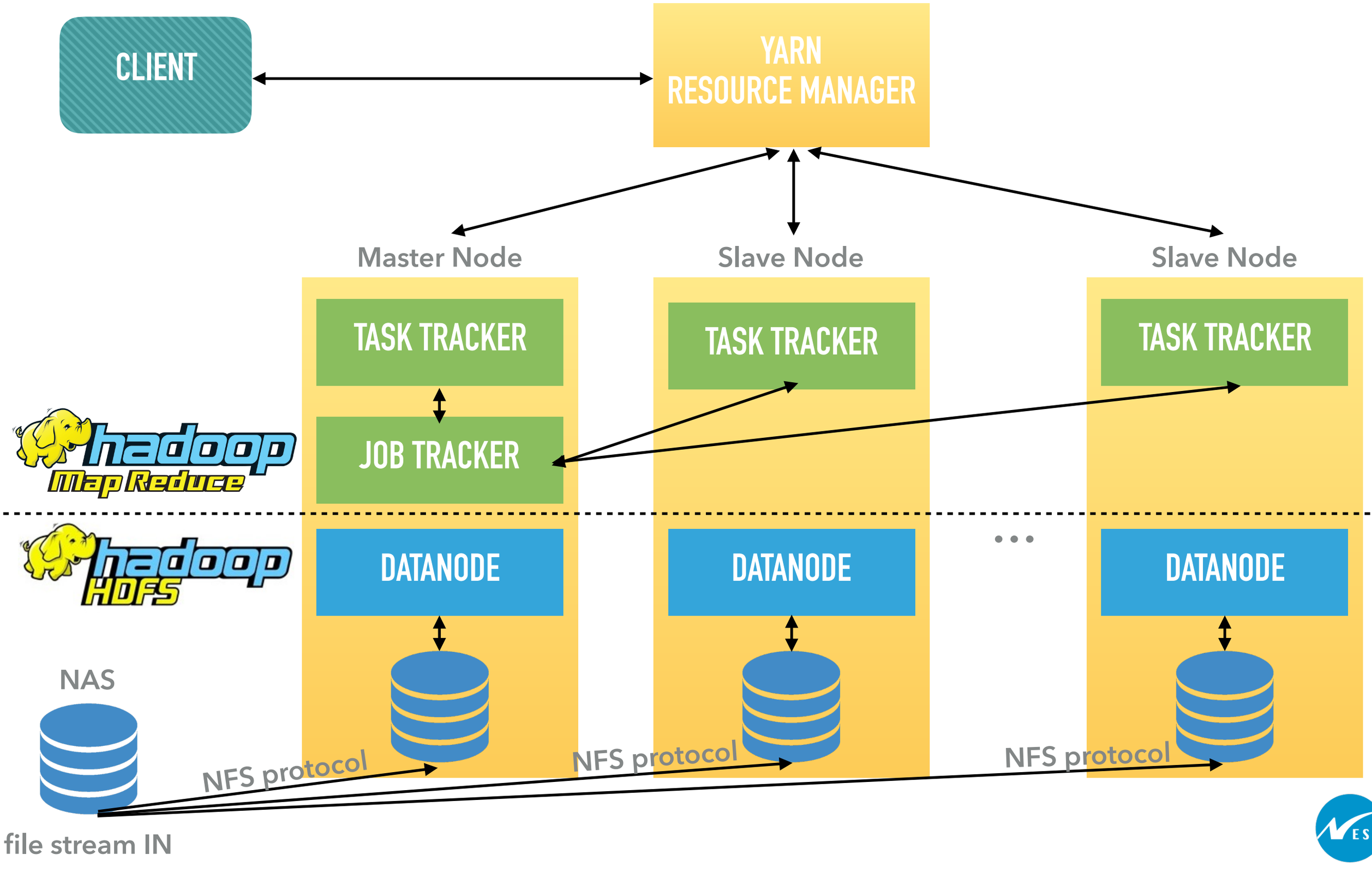
BIG DATA PROCESSING WITH

APACHE SPARK

Cristian-Viktor **Ardelean**
Software Engineer @ Ness
viktor.ardelean@ness.com

CASE STUDY

- data streaming in at 200 MB/s => ~17 TB per day
- custom aggregations, filtering and joins with external data
- 4 different data workflows
- each workflow result persisted for further analytics
- result data partitioned per date
- near real-time results


CLIENT
YARN
RESOURCE MANAGER
Master Node
Slave Node
Slave Node
TASK TRACKER
TASK TRACKER
TASK TRACKER
JOB TRACKER
hadoop
Map Reduce
hadoop
HDFS
DATANODE
DATANODE
DATANODE
NAS
NFS protocol
NFS protocol
NFS protocol
file stream IN

WE CAN'T PROCESS DATA IN NEAR REAL-TIME WITH MAP REDUCE

MAP REDUCE

SPARK

WORD COUNT EXAMPLE

MAP REDUCE

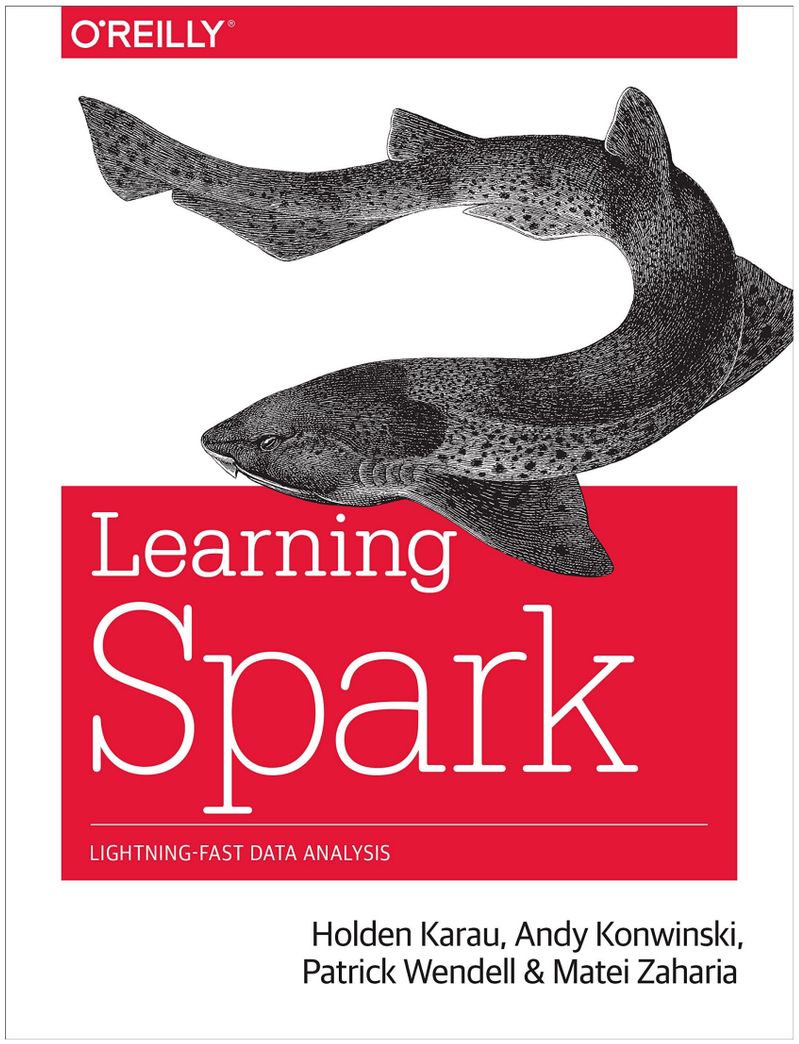
```
package org.myorg;
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class WordCount {
   public static class Map extends MapReduceBase
implements Mapper<LongWritable, Text, Text, IntWritable> {
     private final static IntWritable one = new IntWritable(1);
     private Text word = new Text();

     public void map(LongWritable key, Text value, OutputCollector<Text,
IntWritable> output, Reporter reporter) throws IOException {
       String line = value.toString();
       StringTokenizer tokenizer = new StringTokenizer(line);
       while (tokenizer.hasMoreTokens()) {
         word.set(tokenizer.nextToken());
         output.collect(word, one);
       }
     }
   }

   public static class Reduce extends MapReduceBase
implements Reducer<Text, IntWritable, Text, IntWritable> {
     public void reduce(Text key, Iterator values, OutputCollector<Text,
IntWritable> output, Reporter reporter) throws IOException {
       int sum = 0;
       while (values.hasNext()) {
         sum += values.next().get();
       }
       output.collect(key, new IntWritable(sum));
     }
   }

   public static void main(String[] args) throws Exception {
     JobConf conf = new JobConf(WordCount.class);
     conf.setJobName("wordcount");
     conf.setOutputKeyClass(Text.class);
     conf.setOutputValueClass(IntWritable.class);
     conf.setMapperClass(Map.class);
     conf.setCombinerClass(Reduce.class);
     conf.setReducerClass(Reduce.class);
     conf.setInputFormat(TextInputFormat.class);
     conf.setOutputFormat(TextOutputFormat.class);
     FileInputFormat.setInputPaths(conf, new Path(args[0]));
     FileOutputFormat.setOutputPath(conf, new Path(args[1]));
     JobClient.runJob(conf);
   }
}
```

