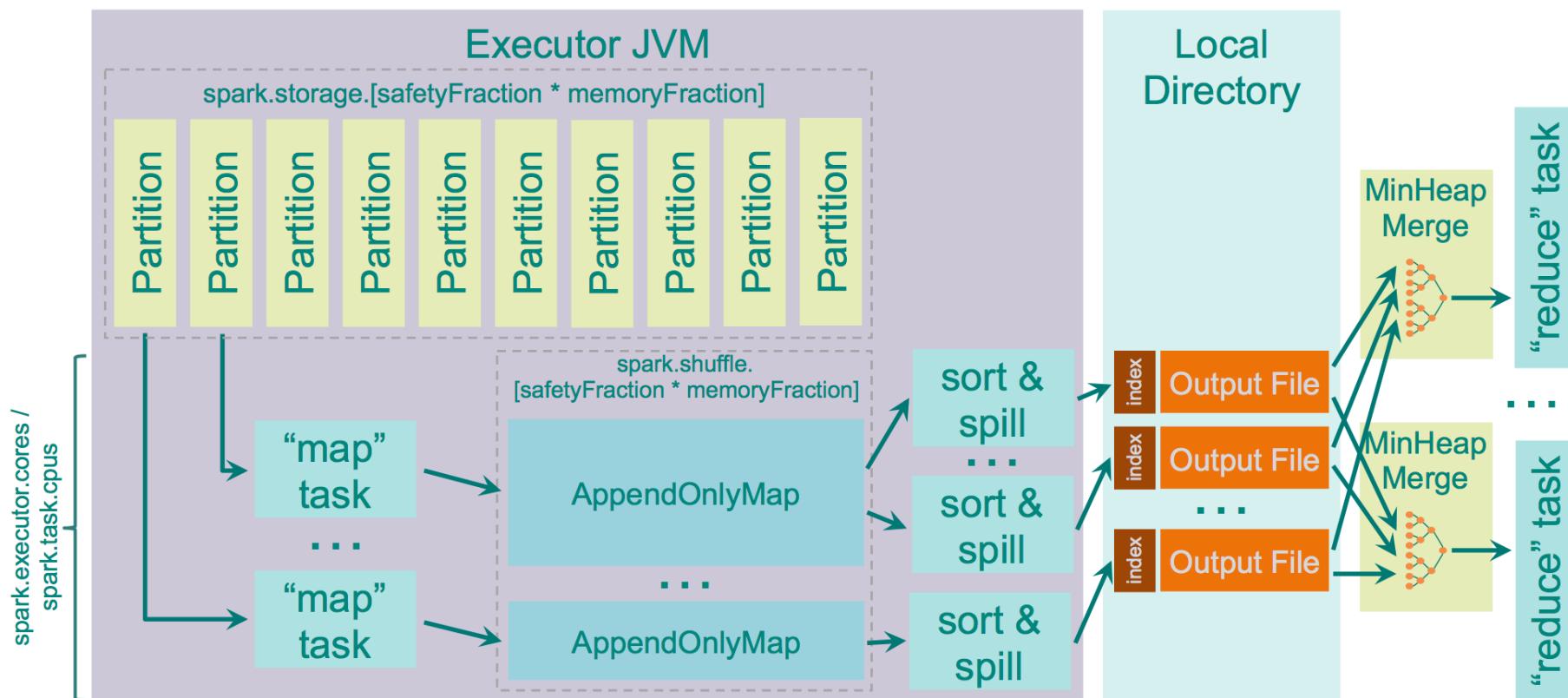
## Hash shuffle



