Spark Introduction

Axel Jacquin

Adaltas

8th November 2016

Up Until Now

Up Until Now :

- Cheap hard drives vs expansive RAM -> MapReduce

Up Until Now

Up Until Now :

- Cheap hard drives vs expansive RAM -> MapReduce
- Limited to MapReduce & Batch-processing

But today

But today :

- RAM is cheap : 192GB of RAM servers

But today

But today :

- RAM is cheap : 192GB of RAM servers
- Virtual machines everywhere

But today

But today :

- RAM is cheap : 192GB of RAM servers
- Virtual machines everywhere
- Multi-core machine

But today

But today :

- RAM is cheap : 192GB of RAM servers
- Virtual machines everywhere
- Multi-core machine

=>

In-Memory Processing :

But today

But today :

- RAM is cheap : 192GB of RAM servers
- Virtual machines everywhere
- Multi-core machine

=>

In-Memory Processing :

- Load your data once

But today

But today :

- RAM is cheap : 192GB of RAM servers
- Virtual machines everywhere
- Multi-core machine

=>

In-Memory Processing :

- Load your data once
- Use all of your cluster's RAM

But today

But today :

- RAM is cheap : 192GB of RAM servers
- Virtual machines everywhere
- Multi-core machine

=>

In-Memory Processing :

- Load your data once
- Use all of your cluster's RAM
- Play with it for as long as you want

But today

But today :

- RAM is cheap : 192GB of RAM servers
- Virtual machines everywhere
- Multi-core machine

=>

In-Memory Processing :

- Load your data once
- Use all of your cluster's RAM
- Play with it for as long as you want
- Spill to disk when necessary

What is Spark ?

... And comes **Apache Spark**

- In-memory processing engine

What is Spark ?

... And comes **Apache Spark**

- In-memory processing engine
- Runs on Hadoop (and other platforms)

What is Spark ?

... And comes **Apache Spark**

- In-memory processing engine
- Runs on Hadoop (and other platforms)
- Access data from multiple sources : HDFS,Hive, HBase

What is Spark ?

... And comes **Apache Spark**

- In-memory processing engine
- Runs on Hadoop (and other platforms)
- Access data from multiple sources : HDFS,Hive, HBase
- 100x faster than MR !

What is Spark ?

- Open Source Apache project living on GitHub : 400+ developers

What is Spark ?

- Open Source Apache project living on GitHub : 400+ developers
- Developed at UC Berkeley - first release 2014)

What is Spark ?

- Open Source Apache project living on GitHub : 400+ developers
- Developed at UC Berkeley - first release 2014)
- Current version : 2.0
- Distributed version : 1.6

How to use it ?

- shells :

How to use it ?

- shells :
  - spark-shell for scala

How to use it ?

- shells :
  - spark-shell for scala
  - pyspark for python

How to use it ?

- shells :
  - spark-shell for scala
  - pyspark for python
- Submit a jar/python file through spark-submit

How to use it ?

- shells :
  - spark-shell for scala
  - pyspark for python
- Submit a jar/python file through spark-submit
- Web UI : Zeppelin (will be covered in December)

Core principles

- Data stored in RDD for every different processing (Resilient Distributed Dataset)

Core principles

- Data stored in RDD for every different processing (Resilient Distributed Dataset)
- Apply two types of operations :

Core principles

- Data stored in RDD for every different processing (Resilient Distributed Dataset)
- Apply two types of operations :

    Transformations

Core principles

- Data stored in RDD for every different processing (Resilient Distributed Dataset)
- Apply two types of operations :
  - Transformations
  - Actions

Core principles

- Data stored in RDD for every different processing (Resilient Distributed Dataset)
- Apply two types of operations :
  - Transformations
  - Actions
- Lazy evaluation

What is an RDD ?

- Immutable collection of objects

What is an RDD ?

