Live Coding

```
import pyspark as ps
import random

# initialize SparkContext
sc = ps.SparkContext()

flips = 1000000

# lazy eval
coins = xrange(flips)

# lazy eval, nothing executed
heads = sc.parallelize(coins) \
          .map(lambda i: random.random()) \
          .filter(lambda r: r < 0.51) \
          .count()

print "{0} heads flipped!".format(heads)

# shutdown SparkContext
sc.stop()
```

What is an RDD?

- **Resilient:** if the data in memory (or on a node) is lost, it can be recreated
- **Distributed:** data is chunked into partitions and stored in memory across the cluster
- **Dataset:** initial data can come from a file or be created programmatically

Note: RDDs are read-only and immutable, we will come back to this later…

Functions Deconstructed

```
import random
flips = 1000000

# lazy eval
coins = range(flips)

# lazy eval, nothing executed
heads = sc.parallelize(coins) \
          .map(lambda i: random.random()) \
          .filter(lambda r: r < 0.51) \
          .count()
```

Python Generator

Create RDD

Transformations

Action (materialize result)

Functions Deconstructed

```
import random
flips = 1000000

# lazy eval
coins = range(flips)

# lazy eval, nothing executed
heads = sc.parallelize(coins) \
        .map(lambda i: random.random()) \
        .filter(lambda r: r < 0.51) \
        .count()


 # create a closure with the lambda function
 # apply function to data
```

Closures

Functions Deconstructed

```
import random
flips = 1000000

# local sequence
coins = range(flips)

# distributed sequence
coin_rdd = sc.parallelize(coins)
flips_rdd = coin_rdd.map(lambda i: random.random())
heads_rdd = flips_rdd.filter(lambda r: r < 0.51)

# local value
head_count = heads_rdd.count()
```

Functional Programming Primer

- Functions are `applied` to data (RDDs)
- RDDs are Immutable: `f(RDD) -> RDD2`
- Function application necessitates creation of new data

Spark Functions

Transformations	Actions
Lazy Evaluation (does not immediately evaluate)	Materialize Data (evaluates RDD lineage)
Returns new RDD	Returns final value (on driver)

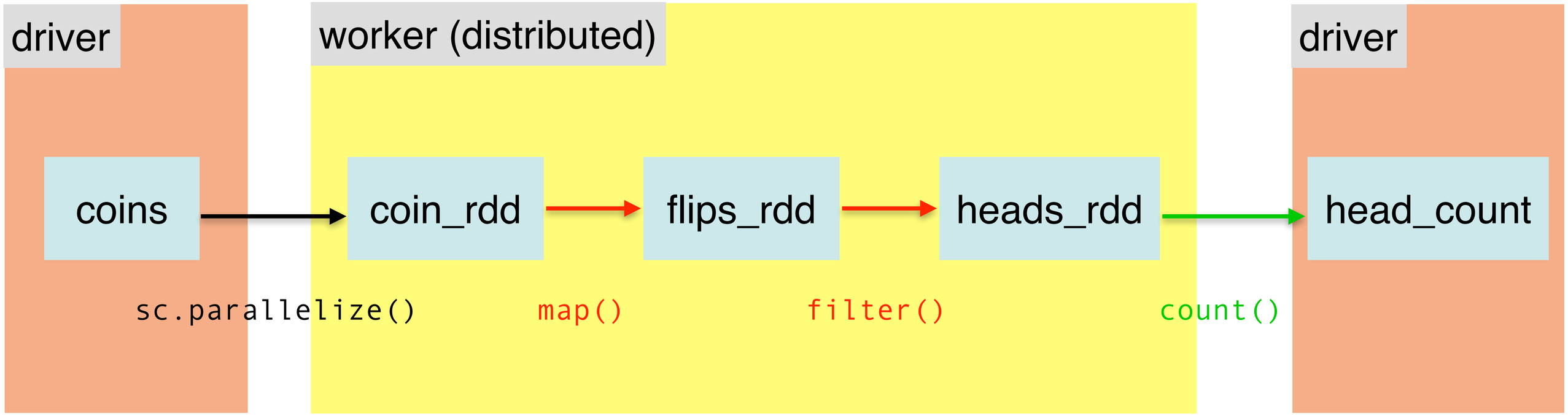
Transformations

```
# Every Spark application requires a Spark Context
# Spark shell provides a preconfigured Spark Context called `sc`
nums = sc.parallelize([1,2,3])

# Pass each element through a function
squared = nums.map(lambda x: x*x) # => {1, 4, 9}

# Keep elements passing a predicate
even = squared.filter(lambda x: x % 2 == 0) # => [4]

# Map each element to zero or more others
nums.flatMap(lambda x: range(x)) # => {0, 0, 1, 0, 1, 2}
```