SPARK

```
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

SPARK

INTRO


O'REILLY
Learning
Spark
LIGHTNING-FAST DATA ANALYSIS
Holden Karau, Andy Konwinski,
Patrick Wendell & Matei Zaharia

DATASTAX

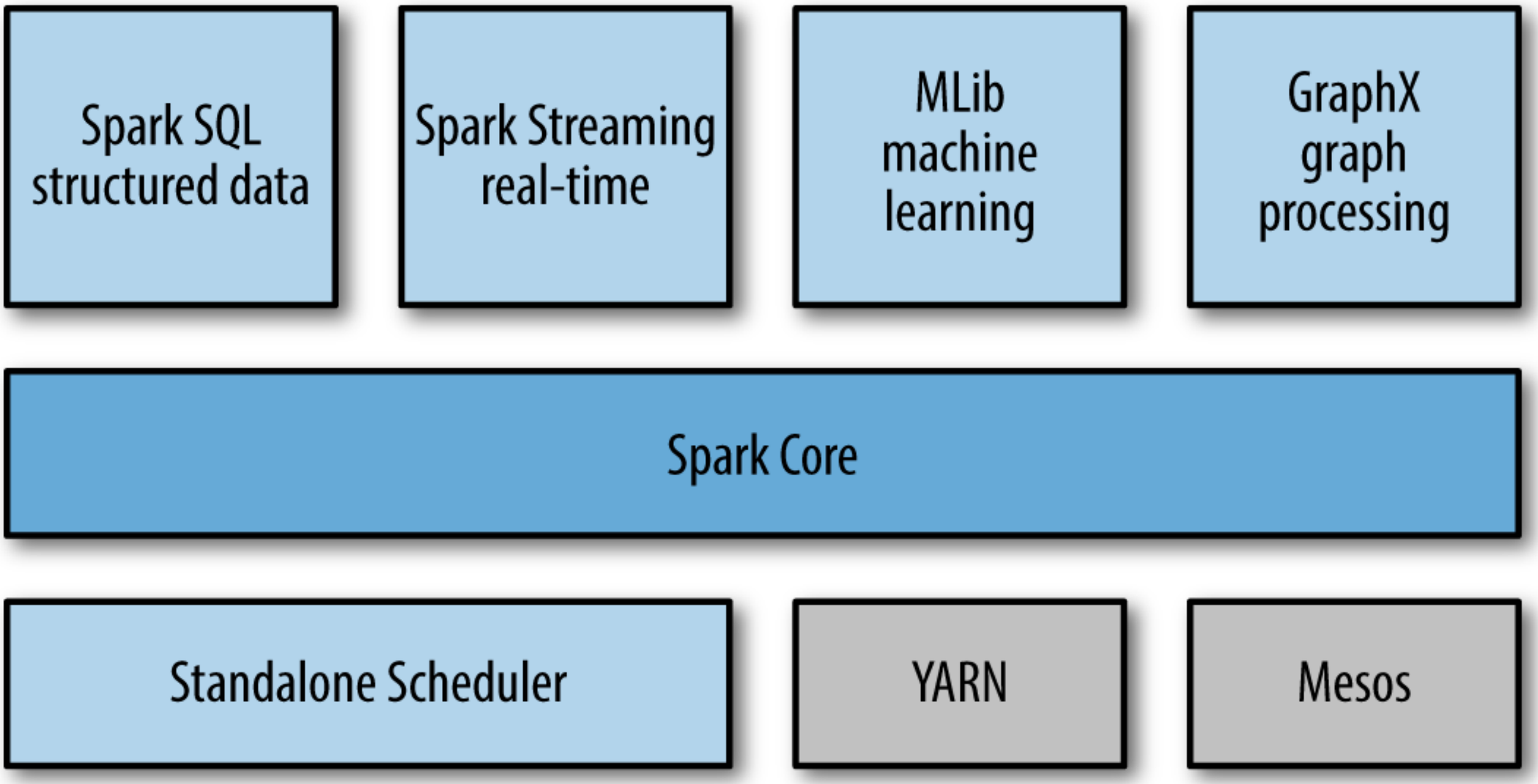
Apache Spark is a cluster computing platform designed to be **fast** and **general purpose**.

FAST

- 10 x faster than MapReduce
- runs computations in memory

GENERAL PURPOSE

- Batch applications - **Spark Core**
- Iterative algorithms - **Spark MLib**
- Interactive queries - **Spark SQL**
- Streaming - **Spark Streaming**
- Graph processing - **GraphX**



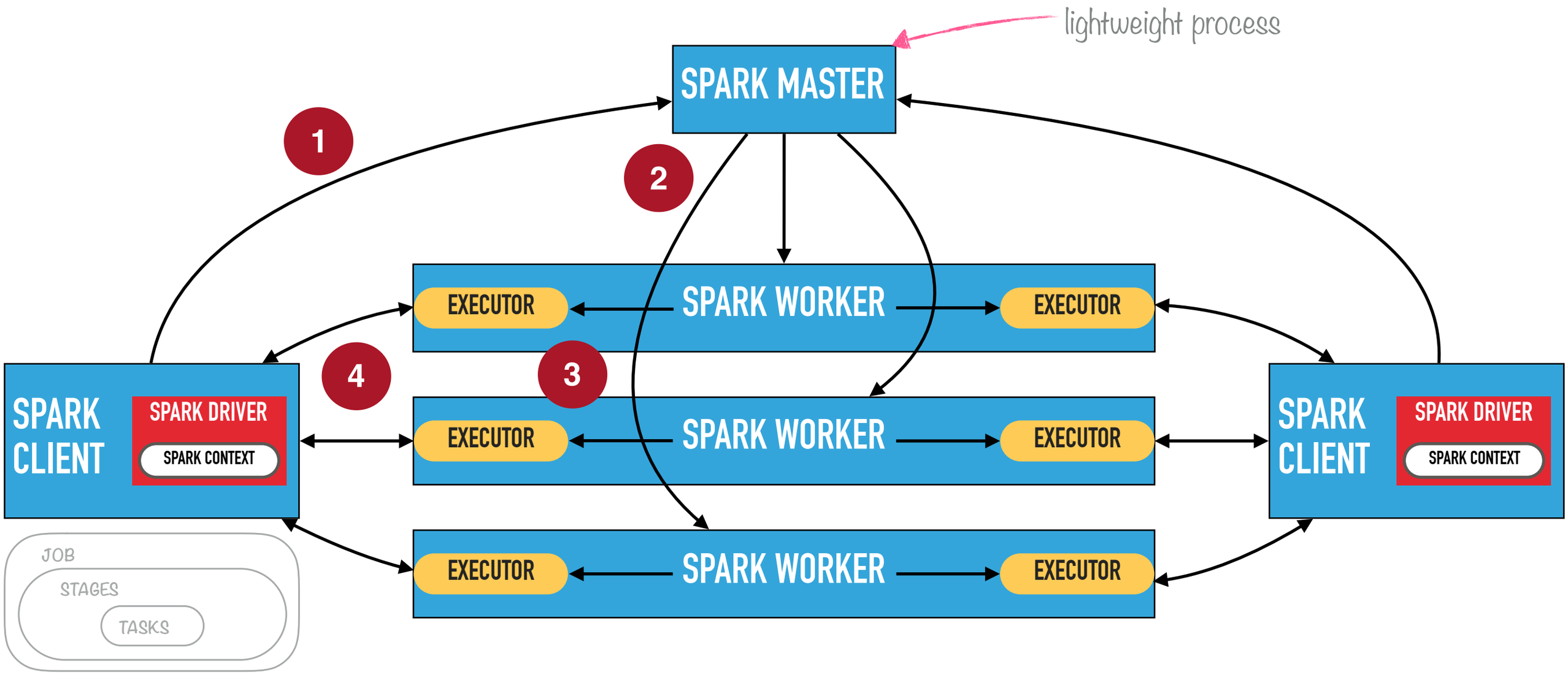
Source: https://www.safaribooksonline.com/library/view/learning-spark/9781449359034/assets/lnsp_0101.png

- Task scheduling
- Memory management
- Fault recovery
- Storage systems interaction

- Spark API that also defines RDDs

RDD **Resilient Distributed Dataset** is Sparks main programming abstraction and represents a distributed collection across cluster nodes on which transformations and actions can be performed in parallel.

SPARK ARCHITECTURE



1 Request for computational resources

```
conf = SparkConf()\
    .setAppName(appName).setMaster(master)\
    .set("spark.executor.memory", "12g")\
    .set("spark.executor.cores", 3)

sparkContext = SparkContext(conf=conf)
```

2 Allocate computational resources

- 1 JVM / Executor
- by default 1 Executor / Worker
- from Spark 1.3.1

spark.dynamicAllocation.enabled (yarn)
in order to dynamically allocate executors

3 Start Executors

4 Schedule and perform computation

Source: https://academy.datastax.com/courses/ds320-analytics-apache-spark/introduction-spark-architecture

- interactive interpreter for Scala, Python and R with Spark API
- predefined objects like:
  - sc : SparkContext
  - sqlContext : SQLContext

Spark Shell Options

Option	Description	Default value
–master MASTER_URL	Specify a Master URL as spark://host:port or local[N], where N is a desired number of cores	local
–name NAME	Spark Shell application name	Spark shell
–driver-memory MEM	Amount of memory for Driver	512M
–executor-memory MEM	Amount of memory per executor	1G
–total-executor-cores NUM	Total number of cores for all Executors.	All available

START SPARK SHELL

Connecting to a remote cluster

```
./spark-shell –master spark://10.89.0.224:7077 \
              –name “Spark App name” \
              –executor-memory 2G \
              –total-executor-cores 4
```

Running locally