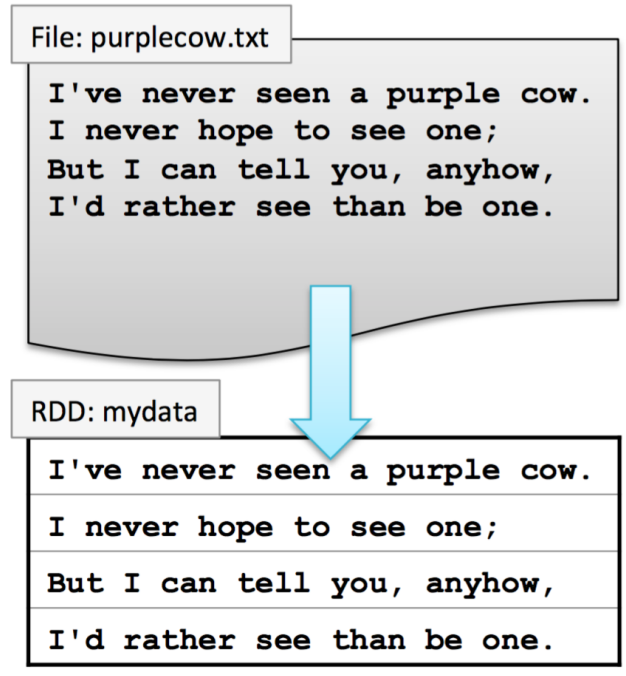
- Immutable collection of objects
- Resilient : lost data can be recomputed

What is an RDD ?

- Immutable collection of objects
- Resilient : lost data can be recomputed
- Partitioned & distributed on a cluster

Creating an RDD : From a file or sequence of files

```
> val mydata = sc.textFile("purplecow.txt")
…
15/01/29 06:20:37 INFO storage.MemoryStore:
   Block broadcast_0 stored as values to
   memory (estimated size 151.4 KB, free 296.8
   MB)

> mydata.count()

…
15/01/29 06:27:37 INFO spark.SparkContext: Job
   finished: take at <stdin>:1, took
   0.160482078 s
4
```



RDD Supported type

RDDs can hold any type of element :

- Primitive types : integers, characters, booleans, etc.
- Sequence types : strings, lists, arrays, tuples, dicts, etc.
- Scala/Java Objects (if serializable)

RDD Creation from files

For file"based RDDs, use SparkContext.textFile :

- sc.textFile("myfile.txt")
- sc.textFile("mydata/*.log")
- sc.textFile("myfile1.txt,myfile2.txt")

textFile only works with line delimited text files : each line in the file(s) is a separate record in the RDD

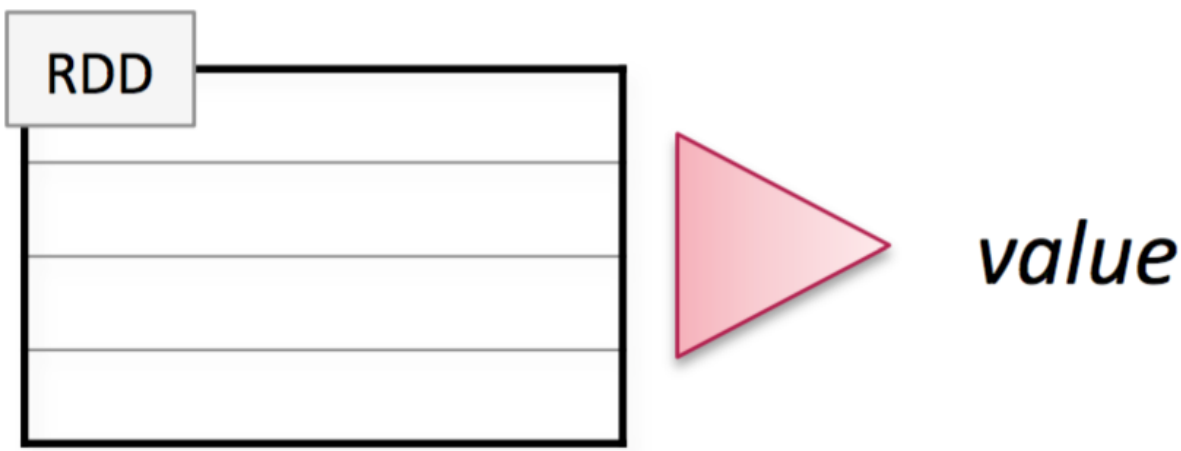
Files are referenced by absolute URI :