Actions

```
nums = sc.parallelize([1, 2, 3])

# Retrieves RDD contents as a local collection
nums.collect() # => [1, 2, 3]

# Returms first K elements
nums.take(2) # => [1, 2]

# Count number of elements
nums.count() # => 3

# Merge elements with an associative function
nums.reduce(lambda: x, y: x + y) # => 6

# Write elements to a text file
nums.saveAsTextFile("hdfs://file.txt")
```

RDD Lineage


driver
worker (distributed)
driver
coins
coin_rdd
flips_rdd
heads_rdd
head_count
sc.parallelize()
map()
filter()
count()

Functions Deconstructed

```
import random
flips = 1000000

# lazy eval
coins = xrange(flips)
```

nothing runs here

```
# lazy eval, nothing executed
heads_rdd = sc.parallelize(coins) \
          .map(lambda i: random.random()) \
          .filter(lambda r: r < 0.51)
```

```
head_count = heads_rdd.count()
```

Everything runs here

What is an RDD?

An *Abstraction!*

What is an RDD?

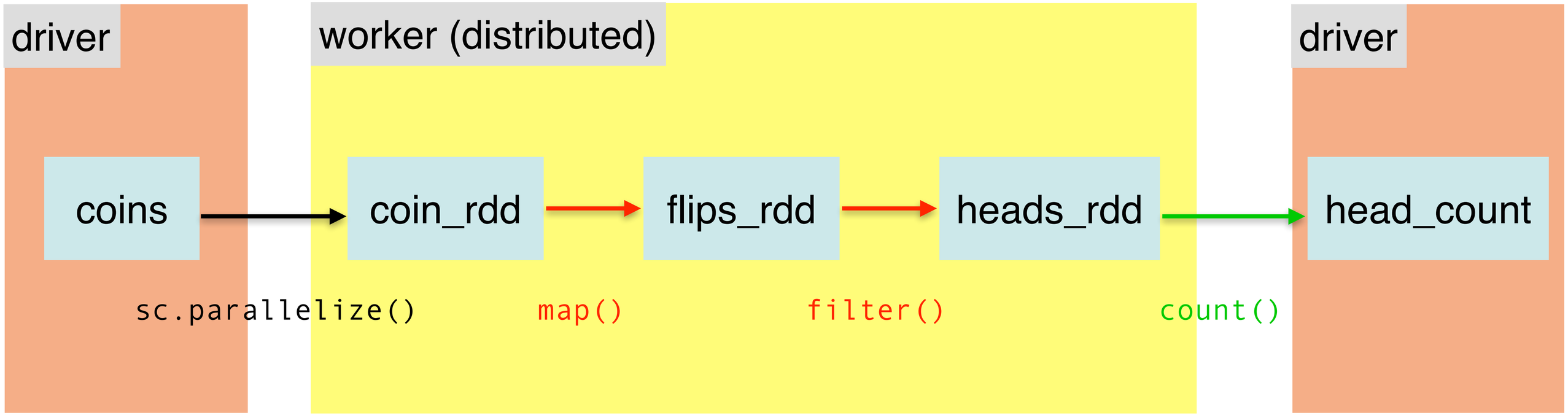
Lineage (required)

1. Set of partitions for current RDD (data)
2. List of dependencies
3. Function to compute partitions (functional paradigm)

4. Partitioner to optimize execution
5. Potential preferred location for partitions

Optimization (optional)

RDD Lineage


driver
worker (distributed)
driver
coins
coin_rdd
flips_rdd
heads_rdd
head_count
sc.parallelize()
map()
filter()
count()

What is an RDD?

Lineage (required)

1. Set of partitions for current RDD (data)
2. List of dependencies
3. Function to compute partitions (functional paradigm)

4. Partitioner to optimize execution
5. Potential preferred location for partitions

Optimization (optional)

RDD as an interface

Operation	*Meaning*
`partitions()`	Return a list of `Partition` objects
`dependencies()`	Return a list of dependencies
`compute(p, parent)`	Compute the elements of `Partition` ***p*** given its ***parent*** `Partitions`
`partitioner()`	Return metadata specifying whether this RDD is hash/range partitioned
`preferredLocations(p)`	List nodes where `Partition` ***p*** can be accessed quicker due to data locality

```
file_rdd.map(line => line.split(""))
   .map(split => (split(0), split(1).toInt))
   .groupByKey()
   .mapValues(iter => iter.reduce(_ + _)).collect()
```