```
./spark-shell –master local[2] \
              ––name “Spark App name”
```

SPARK WEB UI

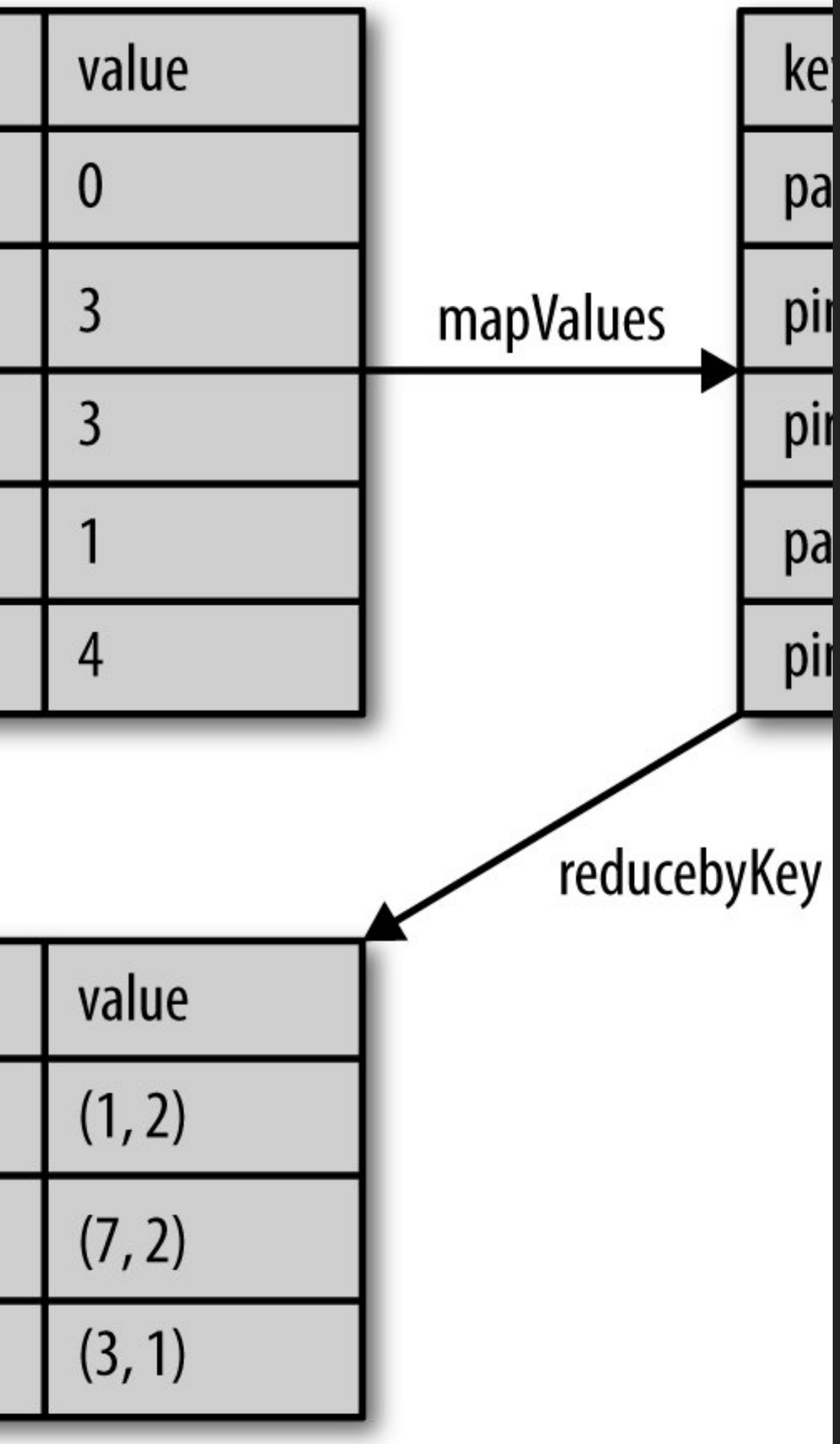
Master Web UI: http://<master-host>:8080

- displays information about Master, Workers and Applications

Application Web UI: http://<driver-host>:4040

- displays information about running and completed jobs, stages, tasks, as well as information about persisted datasets, environment settings, and Executors that are controlled by an application. If more than one application is running, the Web UI ports are assigned as 4040, 4041, and so forth.

- is exposed by the Spark Driver



RDDS – RESILIENT DISTRIBUTED DATASETS

- An RDD is an **immutable** collection of data that is divided into **partitions** and **distributed** across all Spark worker nodes.
- RDDs can be created by loading an external dataset, or by distributing a collection of objects in the driver program.
- We can perform 2 types of operations on RDDs: **transformations** and **actions**.
- **Transformations** consist of executing some operations on a RDD and construct another RDD from the original one. Some common transformations are map, flatmap and filter.
- **Actions** are called on a RDD and compute a result based on it, that is sent to the driver program or saved to an external storage system. Common actions are: first(), collect(), count(), sum().

- Spark computes RDDs in a **lazy** fashion.

PYTHON

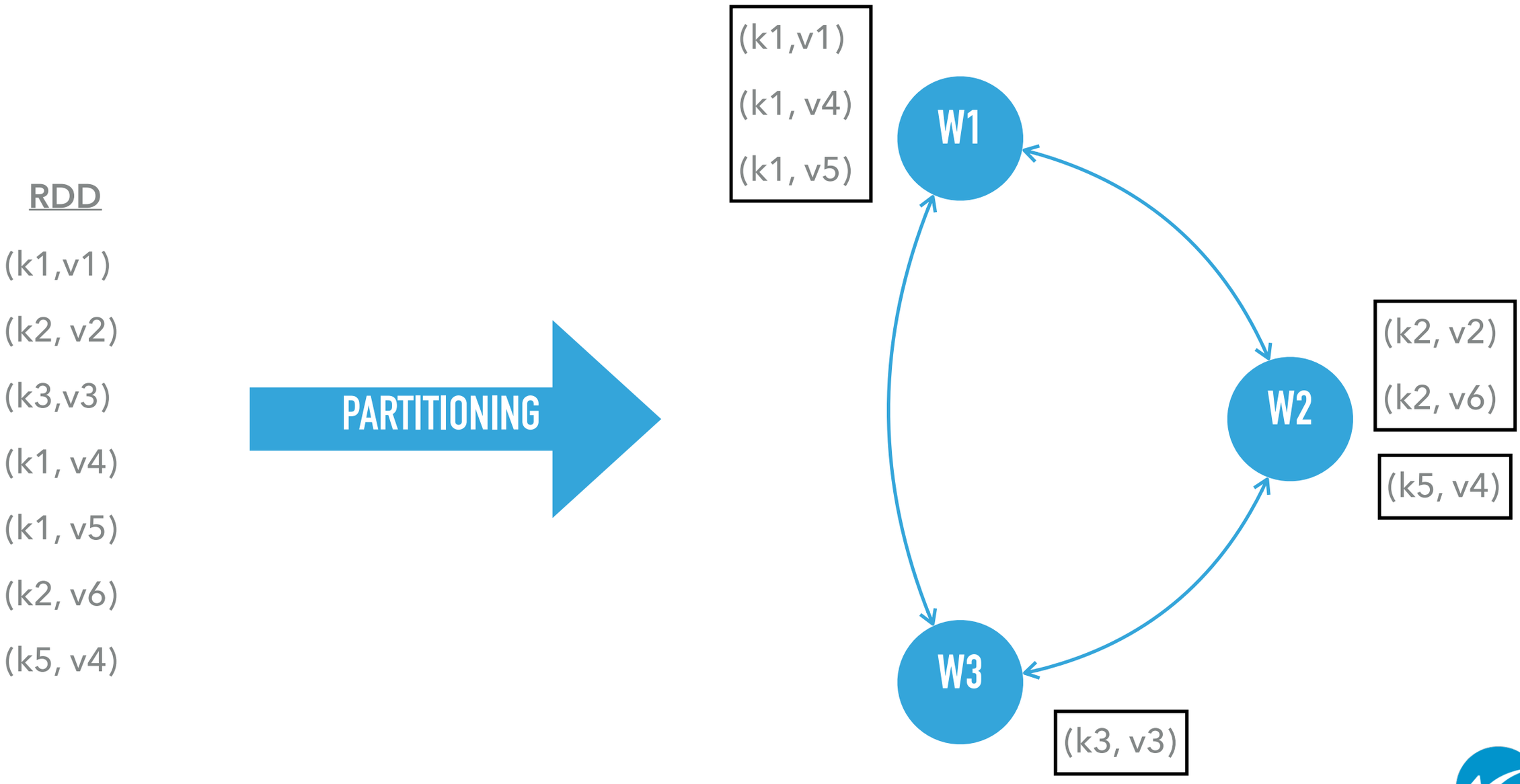
```
lines = sc.textFile("file:///Users/p3700698/100meters.txt")

usainLines = lines.filter(lambda line: "Usain" in line)

usainLines.first()
```

SCALA

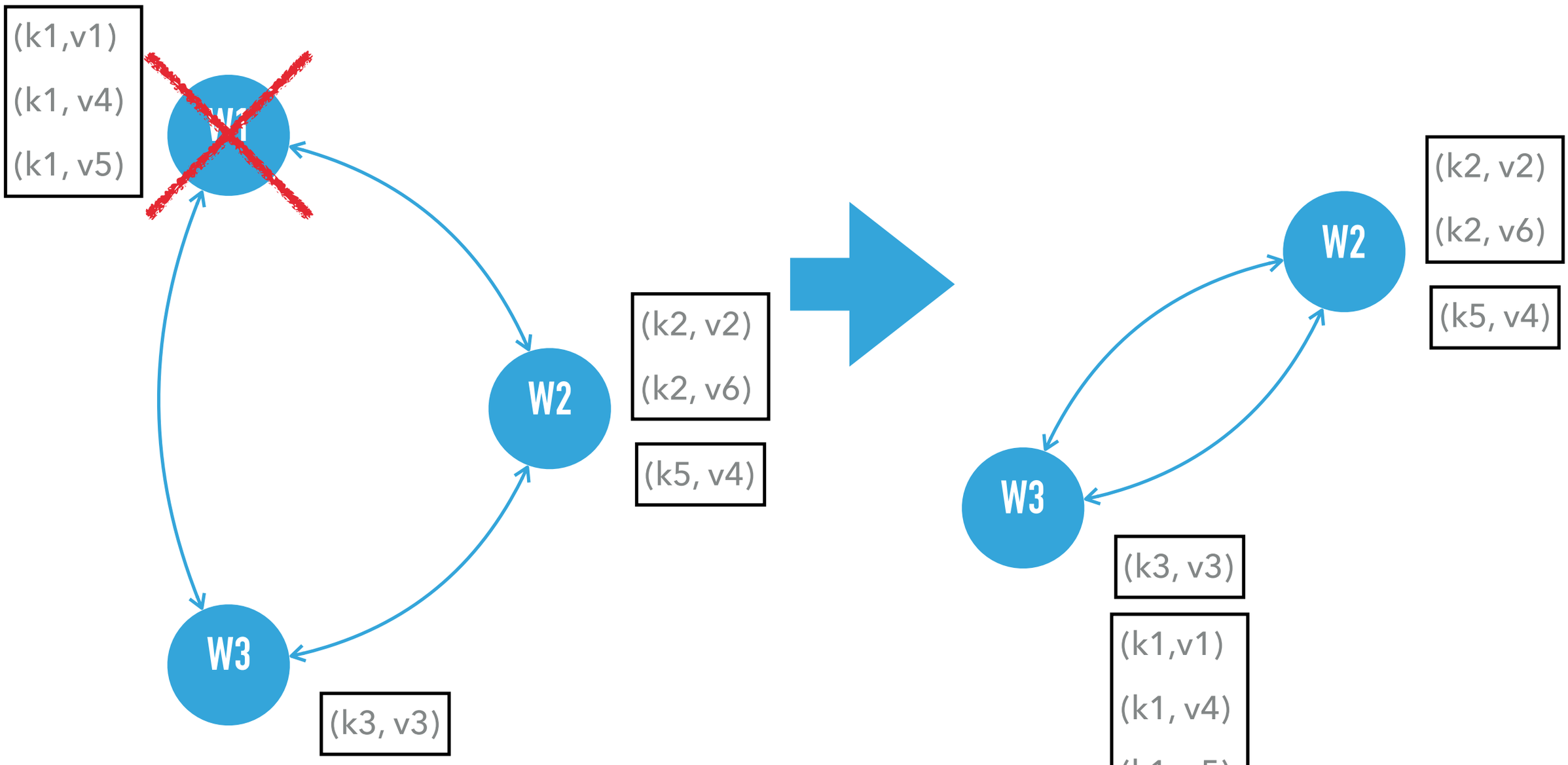
```
var lines = sc.textFile("file:///Users/p3700698/100meters.txt")