- hdfs ://localhost/loudacre/myfile.txt
- file :/home/training/myfile.txt

RDD operations

Actions : return a value from a RDD

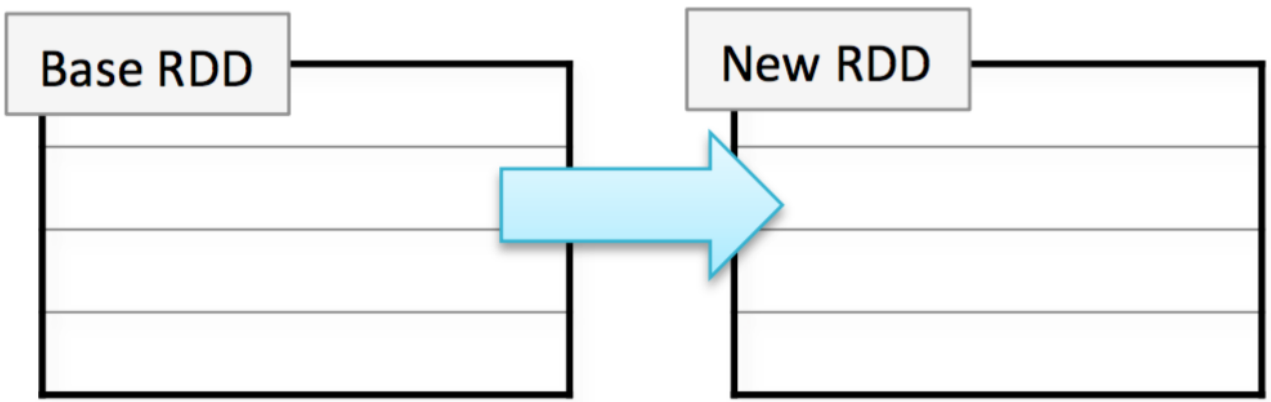
- count() : return the number of elements
- take(n) : return an array of the first n elements
- collect() : return an array of all elements