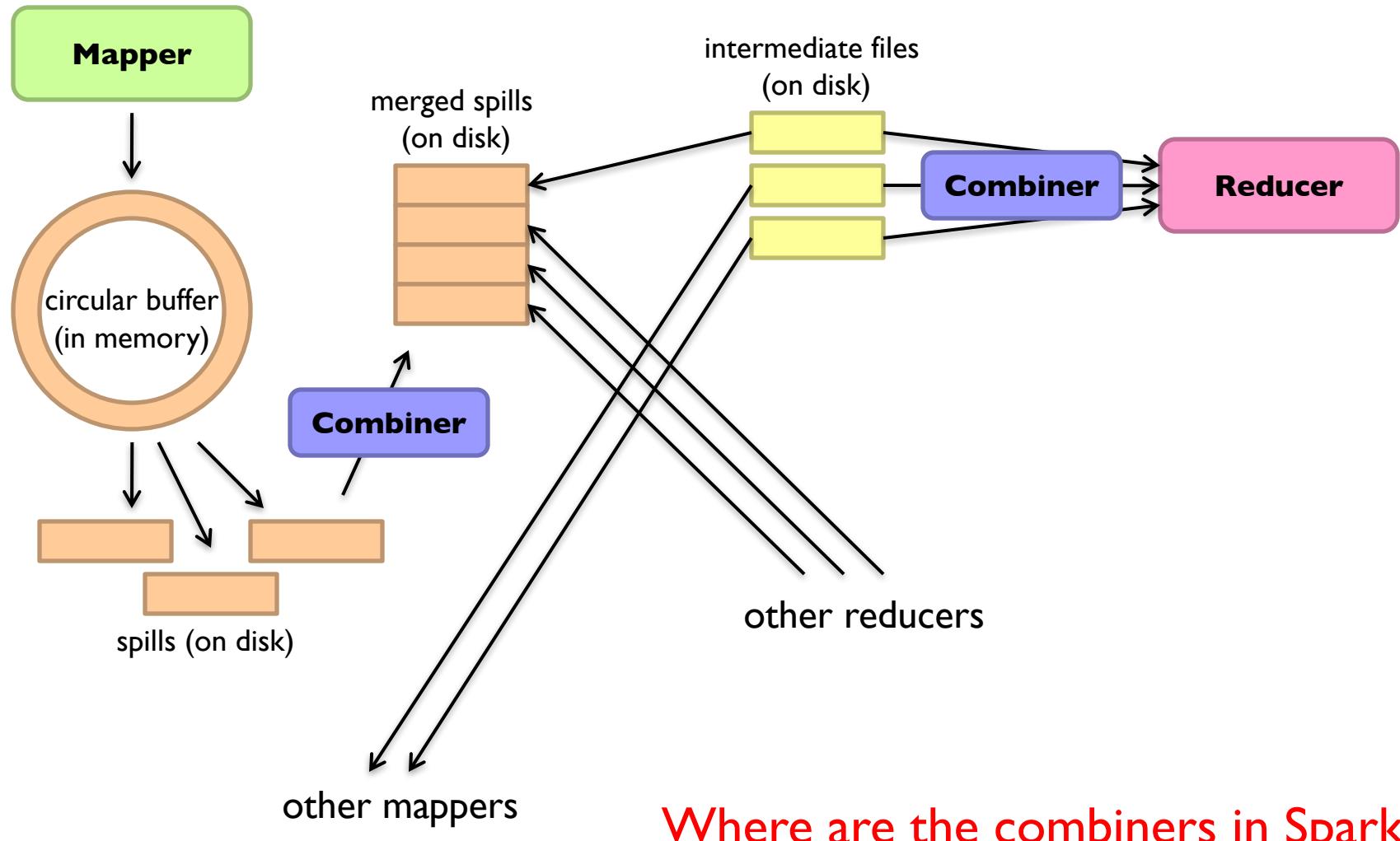
What happened to sorting?