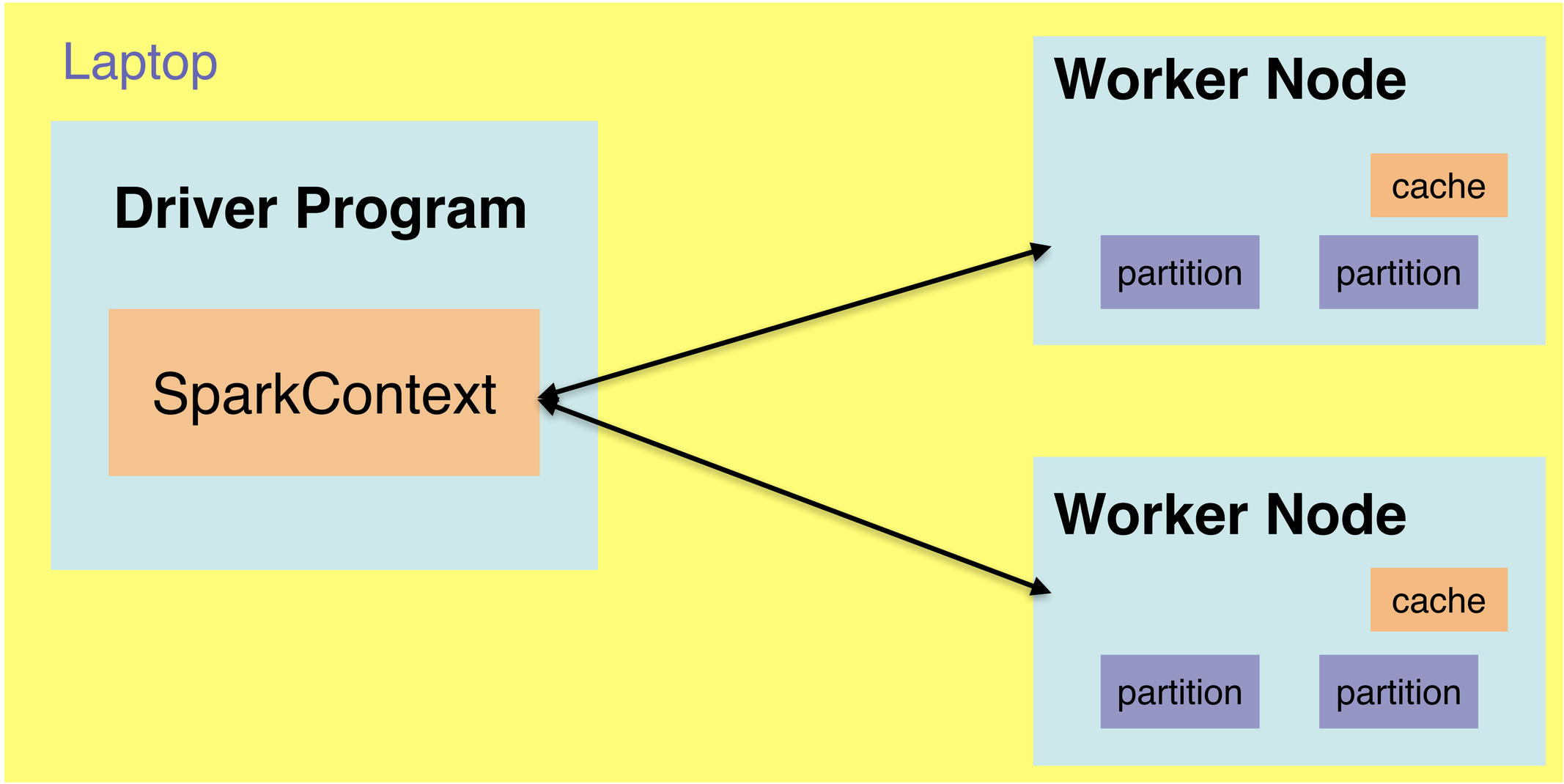
dummy.txt

```
jon 2
mary 3
anna 1
jon 1
jesse 3
mary 5
```