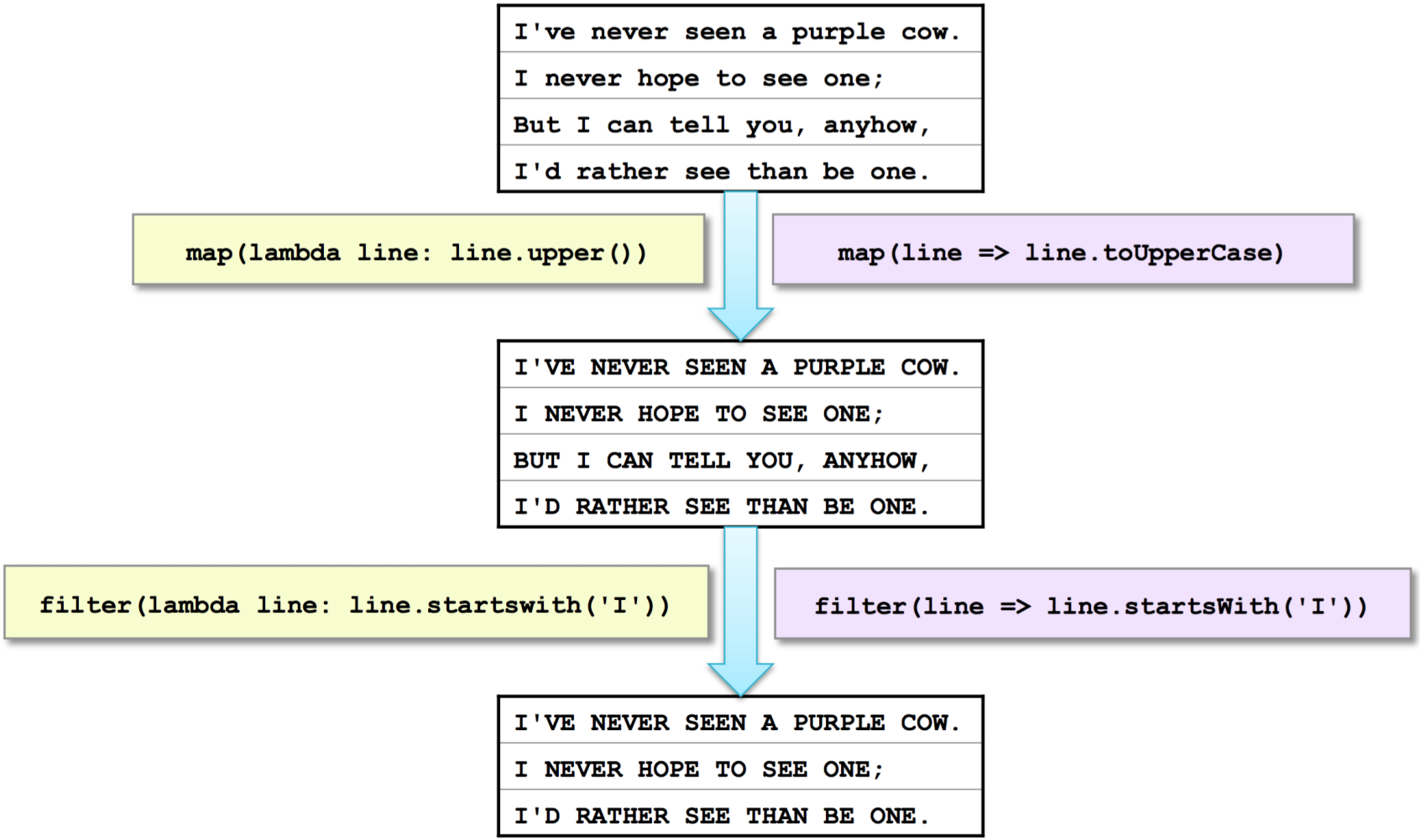
RDD operations

Transformations : create a new RDD from an existing one

- map(function) : creates a new RDD by performing a function on each record in the base RDD
- filter(function) : creates a new RDD by including or excluding each record in the base RDD according to a boolean function



Transformation examples



Lazy evaluation

- **Data in RDDs is not processed until an *action* is performed**

File: purplecow.txt

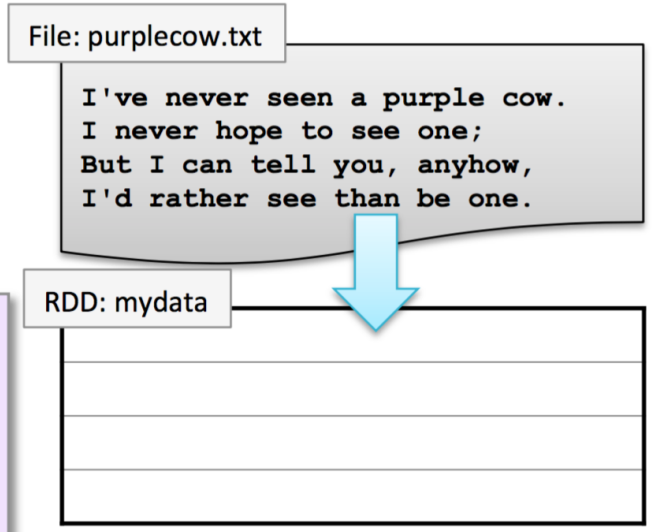
```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