# Spark Shuffle Implementations

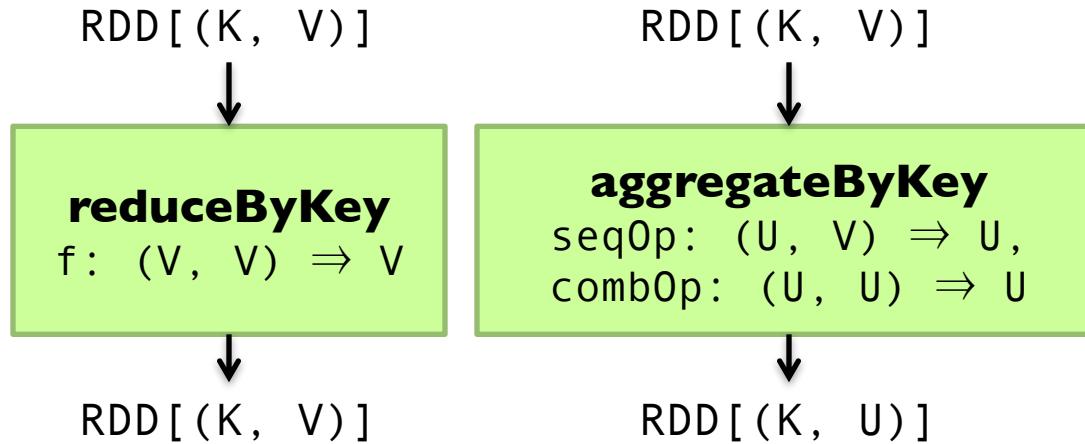
## Sort shuffle



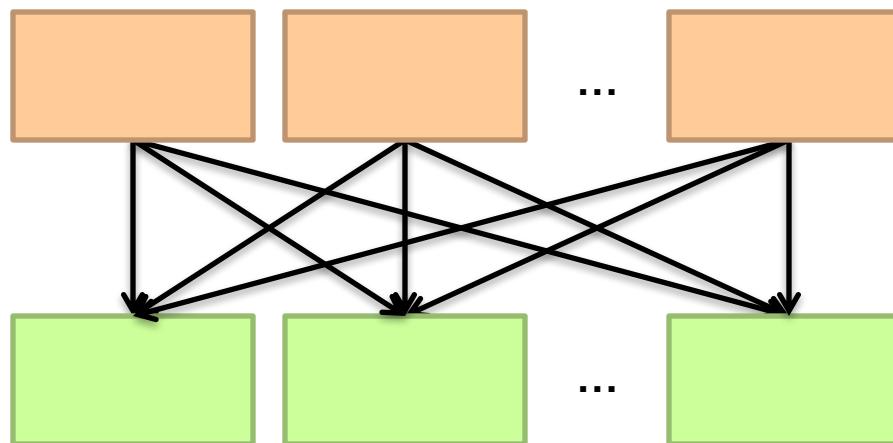
# Remember this?



# Reduce-like Operations



What happened to combiners?



# Spark #wins

Richer operators

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

Scala, Java, Python, R, bindings

# Spark #wins

The screenshot shows a Databricks notebook interface with the following details:

- Title Bar:** "Quick Start Using Scala - Data" and "Jimmy".
- URL:** [https://cdn2.hubspot.net/hubfs/438089/notebooks/Quick\\_Start/Quick\\_Start\\_Using\\_Scala.html](https://cdn2.hubspot.net/hubfs/438089/notebooks/Quick_Start/Quick_Start_Using_Scala.html)
- Header:** "databricks" logo, "Quick Start Using Scala (Scala)", and "Import Notebook" button.
- Code Cell 1:** Shows Scala code to display files in a directory and a table of results.

```
> // Take a look at the file system
display(dbutils.fs.ls("/databricks-datasets/samples/docs/"))

path                                              name          size
dbfs:/databricks-datasets/samples/docs/README.md  README.md    3137
```

A download icon is present below the table.
- Code Cell 2:** Shows Scala code to read a file and its resulting RDD.

```
> // Setup the textFile RDD to read the README.md file
// Note this is lazy
val textFile = sc.textFile("/databricks-datasets/samples/docs/README.md")

textFile: org.apache.spark.rdd.RDD[String] = /databricks-datasets/samples/docs/README.md MapPartitionsRDD[1873222] at textFile at <console>:34
```
- Note:** "RDDs have **actions**, which return values, and **transformations**, which return pointers to new RDDs."
- Code Cell 3:** Shows Scala code to perform an action (count) on the RDD and its resulting value.

```
> // When performing an action (like a count) this is when the textFile is read and aggregate calculated
// Click on [View] to see the stages and executors
textFile.count()

res2: Long = 65
```
- Section Header:** "Scala Count (Jobs)"
- Navigation Bar:** "Jobs" (selected), "Stages", "Storage", "Environment", "Executors", "SQL", "JDBC/ODBC Server".

# Spark #lose

Java serialization (w/ Kryo optimizations)  
Scala: poor support for primitives



# Algorithm design, redux



Two superpowers:

Associativity  
Commutativity  
(sorting)

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

# The Power of Associativity

You can put parentheses where ever you want!