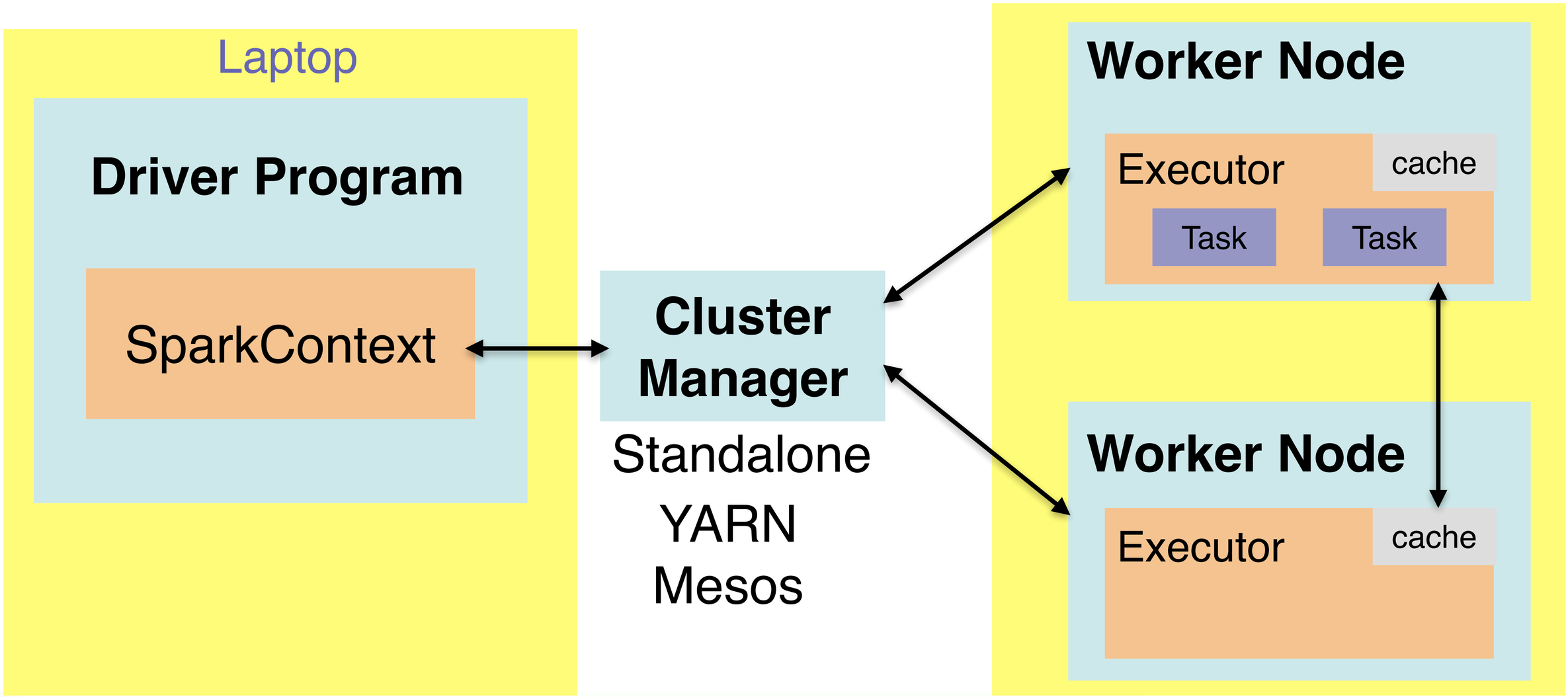
Key-Value Operations

```
pets = sc.parallelize([("cat", 1), ("dog", 1), ("cat", 2)])

pets.reduceByKey(lambda x, y: x + y) # => {(cat, 3), (dog, 1)}

pets.groupByKey() # => {(cat, [1, 2]), (dog, [1])}

pets.sortByKey() # => {(cat, 1), (cat, 2), (dog, 1)}
```