```
>
```

Lazy evaluation

- **Data in RDDs is not processed until an *action* is performed**

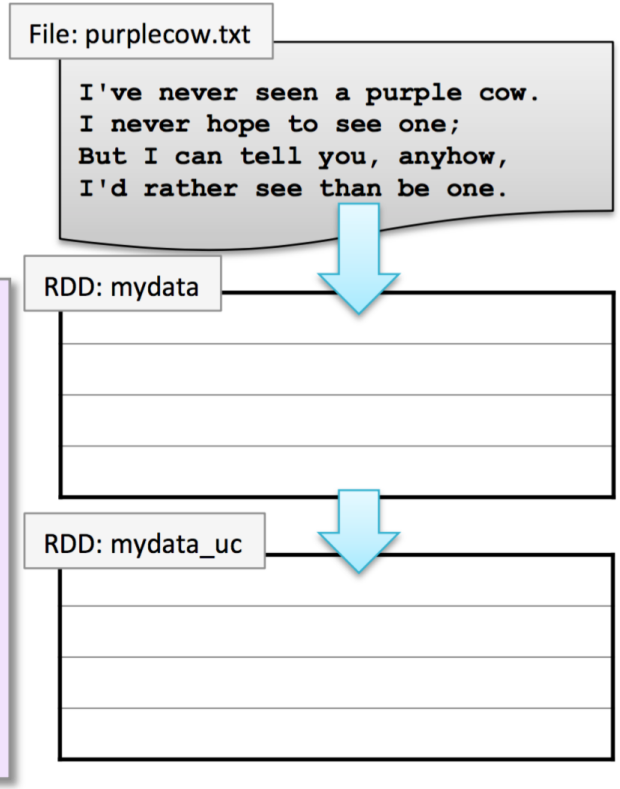
```
> val mydata = sc.textFile("purplecow.txt")
```



Lazy evaluation

- **Data in RDDs is not processed until an *action* is performed**

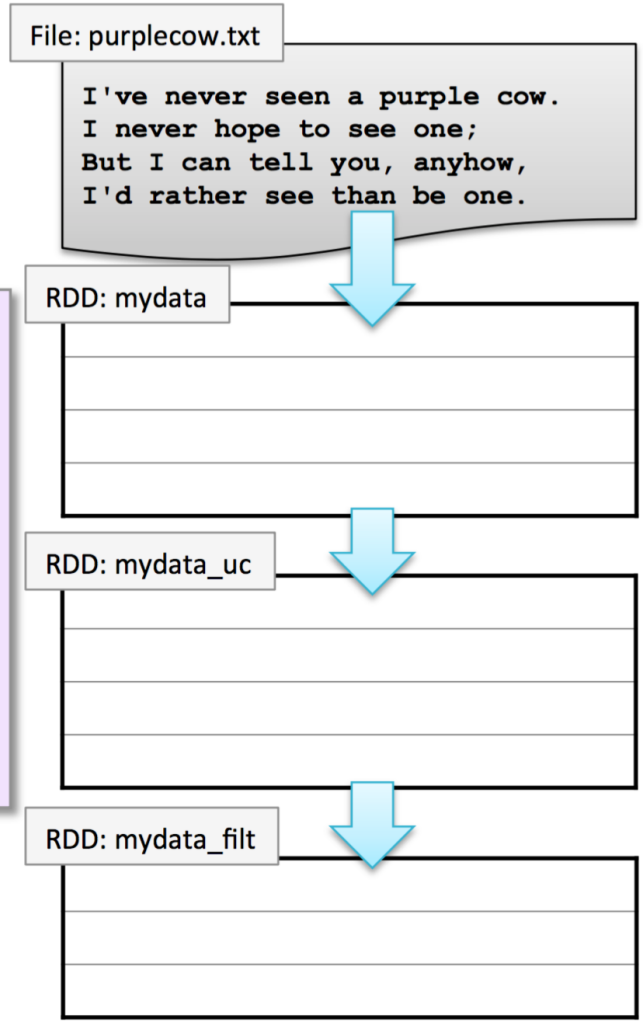
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
```