var usainLines = lines.filter(line => line contains "Usain")

usainLines.first()
```

- Spark RDDs are each time recomputed when we run an action on them. If a RDD is reused more than once, it is usually recommended to cache it, in order to avoid recomputing.
- We can opt to cache in memory or on disk.

Spark automatically distributes a RDD across worker nodes.



Spark remembers the lineage of the data and therefore achieves fault-tolerance.



- Creating RDDs:

Loading from collection:

```
#Python
lines = sc.parallelize(["you", "shall", "not pass"])

//Scala
val lines = sc.parallelize(List("you", "shall", "not pass"))
```

Loading external datasource:

```
#Python
lines = sc.textFile("file:///Users/p3700698/100meters.txt")

//Scala
val lines = sc.textFile("file:///Users/p3700698/100meters.txt")
```

- Creating RDDs:

From another rdd:

```
//Scala
val rdd    = sc.parallelize(List("you", "shall", "not pass"))
val newRDD = rdd.filter(word => word.equals("you"))

#Python
rdd    = sc.parallelize(["you", "shall", "not pass"])
newRDD = rdd.filter(lambda word: word == "you")
```

- A transformation takes a RDD as an input and creates another RDD as its output. No actual computation takes place.

Unary transformations

filter(f) - creates a RDD that contains only values from the original RDD on which the condition of the f function is *true*.

map(f) - creates a RDD which has each element modified by the f function. There is a *one-to-one* correlation between the elements of the source and resulting RDD.

flatmap(f) - creates a RDD which has each element modified by the f function. There is a *one-to-many* correlation between the elements of the source and resulting RDD. So the function transforms each element into a Seq and merges them together at the end.

distinct() - creates a RDD where duplicated elements are removed.

sample() - creates a RDD that contains a fraction of data from the original RDD.

- A transformation takes a RDD as an input and creates another RDD as its output. No actual computation takes place.

Binary transformations

union(secondRDD) - creates a RDD that contains all the elements from both RDDs. Duplicates are allowed.

intersection(secondRDD) - creates a RDD that contains all the common elements of the two RDDs with no duplicates.

substract(secondRDD) - creates a RDD that contains all the elements from the sourceRDD that are not in the secondRDD. Duplicates are allowed.

cartesian(secondRDD) - creates a RDD that contains all pairs of elements (x,y) where x can take all values from the sourceRDD and b can take all values from the secondRDD. Duplicates are allowed

- An action triggers the computation of all transformations through the dependency graph. The result is returned to the driver program.

Common actions

collect() - Returns all the RDD values.

count() - Return the number of values contained in the RDD.

take(n) - Returns the first n values from the RDD.

first() - Returns just the first value from the RDD.

saveAsTextFile(*path*) - Saves every value from the RDD into a text file.

reduce(f) - aggregates all elements using function f and returns result

1. Find all cities where 100 meter sprint competitions where organized in 2010.

2. Get all sprinter names.

3. Print out all possible pairs of sprinters. (sprinter1, sprinter2)

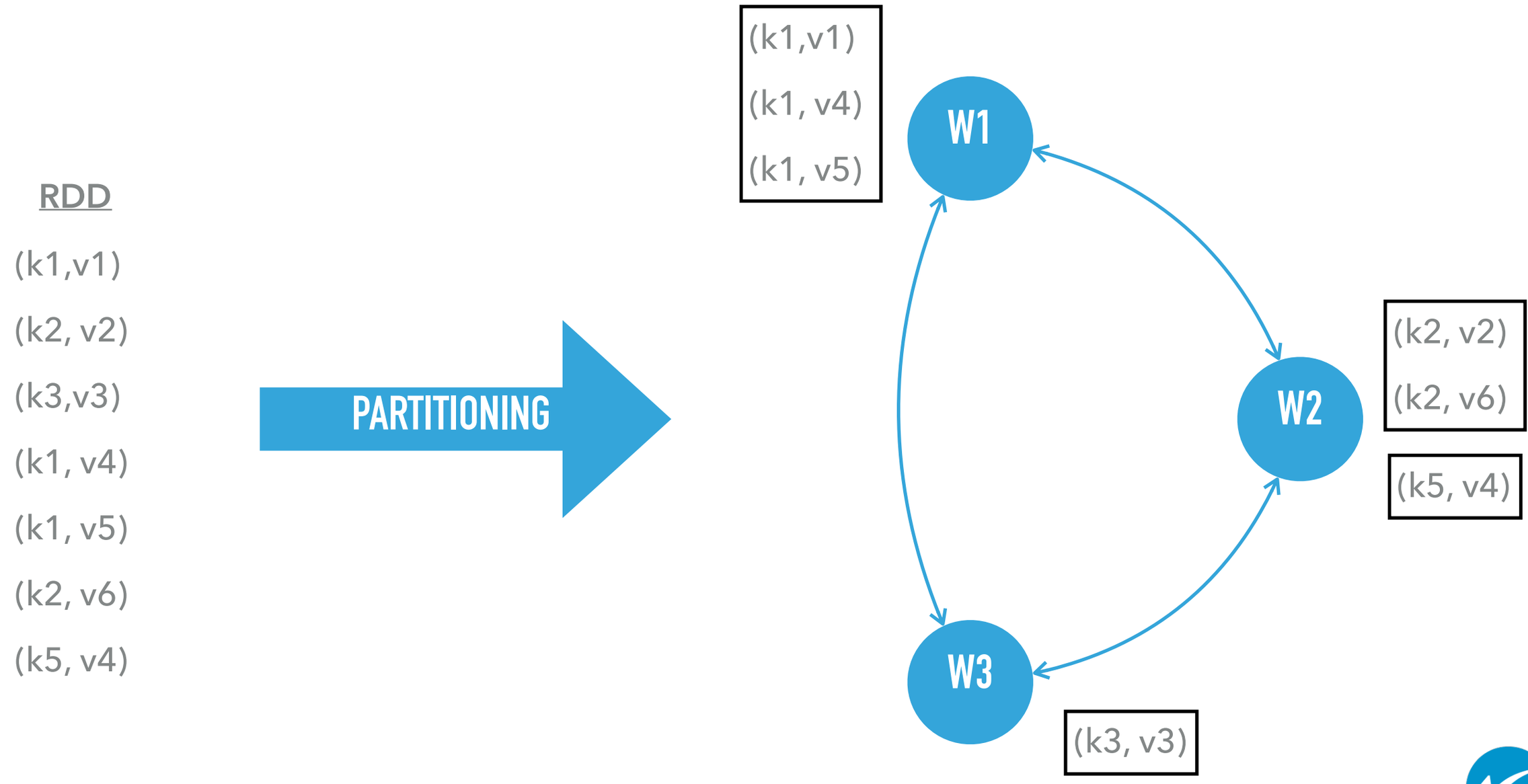
4. Print out the sum of all 100 meter sprint times.

PARTITIONING

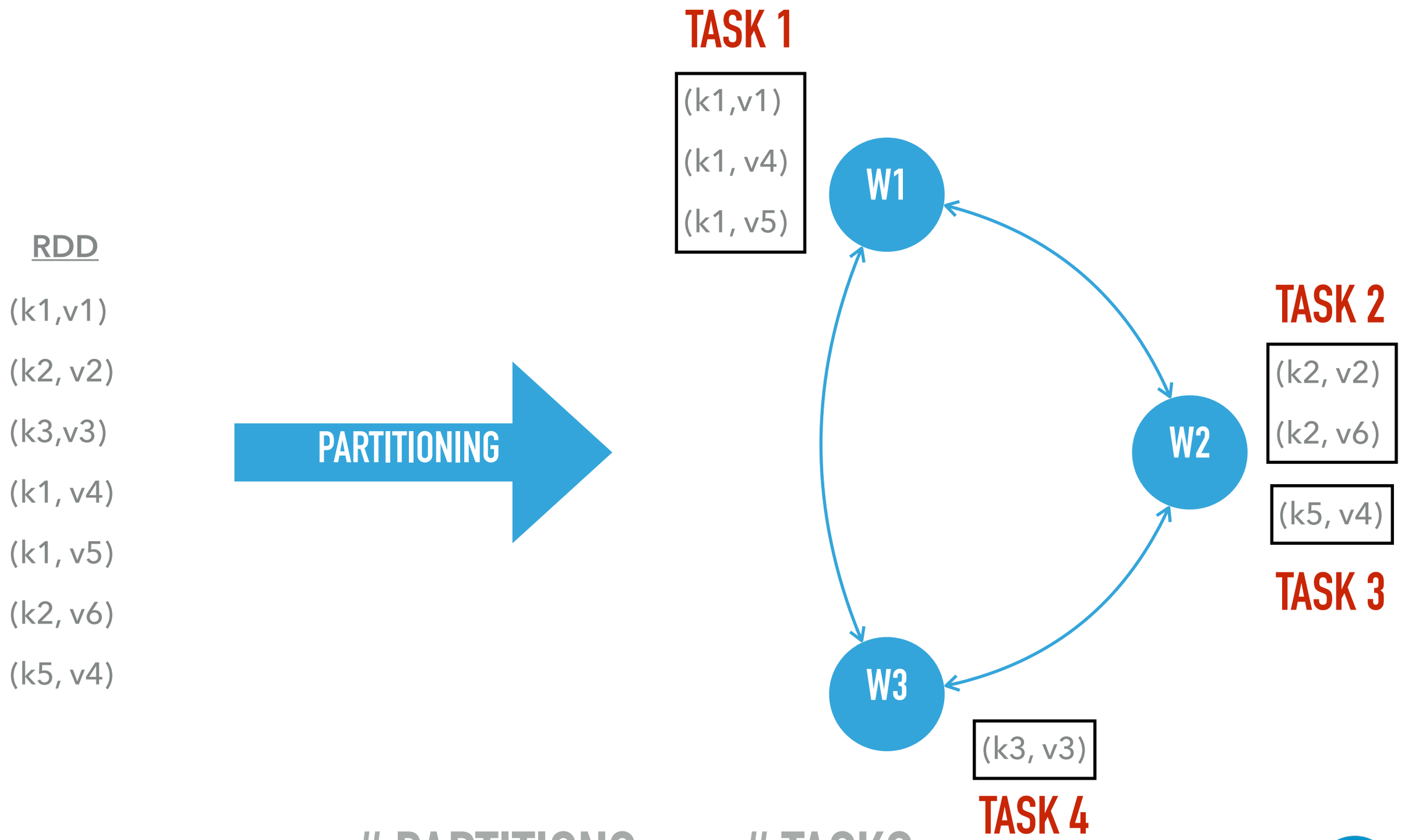
Because RDDs are too big to fit on one machine, they are divided into **partitions**.
Spark automatically does this for us and also distributes the partitions across the worker nodes.

rdd.partitions.size - returns the number of partitions the RDD is divided in.

rdd.partitioner - returns the concrete partitioner for that RDD. Can be *RangePartitioner, HashPartitioner* *None* (random and uniform) or a custom partitioner.



Computations are done on each partition. Each partition gets assigned a task in order to perform all the transformations and actions.



PARTITIONS == # TASKS

The number of partitions can be controlled by a spark configuration setting called **spark.default.parallelism**.

spark.default.parallelism default is the number of cores allocated for the Spark job.

Recommendation is to have **spark.default.parallelism** at least **2 times number of cores**.

EXAMPLE:

```
appName = properties.app_name
master = properties.spark_master
conf = SparkConf().setAppName(appName).setMaster(master)\
    .set("spark.executor.memory", "20g")\
    .set("spark.executor.cores", 6)\
    .set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")\
    .set("spark.dynamicAllocation.enabled", "true")\
    .set("spark.shuffle.service.enabled", "true")\
    .set("spark.default.parallelism", 2 * 9)\
    .set("spark.scheduler.mode", "FAIR")\
sparkContext = SparkContext(conf=conf)
```

WE CAN CONTROL

Number of partitions: in order to have at least a task per core.

Partitioner: in order to avoid shuffling multiple times.

NUMBER OF PARTITIONS

Too few: bad concurrency, memory issues, longer recovery, uneven partitions.

Too many: increased scheduling times, lot of lineage information.

2 X # CORES < NUMBER OF PARTITIONS < EACH TASK EXECUTION TIME AT LEAST 100MS

CHANGE NUMBER OF PARTITIONS FOR A RDD:

SPARK CONF

spark.default.parallelism = 2 * nrOfCores

TRANSFORMATION PARAMETERS

parallelize(…, **numPartitions**), textFile(…, **numPartitions**), reduceByKey(…, **numPartitions**), join(…, **numPartitions**)

REPARTITIONING TRANSFORMATIONS

rdd.coalesce(numPartitions) - repartitions RDD by joining existing partitions locally on each node, without requiring any shuffling. We can just decrease the number of partitions.

rdd.repartition(numPartitions) - repartitions RDD by shuffling the whole data. New partition count can be larger or smaller than the original.

pairRDD.**partitionBy(Partitioner)** - repartitions RDD into the same number of partitions, by using the partitioner given as parameter.

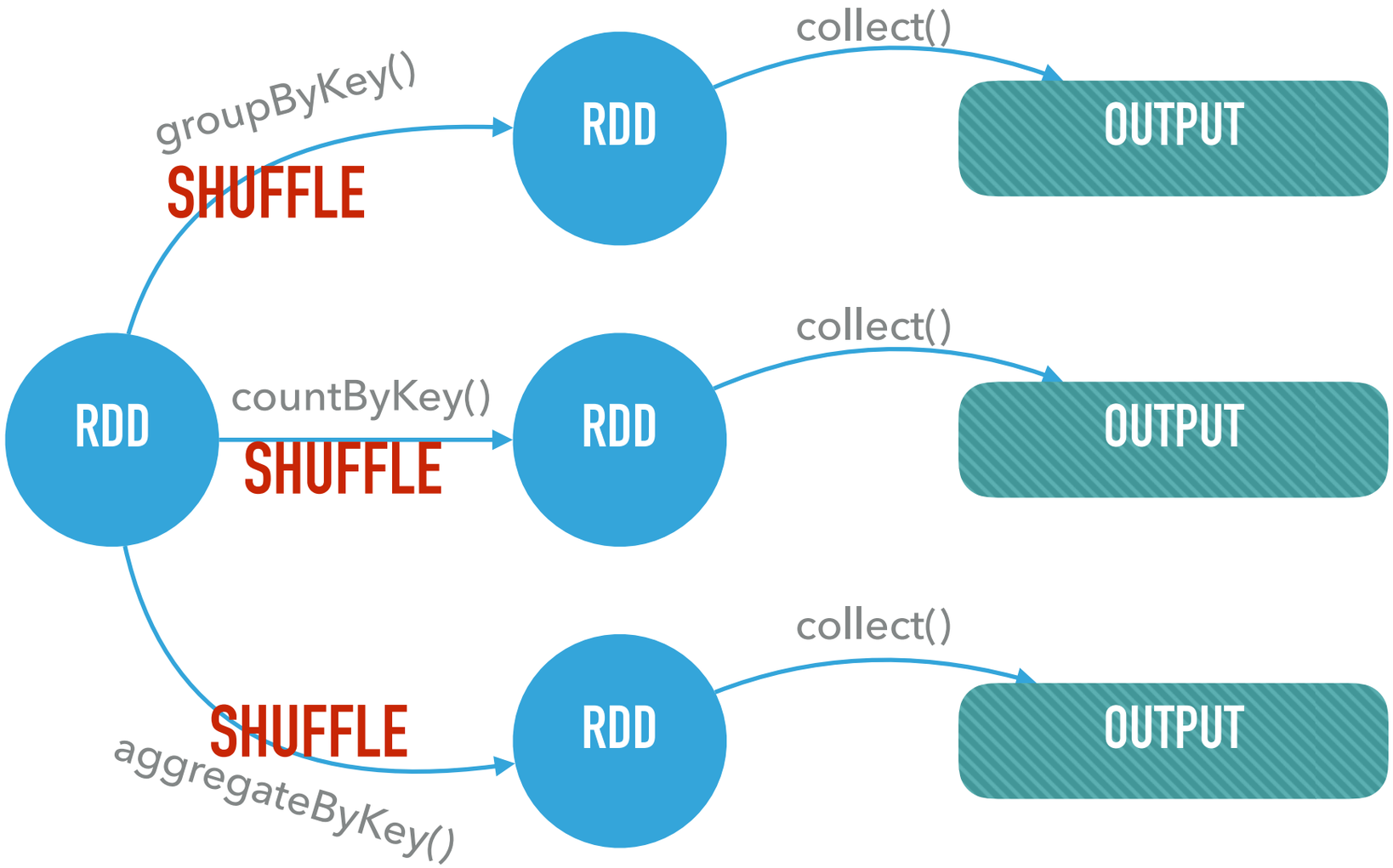
EXAMPLE:

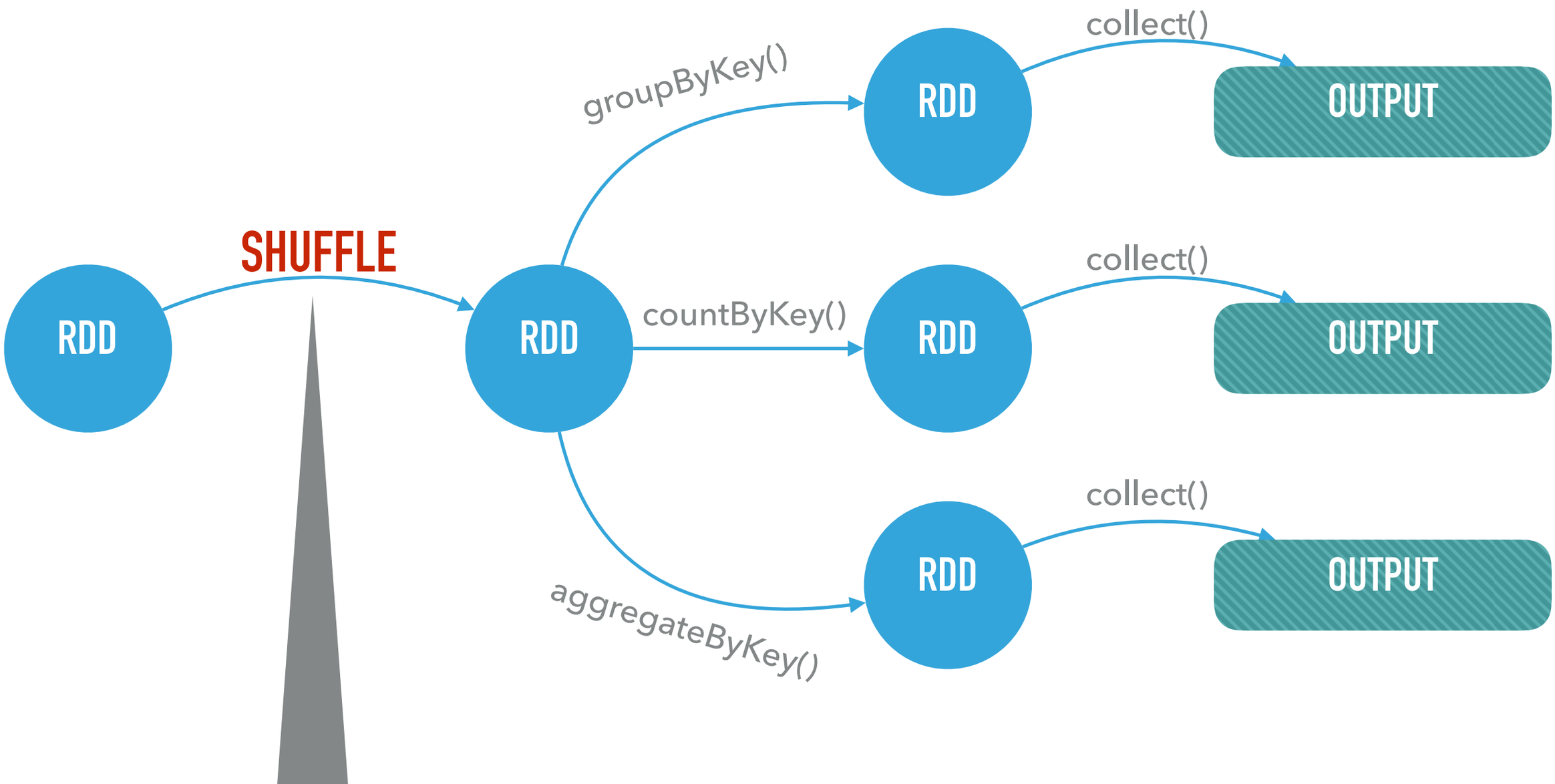
scala> **val pairs = sc.parallelize(List((1, "Alice"), (2, "Bob"), (3, "Eve")))**
pairs: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[0] at parallelize at <console>:27

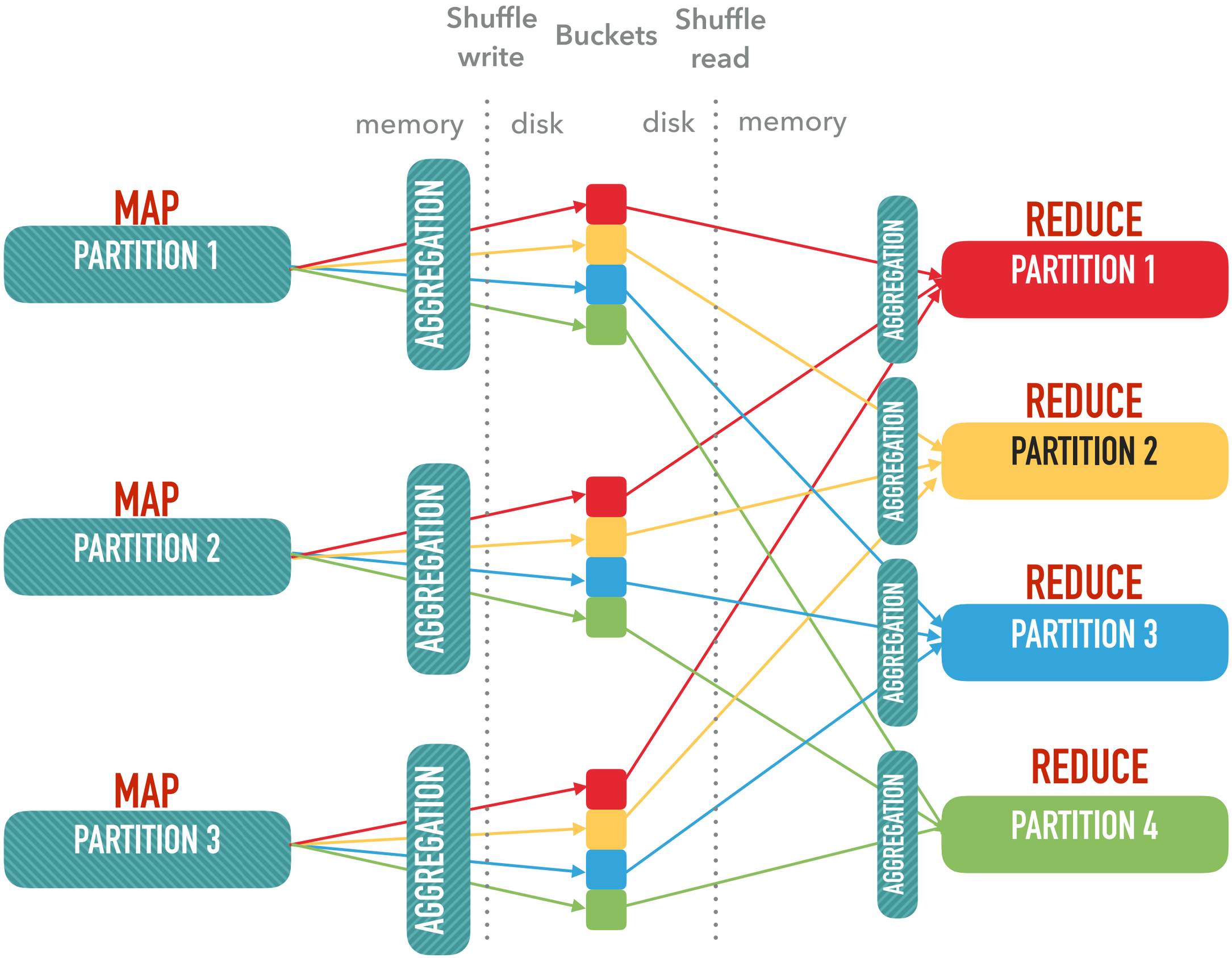
scala> **pairs.partitioner**
res0: Option[org.apache.spark.Partitioner] = None

scala> **val partitionedPairs = pairs.partitionBy(new org.apache.spark.HashPartitioner(2))**
partitioned: org.apache.spark.rdd.RDD[(Int, String)] = ShuffledRDD[1] at partitionBy at <console>:29

scala> **partitionedPairs.partitioner**
res1: Option[org.apache.spark.Partitioner] = Some(org.apache.spark.HashPartitioner@2)


groupByKey()
SHUFFLE