Local Spark Execution Context



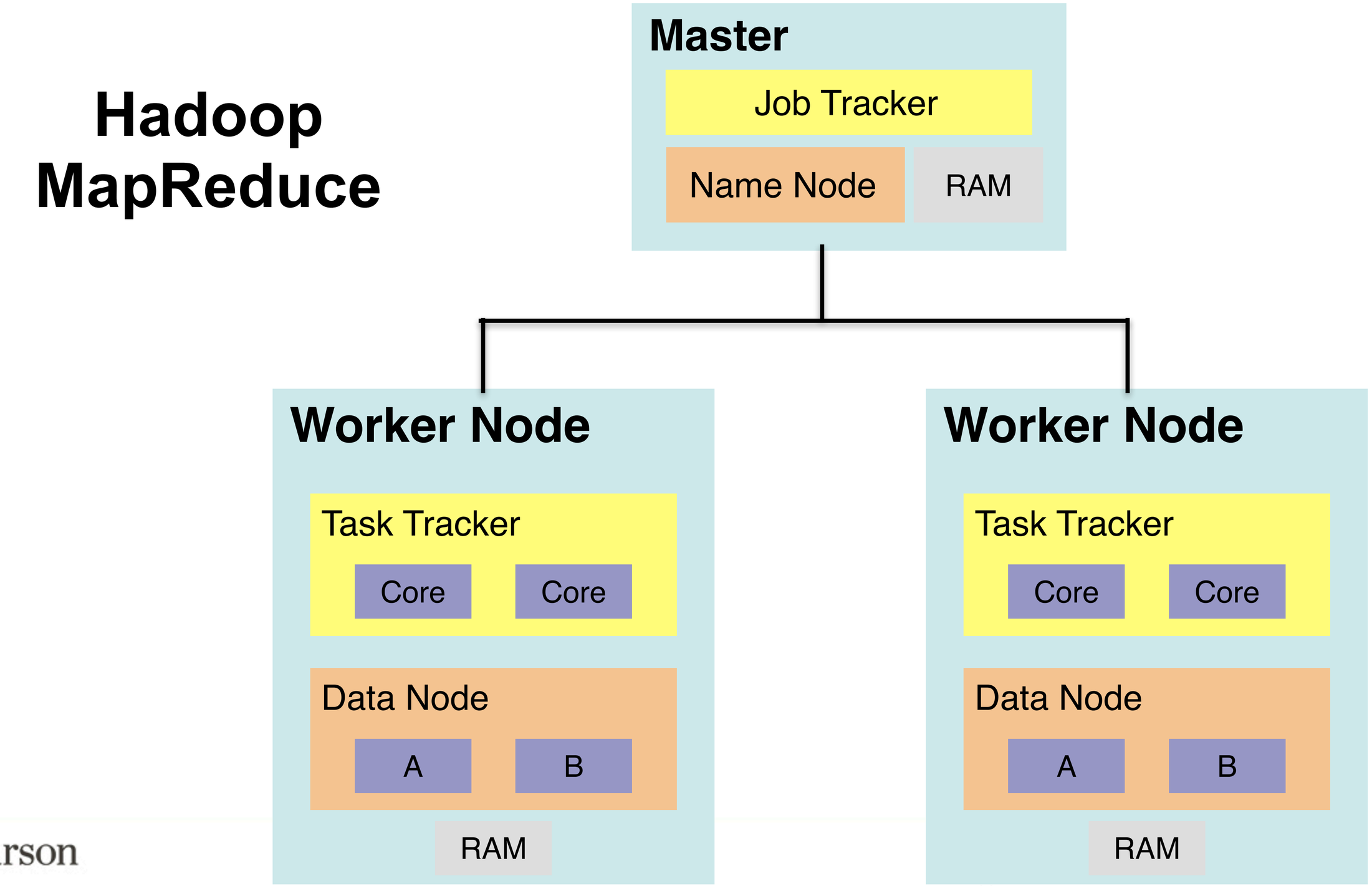
Spark on a Cluster



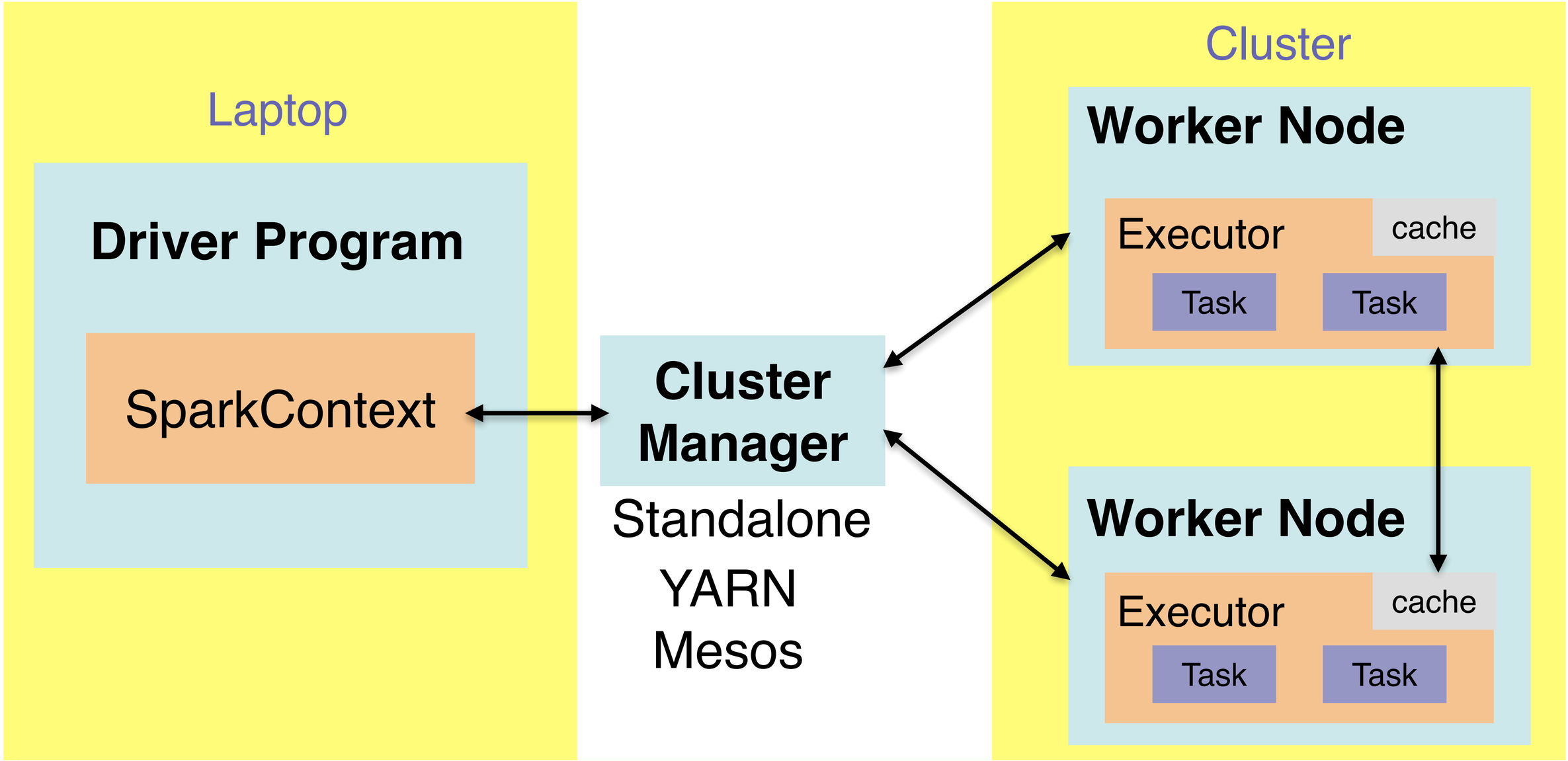
Terminology

Term	*Meaning*
Driver	*Process that contains the SparkContext*
Executor	*Process that executes one or more Spark tasks*
Master	*Process that manages applications across the cluster*
Worker	*Process that manages executors on a particular node*

Hadoop MapReduce



Spark Execution Context