Lazy evaluation

- **Data in RDDs is not processed until an *action* is performed**

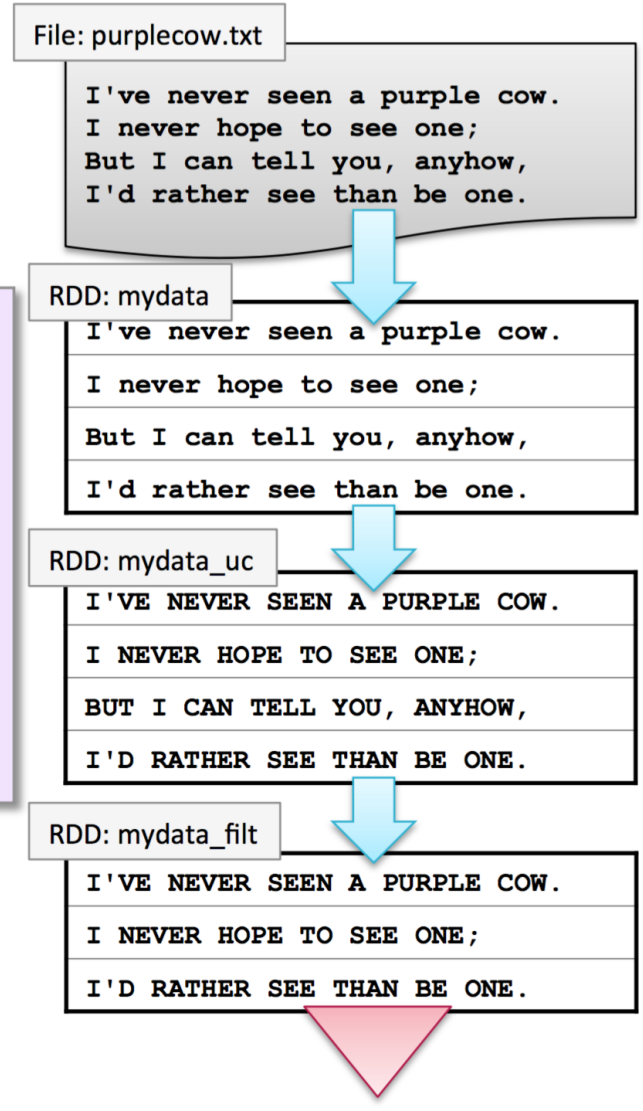
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
```



Lazy evaluation

- **Data in RDDs is not processed until an *action* is performed**

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.count()
3
```



Transformation may be chained

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line => line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line => line.startsWith("I"))
> mydata_filt.count()
3
```

is exactly equivalent to

```
> sc.textFile("purplecow.txt").map(line => line.toUpperCase()).
   filter(line => line.startsWith("I")).count()
3
```

Functionnal programming in Spark

Spark depends heavily on the concepts of functional programming :

- Functions are the fundamental unit of programming
- Functions have input and output only

Key concepts :

- Passing functions as input to other functions
- Anonymous functions

Passing functions as input to other functions

Many RDD operations take functions as parameters
map() for example :

```
> def toUpper(s: String): String =
    { s.toUpperCase }
> val mydata = sc.textFile("purplecow.txt")
> mydata.map(toUpper).take(2)
```

Anonymous functions

Functions defined in-line without an identifier

In Scala : => ...

In Python : lambda x : ...

```
> mydata.map(line => line.toUpperCase()).take(2)
```

OR

```
> mydata.map(_.toUpperCase()).take(2)
```

Scala allows anonymous parameters using underscore (_)

Let's do it !

Please refer to the TP-SPARK-CORE. Your VM :

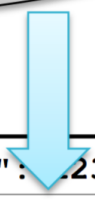
- Log in as user **training** (password **training**)
- Preinstalled and configured with Spark, CDH, some tools as Maven, Intellij, Emacs

Whole File-Based RDDs

- **`sc.wholeTextFiles(`*`directory`*`)`**
  - Maps entire contents of each file in a directory to a single RDD element
  - Works only for small files (element must fit in memory)

file2.json