RDD
collect()
OUTPUT
RDD
countByKey()
SHUFFLE
RDD
collect()
OUTPUT
SHUFFLE
aggregateByKey()
RDD
collect()
OUTPUT


groupByKey()
RDD
collect()
OUTPUT
SHUFFLE
RDD
RDD
countByKey()
RDD
collect()
OUTPUT
collect()
RDD
OUTPUT
aggregateByKey()
partitionBy(new org.apache.spark.HashPartitioner(2 * sc.defaultParallelism)).cache()


Shuffle write
Buckets
Shuffle read
memory
disk
disk
memory
MAP
PARTITION 1
MAP
PARTITION 2
MAP
PARTITION 3
AGGREGATION
AGGREGATION
AGGREGATION
AGGREGATION
AGGREGATION
REDUCE
PARTITION 1
REDUCE
PARTITION 2
REDUCE
PARTITION 3
REDUCE
PARTITION 4

PAGERANK ALGORITHM EXAMPLE:

The algorithm maintains two datasets: one of **(pageID, linkList)** elements containing the list of neighbors of each page, and one of **(pageID, rank)** elements containing the current rank for each page. It proceeds as follows:

1. Initialize each page's rank to **1.0**.

2. On each iteration, have **page p** send a contribution of **rank(p)/ numNeighbors(p)** to its neighbors (the pages it has links to).

3. Set each page's rank to **0.15 + 0.85 * contributionsReceived**.

4. The last two steps repeat for several iterations, during which the algorithm will converge to the correct PageRank value for each page. In practice, it's typical to run about **10 iterations**.

SOLUTION:

scala> **var pageWithLinks = sc.objectFile[(Int, List[Int])]("/Users/viktor/links.obj")**
a: org.apache.spark.rdd.RDD[(Int, List[Int])] = MapPartitionsRDD[13] at objectFile at <console>:27

scala> **pageWithLinks.collect()**
res5: Array[(Int, List[Int])] = Array((1,List(2, 3, 4, 5)), (2,List(3, 6, 8)), (3,List(3, 8, 9)), (4,List(5)), (5,List(3)), (6,List(1)), (7,List(2)), (8,List(1)), (9,List(4)))

scala> **var ranks = *pageWithLinks*.map{case(pageId, links) => (pageId,1.0)}**

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

for (i <- 0 until 10) {
  val contributions = pageWithLinks.join(ranks).flatMap {
    case (pageId, (links, pageRank)) =>
      links.map(link => (link, pageRank / links.size))
  }
  ranks = contributions.reduceByKey((x, y) => x + y).map{case (key, value) => (key, 0.15 + 0.85*value)}
}
ranks.collect()
res21: Array[(Int, Double)] = Array((1,1.1486498813120 3), (2,0.3980419056085 3125),
(3,2.279429274035 5973), (4,1.079395267156 77), (5,1.329650085483 8882), (6,0.262872145019 0979),
(8,0.905853995371 9519), (9,0.792981850352 854))
```