Spark vs. Hadoop

- Spark only replaces MapReduce (computation)
- Still need a data store: HDFS, HBase, Hive, etc.
- Spark has a more flexible/general programming model
- Spark often faster for iterative computation

A Day in the Life of a Spark Application

1. Determine RDD lineage: construct DAG

2. Create execution plan for DAG: determine stages

3. Schedule and execute tasks

```
file_rdd.map(line => line.split(""))
   .map(split => (split(0), split(1).toInt))
   .groupByKey()
   .mapValues(iter => iter.reduce(_ + _)).collect()
```

dummy.txt

```
jon 2
mary 3
anna 1
jon 1
jesse 3
mary 5
```

What's in a Task?

```
sc.textFile('file:///dummy.txt')
```

```
map(line => line.split(" "))
```

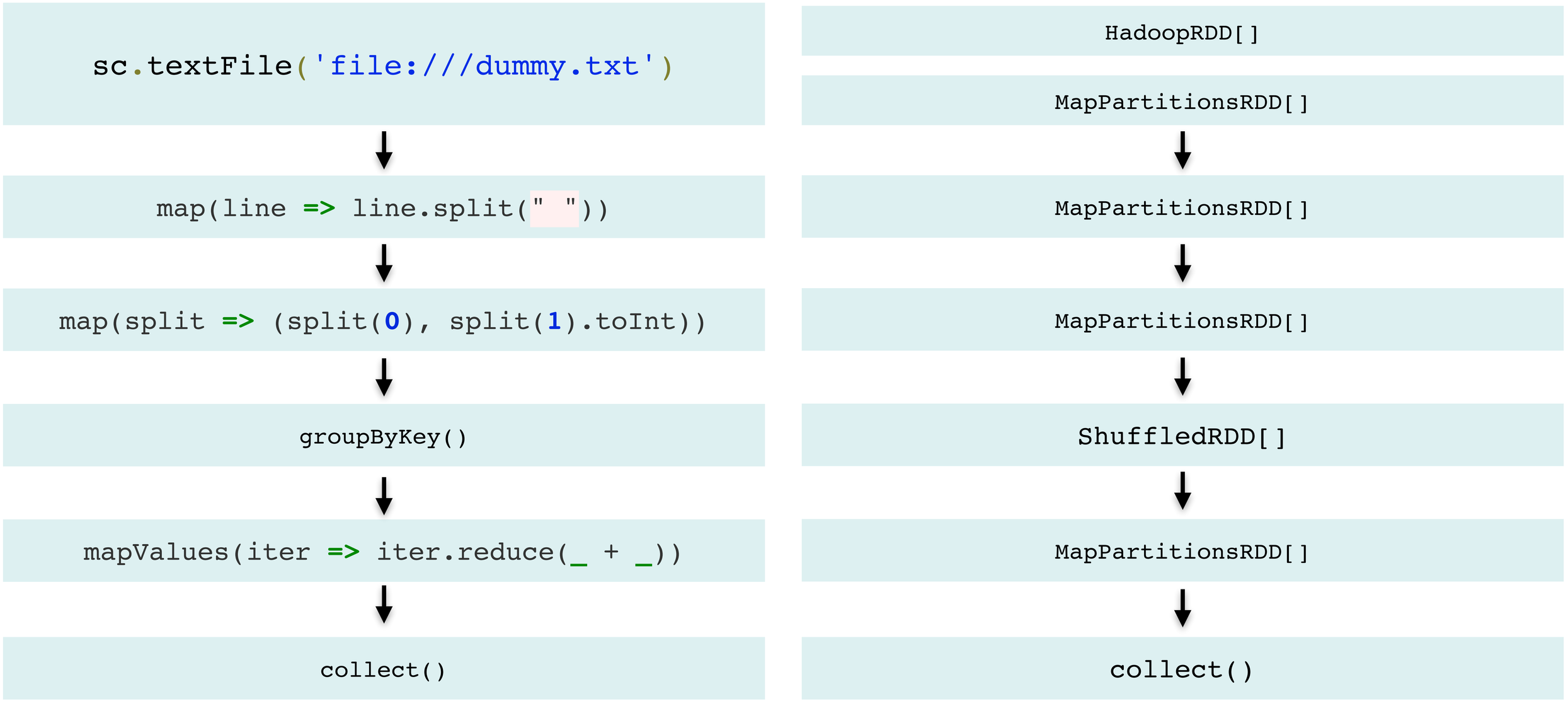
```
map(split => (split(0), split(1).toInt))
```

```
groupByKey()
```

```
mapValues(iter => iter.reduce(_ + _))
```