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

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

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

# 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!

# Word Count: Baseline

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

# Fancy Labels for Simple Concepts...

**Semigroup** = (  $M$  ,  $\oplus$  )

$\oplus : M \times M \rightarrow M$ , 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

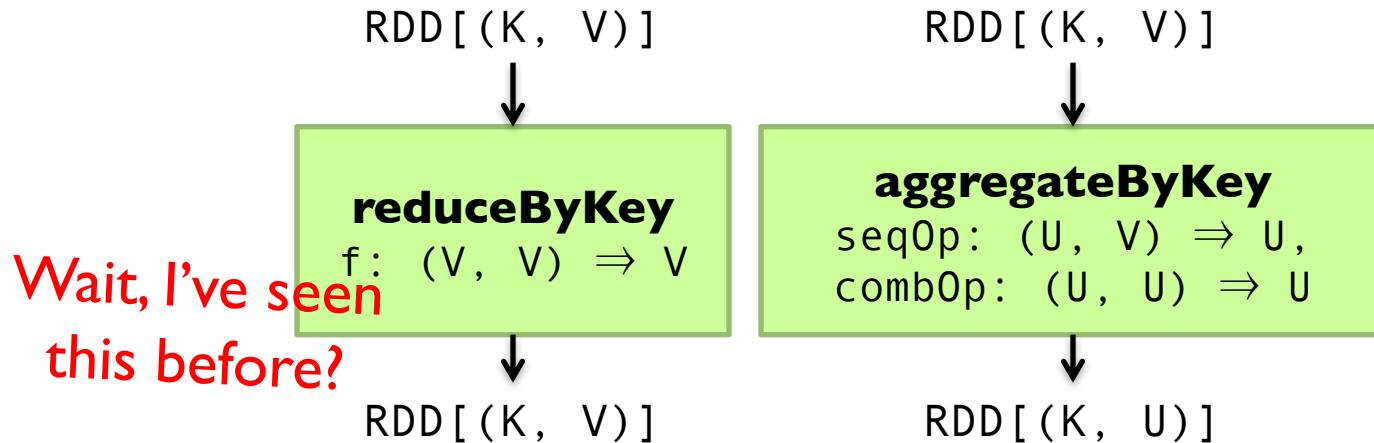
$\varepsilon$  s.t.,  $\varepsilon \oplus m = m \oplus \varepsilon = m$ ,  $\forall m \in M$

**Commutative Monoid** = Monoid + commutativity

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

A few examples?  
(hint, previous slide!)

# Back to these...





Spark / SPARK-12844

Spark documentation should be more precise about the algebraic properties of functions in various transformations

## Description

---

Spark documentation should be more precise about the algebraic properties of functions in various transformations. The way the current documentation is written is potentially confusing. For example, in Spark 1.6, the scaladoc for reduce in RDD says:

> Reduces the elements of this RDD using the specified commutative and associative binary operator.

This is precise and accurate. In the documentation of reduceByKey in PairRDDFunctions, on the other hand, it says:

> Merge the values for each key using an associative reduce function.

To be more precise, this function must also be commutative in order for the computation to be correct. Writing commutative for reduce and not reduceByKey gives the false impression that the function in the latter does not need to be commutative.

# Computing the mean via (again)

```
class Mapper {  
    def map(key: Text, value: Int, context: Context) = {  
        context.write(key, value)  
    }  
}  
  
class Reducer {  
    def reduce(key: Text, values: Iterable[Int], context: Context) {  
        var sum = 0  
        var cnt = 0  
        for (value <- values) {  
            sum += value  
            cnt += 1  
        }  
        context.write(key, sum/cnt)  
    }  
}
```

# Computing the mean v3 (again)

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

Wait, I've seen  
this before?

RDD[(K, V)]

**reduceByKey**

f: (V, V) ⇒ V

RDD[(K, V)]

# Co-occurrence Matrix: Stripes

```
class Mapper {  
    def map(key: Long, value: Text, context: Context) = {  
        for (u <- tokenize(value)) {  
            val map = new HashMap()  
            for (v <- neighbors(u)) {  
                map(v) += 1  
            }  
            context.write(u, map)  
        }  
    }  
}  
  
class Reducer {  
    def reduce(key: Text, values: Iterable[Map], context: Context) = {  
        val map = new HashMap()  
        for (value <- values) {  
            map += value  
        }  
        context.write(key, map)  
    }  
}
```

Wait, I've seen this before!

RDD[(K, V)]

↓

**reduceByKey**

f: (V, V) ⇒ V

↓

RDD[(K, V)]

# Computing the mean v2 (again)

```
class Mapper {  
    def map(key: Text, value: Int, context: Context) =  
        context.write(key, value)  
}  
class Combiner {  
    def reduce(key: Text, values: Iterable[Int], context: Context) = {  
        for (value <- values) {  
            sum += value  
            cnt += 1  
        }  
        context.write(key, (sum, cnt))  
    }  
}  
class Reducer {  
    def reduce(key: Text, values: Iterable[Pair1], context: Context) = {  
        for (value <- values) {  
            sum += value.left  
            cnt += value.right  
        }  
        context.write(key, sum/cnt)  
    }  
}
```

The diagram illustrates the transformation process. It starts with an input RDD labeled  $\text{RDD}[(\text{K}, \text{V})]$ . An arrow points down to a green box containing the method **aggregateByKey**, which includes the definitions for **seqOp**:  $(\text{U}, \text{V}) \Rightarrow \text{U}$  and **combOp**:  $(\text{U}, \text{U}) \Rightarrow \text{U}$ . A second arrow points down from the green box to an output RDD labeled  $\text{RDD}[(\text{K}, \text{U})]$ .