OPTIMIZED SOLUTION:

```
scala> var pageWithLinks = sc.objectFile[(Int, List[Int])]("/Users/viktor/links.obj").partitionBy(new
org.apache.spark.HashPartitioner(100)).persist()
links: org.apache.spark.rdd.RDD[(Int, List[Int])] = ShuffledRDD[16] at partitionBy at <console>:27

scala> pageWithLinks.collect()
res5: Array[(Int, List[Int])] = Array((1,List(2, 3, 4, 5)), (2,List(3, 6, 8)), (3,List(3, 8, 9)), (4,List(5)), (5,List(3)),
(6,List(1)), (7,List(2)), (8,List(1)), (9,List(4)))

scala> var ranks = pageWithLinks.mapValues(links => 1.0)
```

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

for (i <- 0 until 10) {
  val contributions = pageWithLinks.join(ranks).flatMap {
    case (pageId, (links, pageRank)) =>
      links.map(link => (link, pageRank / links.size))
  }
  ranks = contributions.reduceByKey((x, y) => x + y).mapValues(contribution => 0.15 + 0.85*contribution)
}
ranks.collect()
res21: Array[(Int, Double)] = Array((1,1.1486498813120 3), (2,0.3980419056085 3125),
(3,2.279429274035 5973), (4,1.079395267156 77), (5,1.329650085483 8882), (6,0.262872145019 0979),
(8,0.905853995371 9519), (9,0.792981850352 854))
```

THANK YOU!

KEY-VALUE PAIRS

- A Pair RDD is a “special” RDD who's elements are (key,value) pairs RDD<K,V>.
- Having this structure special transformations can be performed on them.
- The key and value data type can be a primitive or a custom one.
- Besides all the transformations we saw earlier, Pair RDDs enable us to do **grouping**, **aggregations** and **joins**.

keys() - creates a RDD that contains just the "key" elements of the original RDD.

values() - creates a RDD that contains just the "value" elements of the original RDD.

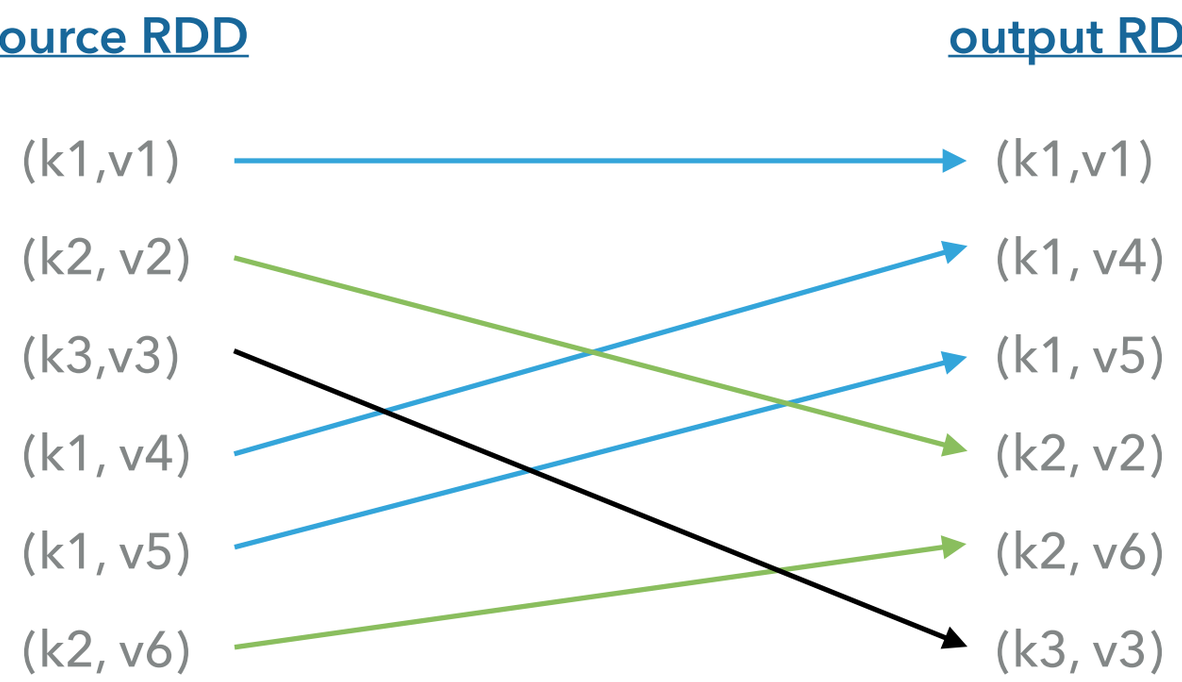
can be done also with map

mapValues(f) - creates a RDD by applying f just on the values of the original RDD, leaving the keys untouched (one-to-one correspondence). **Preserves key based partitioning!**

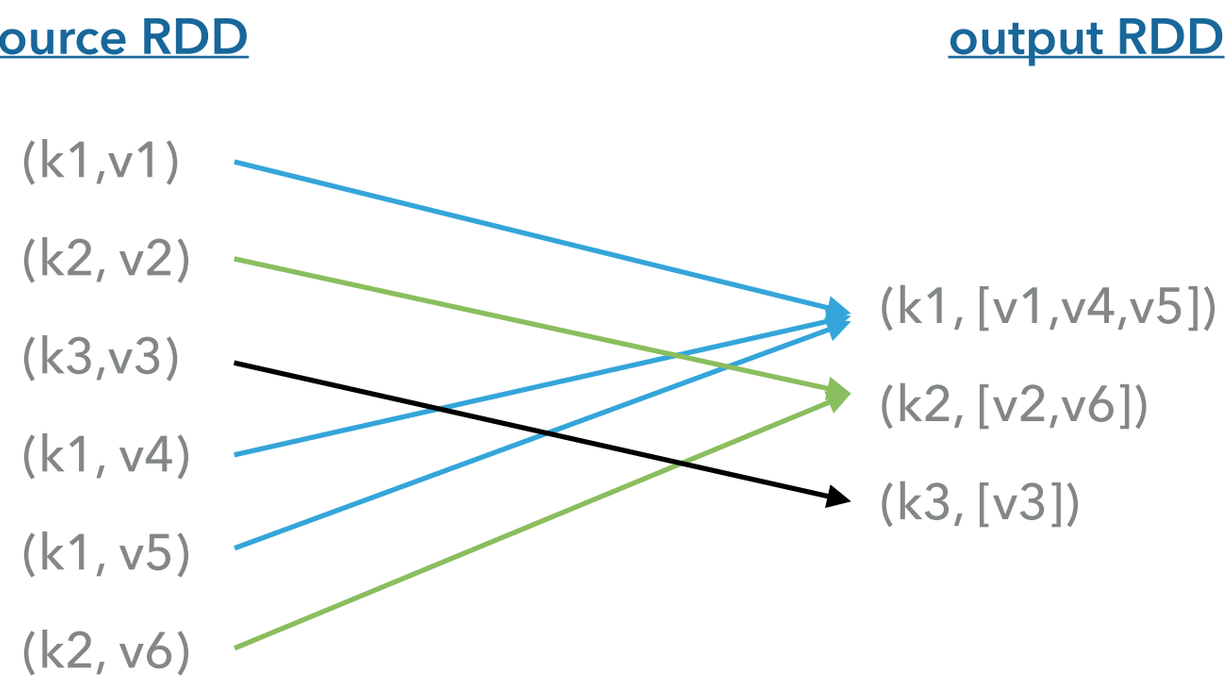
flatMapValues(f) - same as mapValues() with the difference that f returns a Seq of values. (one-to-many correspondence)