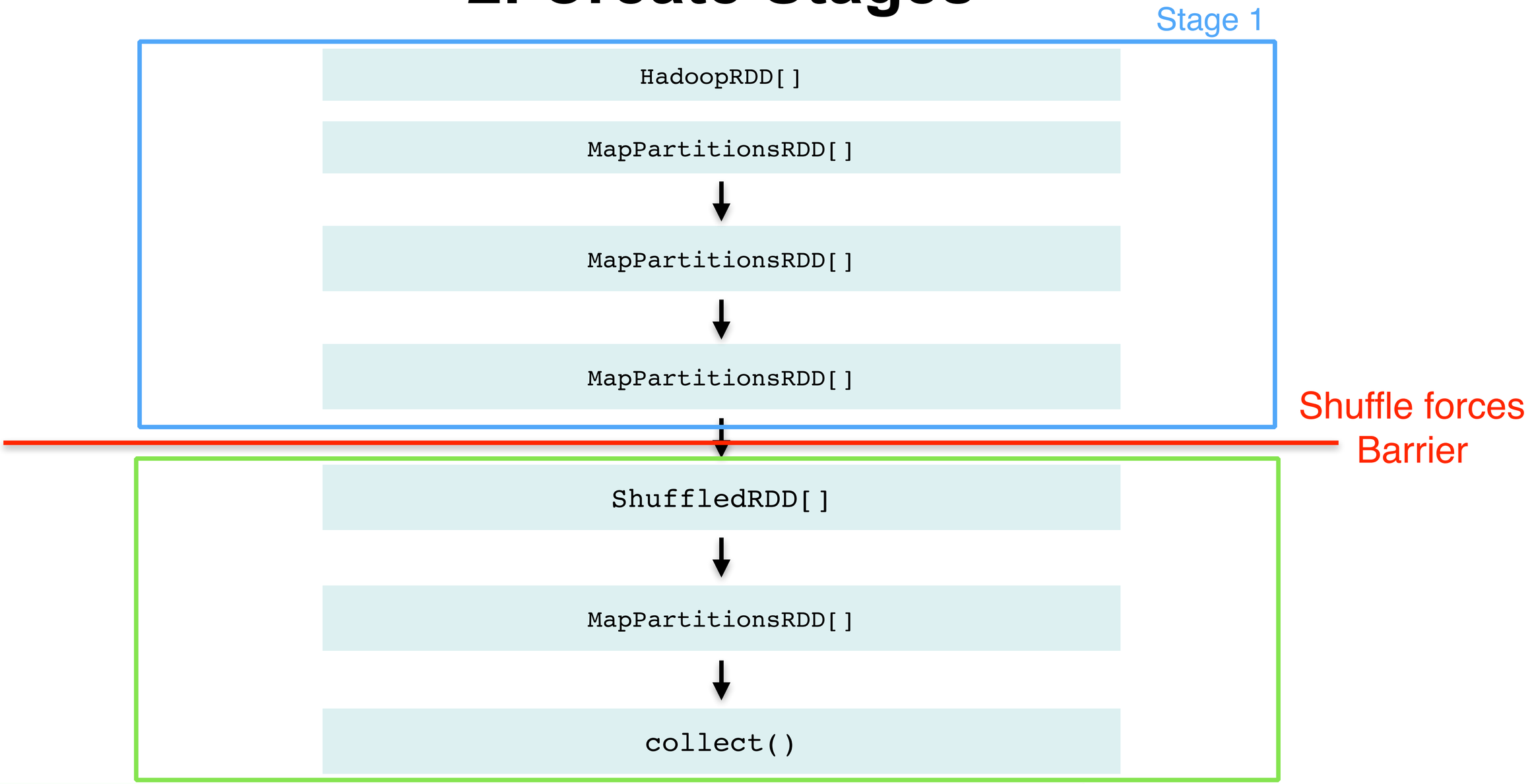
```
collect()
```

1. Create RDDs



```
scala> file_rdd.partitions
res51: Array[org.apache.spark.Partition] = Array(org.apache.spark.rdd.HadoopPartition@a17, org.apache.spark.rdd.HadoopPartition@a18)

scala> file_rdd.partitions.size
res52: Int = 2

scala> file_rdd.map(line => line.split(" ")).toDebugString
res53: String =
(2) MapPartitionsRDD[65] at map at <console>:24 []
 |  MapPartitionsRDD[23] at textFile at <console>:21 []
 |  file:///Users/jonathandinu/spark-ds-applications/dummy.txt HadoopRDD[22] at textFile at <console>:21 []

scala> file_rdd.map(line => line.split(" ")).map(split => (split(0), split(1).toInt)).toDebugString
res54: String =
(2) MapPartitionsRDD[67] at map at <console>:24 []
 |  MapPartitionsRDD[66] at map at <console>:24 []
 |  MapPartitionsRDD[23] at textFile at <console>:21 []
 |  file:///Users/jonathandinu/spark-ds-applications/dummy.txt HadoopRDD[22] at textFile at <console>:21 []

scala> file_rdd.map(line => line.split(" ")).map(split => (split(0), split(1).toInt)).groupByKey().toDebugString
res55: String =
(2) ShuffledRDD[70] at groupByKey at <console>:24 []
 +-(2) MapPartitionsRDD[69] at map at <console>:24 []
    |  MapPartitionsRDD[68] at map at <console>:24 []
    |  MapPartitionsRDD[23] at textFile at <console>:21 []
    |  file:///Users/jonathandinu/spark-ds-applications/dummy.txt HadoopRDD[22] at textFile at <console>:21 []

scala> file_rdd.map(line => line.split(" ")).map(split => (split(0), split(1).toInt)).groupByKey().mapValues(iter => iter.reduce(_ + _)).toDebugString
res56: String =
(2) MapPartitionsRDD[74] at mapValues at <console>:26 []
 |  ShuffledRDD[73] at groupByKey at <console>:26 []
 +-(2) MapPartitionsRDD[72] at map at <console>:26 []
    |  MapPartitionsRDD[71] at map at <console>:26 []
    |  MapPartitionsRDD[23] at textFile at <console>:21 []
    |  file:///Users/jonathandinu/spark-ds-applications/dummy.txt HadoopRDD[22] at textFile at <console>:21 []

scala> file_rdd.map(line => line.split(" ")).map(split => (split(0), split(1).toInt)).groupByKey().mapValues(iter => iter.reduce(_ + _)).collect()
res57: Array[(String, Int)] = Array((anna,1), (jesse,3), (jon,3), (mary,8))
```

2. Create Stages



3. Schedule Tasks

1. Split each stage into tasks to execute

2. Task is simply a partition (data) and computation (`lambda`)

3. Execute all tasks in a stage before continuing

3. Schedule Tasks

Stage 1

```
sc.textFile('file:///dummy.txt')
        ↓
map(line => line.split(" "))
        ↓
map(split => (split(0), split(1).toInt))
```

dummy.txt

```
jon 2
mary 3
anna 1
jon 1
```

```
jesse 1
mary 5
```

Task 0

Task 1

Task 0

```
partition 0
     ↓
HadoopRDD
     ↓
map(line => line.split(" "))
     ↓
map(split => (split(0), split(1).toInt))
```

Task 1

```
partition 1
     ↓
HadoopRDD
     ↓
map(line => line.split(" "))
     ↓
map(split => (split(0), split(1).toInt))
```

Scheduler

- RDD and partition DAG as input
- Tasks (within stages) to execute as output

Roles

- Builds stages to execute
- Submit stages to Cluster Manager
- Resubmit failed stages when output is lost

A Day in the Life of a Spark Application