# 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

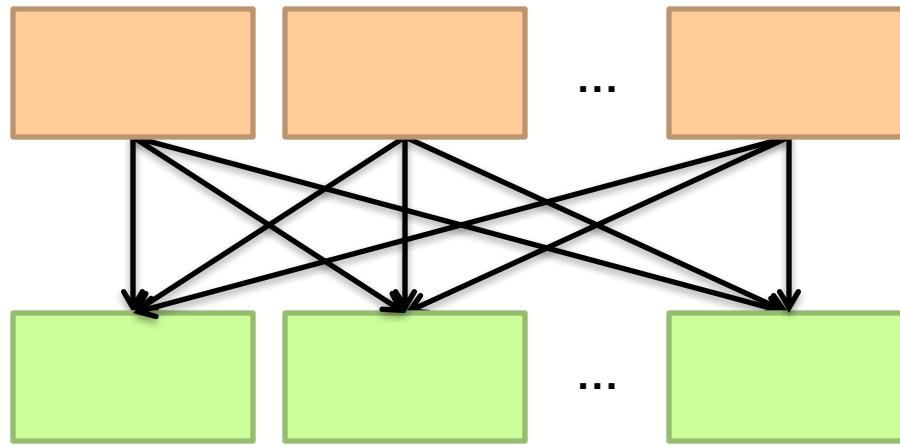
Approach 2: data structures that bring partial results together

Each reducer receives all the data it needs to complete the computation

Illustrated by the “stripes” approach

Commutative monoids!

# Because you can't avoid this...



But commutative monoids help

# 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

What about this?

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

Commutative monoids!

# $f(B|A)$ : “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)

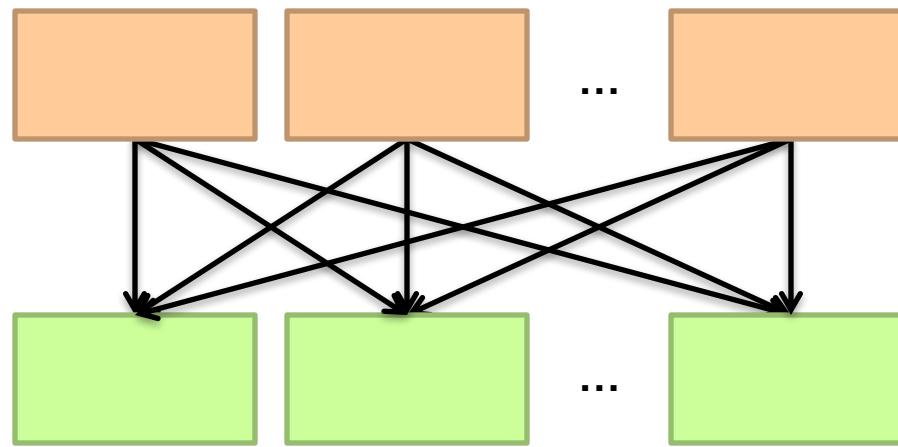
Make sure  $(a, *)$  comes first (define sort order)

Hold state in reducer across different key-value pairs



Two superpowers:  
Associativity  
Commutativity  
(sorting)

# When you can't “monoidify”



Sequence your computations by sorting

# An Apt Quote

All problems in computer science can be solved  
by another level of indirection... Except for the  
problem of too many layers of indirection.

- David Wheeler

An aerial photograph of a large data center complex during sunset. The sky is a warm orange and yellow. In the foreground, there are several large white industrial buildings, some with flat roofs and others with green roofs. There are also numerous white shipping containers stacked in a row. A network of roads and parking lots connects the buildings. In the background, there are more industrial structures, a highway with traffic, and a vast, green, agricultural landscape stretching to the horizon.

The datacenter *is* the computer!  
What's the instruction set?  
What are the abstractions?

# 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)

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?