using map instead, will disturb the partitioning

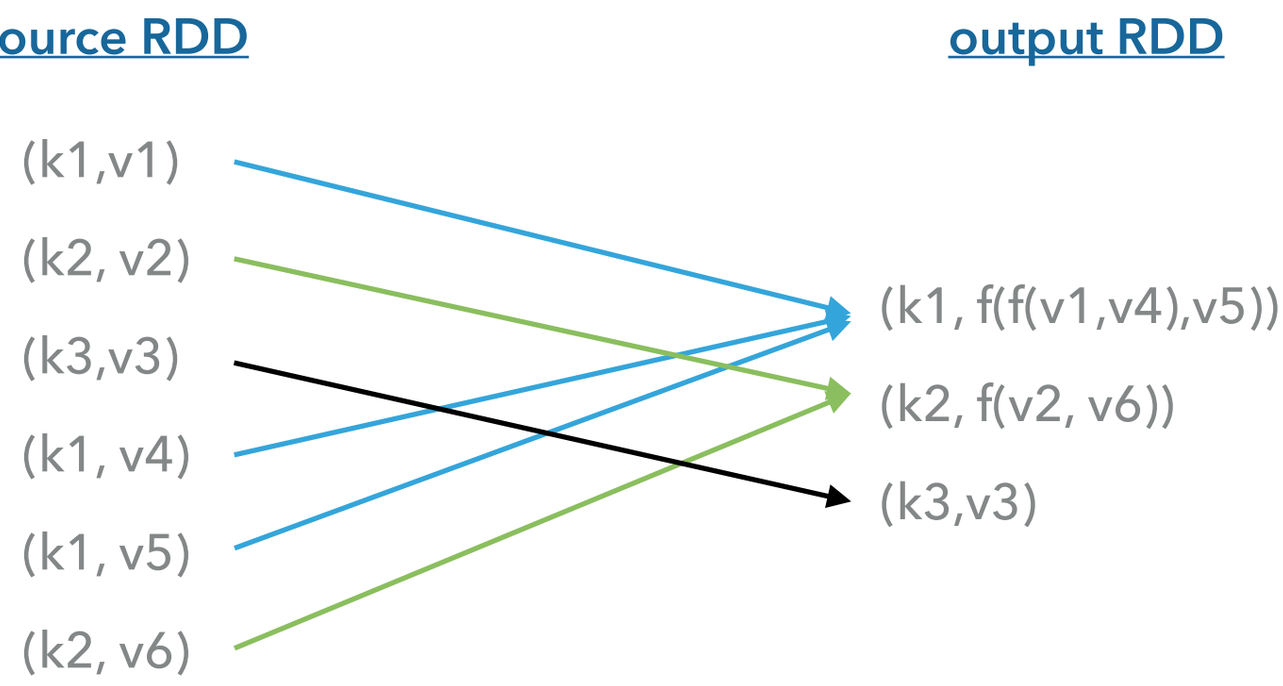
sortByKey([ascending], [numtasks]) - creates a new Pair RDD sorting elements by key in ascending (default) or descending order <sortByKey(false)>.



groupByKey(f, [numtasks]) - creates a new Pair RDD of (K, Iterable<V>) from (K,V) by grouping all the values with the same key.



reduceByKey(f, [numtasks]) - creates a new Pair RDD by aggregating every values that have the same key. The aggregation is performed by the f function that takes two values as parameters and returns an aggregated value. “numtasks” can be specified if we want the resulting RDD to have a specific number of partitions.

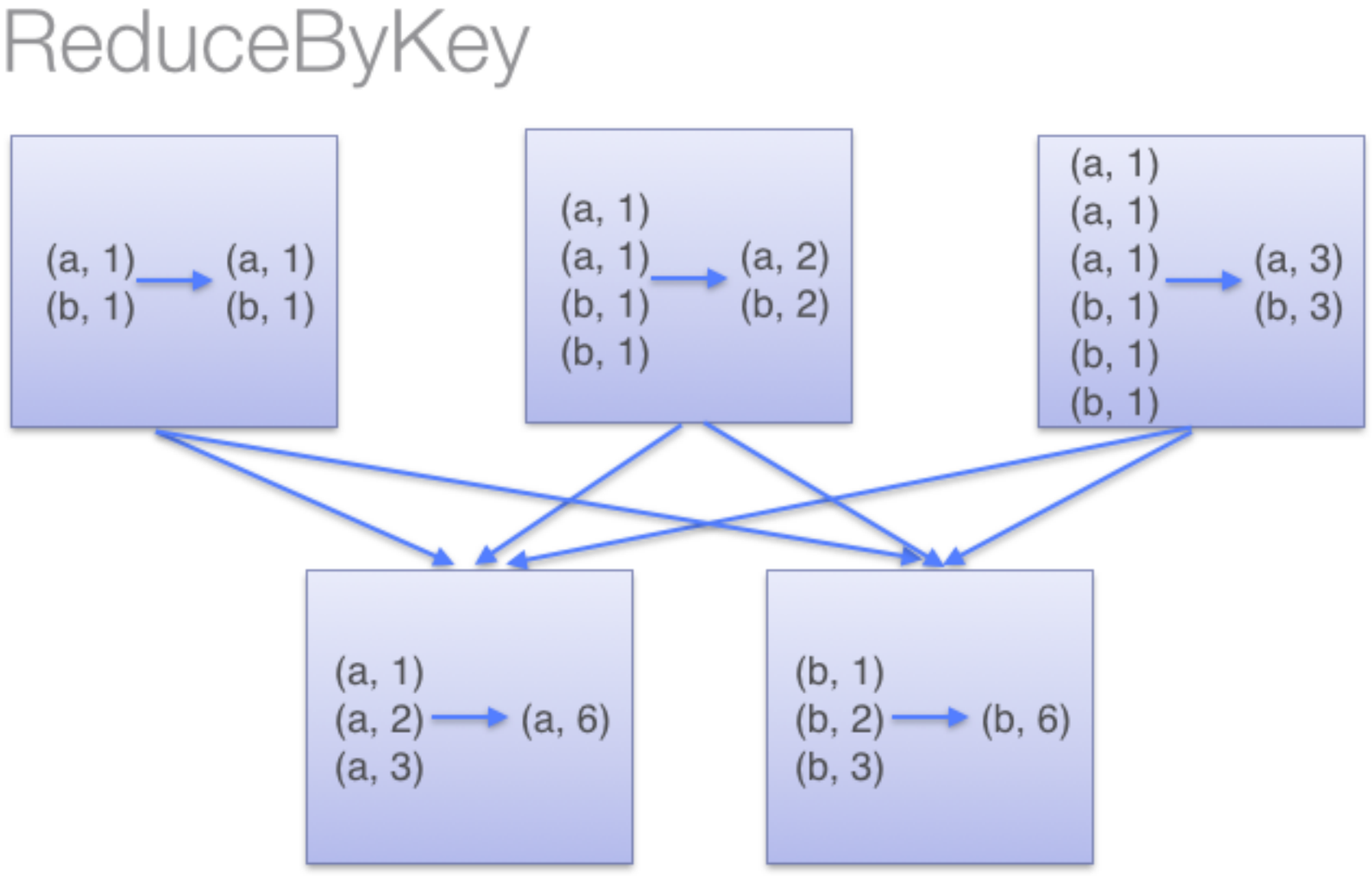


Avoid groupByKey()!

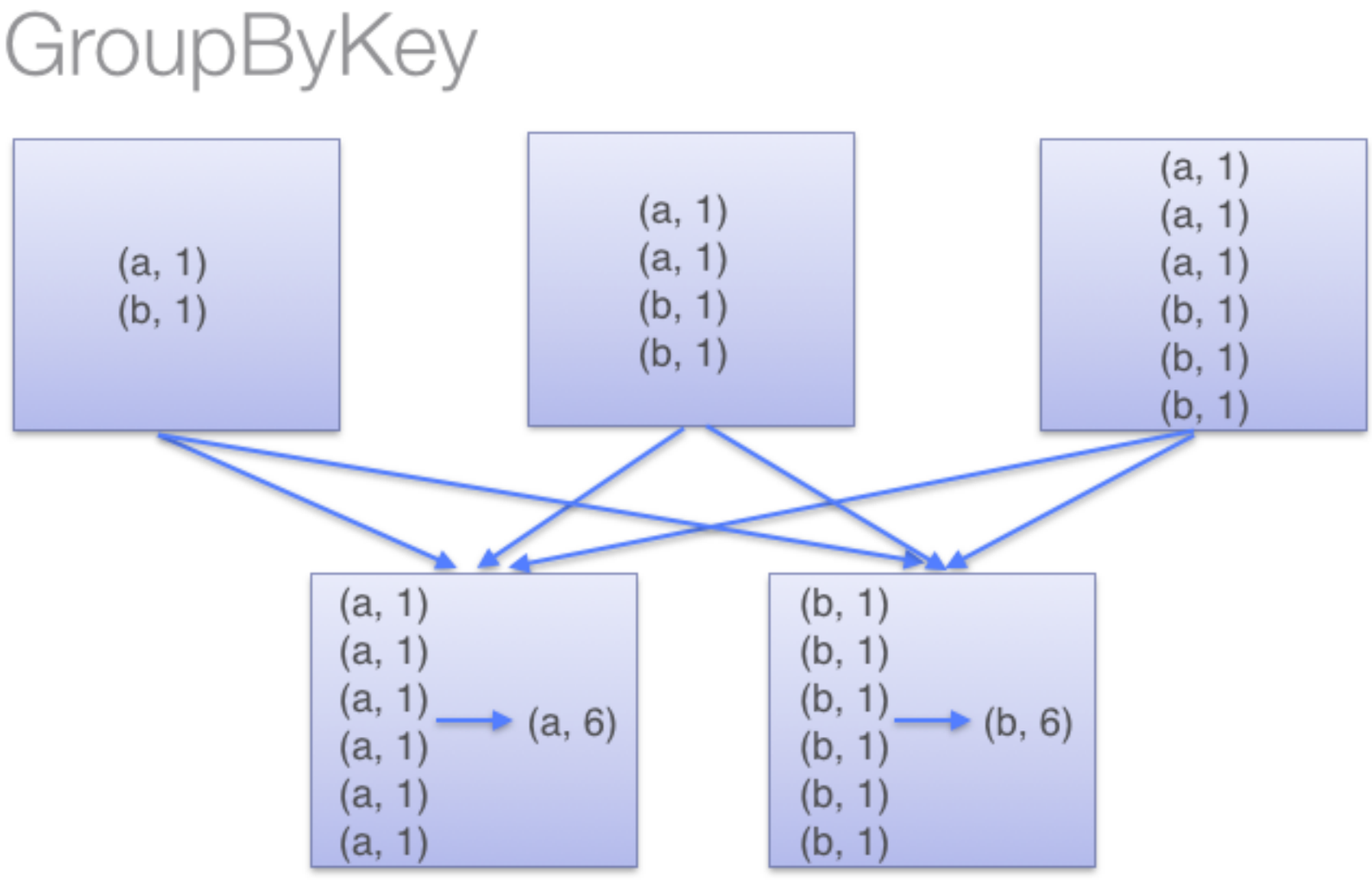
Use reduceByKey() or aggregateByKey() instead!

```
val words = Array("one", "two", "two", "three", "three", "three")
val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))

val wordCountsWithReduce = wordPairsRDD
  .reduceByKey(_ + _)
  .collect()

val wordCountsWithGroup = wordPairsRDD
  .groupByKey()
  .map(t => (t._1, t._2.sum))
  .collect()
```

ReduceByKey combines values with the same key locally first.



GroupByKey shuffles all the values and combines them later.



KEEP IN MIND:

- **Grouping** and **sorting** can be very **expensive** on large datasets.
- For aggregations do not use **groupByKey**(), use instead **aggregateByKey**() because it is more efficient.
- Grouping can result in (key,value) pairs where the value is very large. This can cause **OutOfMemory** errors because a single RDD element must reside on one executor. Even if it fits into memory, the result can contain disproportional (key,value) pairs. Executing further transformation on them will be very slow, because the big (key, value) pair will be executed on only one core.
- In this case a solution is to group first with a key that distributes the values more evenly.

1. Find out how many times Usain Bolt won in every year.

2. Get the best sprint times Usain Bolt got for every year, ordered by year descending.

THANK YOU!