```
{
 "firstName":"Barney",
 "lastName":"Rubble",
 "userid":"234"
}
```



(file1.json,{"firstName":"Fred","lastName":"Flintstone","userid":"123"})
(file2.json,{"firstName":"Barney","lastName":"Rubble","userid":"123"})
(file3.xml,…)
(file4.xml,…)

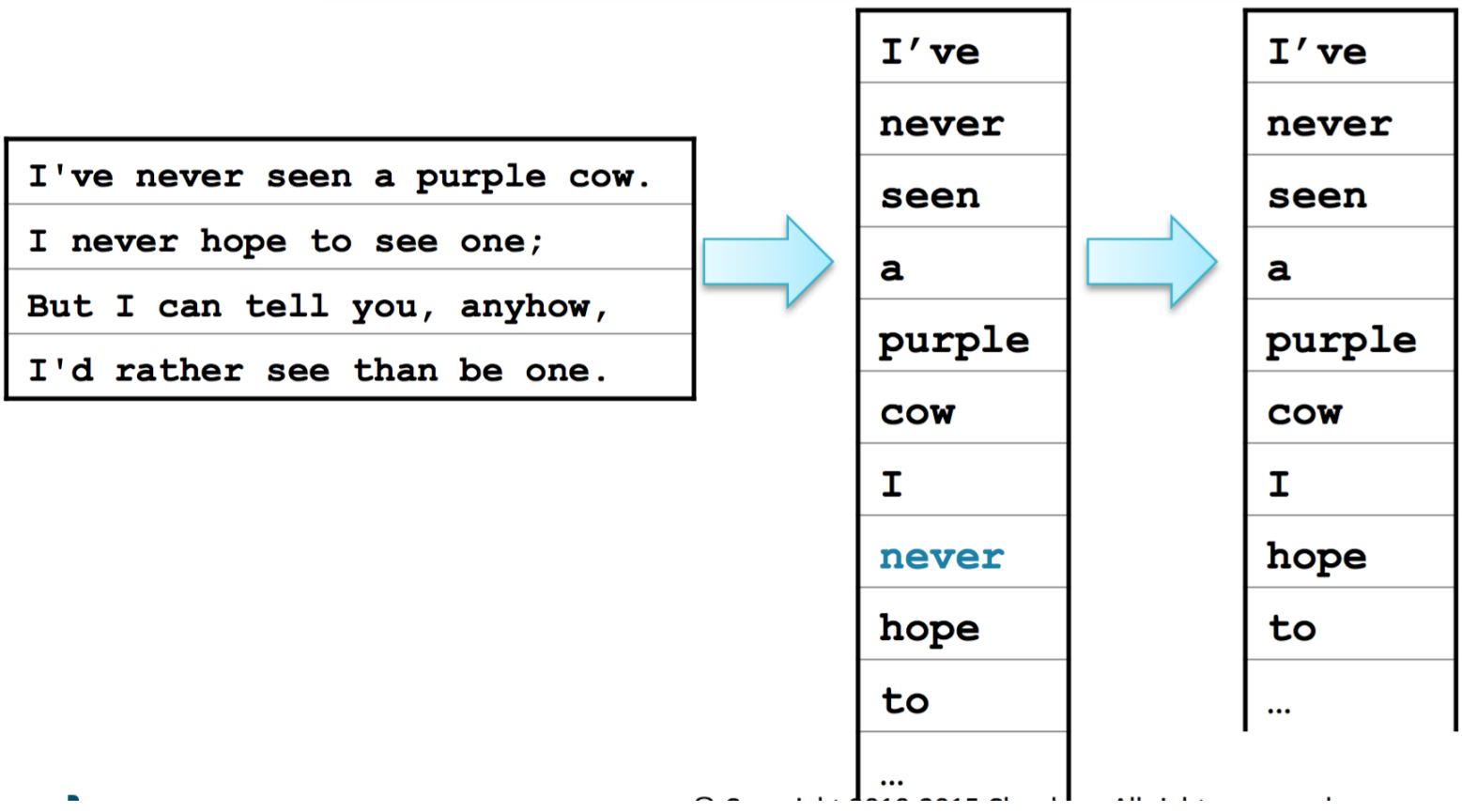
Some other operations

- **Single-RDD Transformations**
  - **`flatMap`** – maps one element in the base RDD to multiple elements
  - **`distinct`** – filter out duplicates
  - **`sortBy`** – use provided function to sort
- **Multi-RDD Transformations**
  - **`intersection`** – create a new RDD with all elements in both original RDDs
  - **`union`** – add all elements of two RDDs into a single new RDD
  - **`zip`** – pair each element of the first RDD with the corresponding element of the second

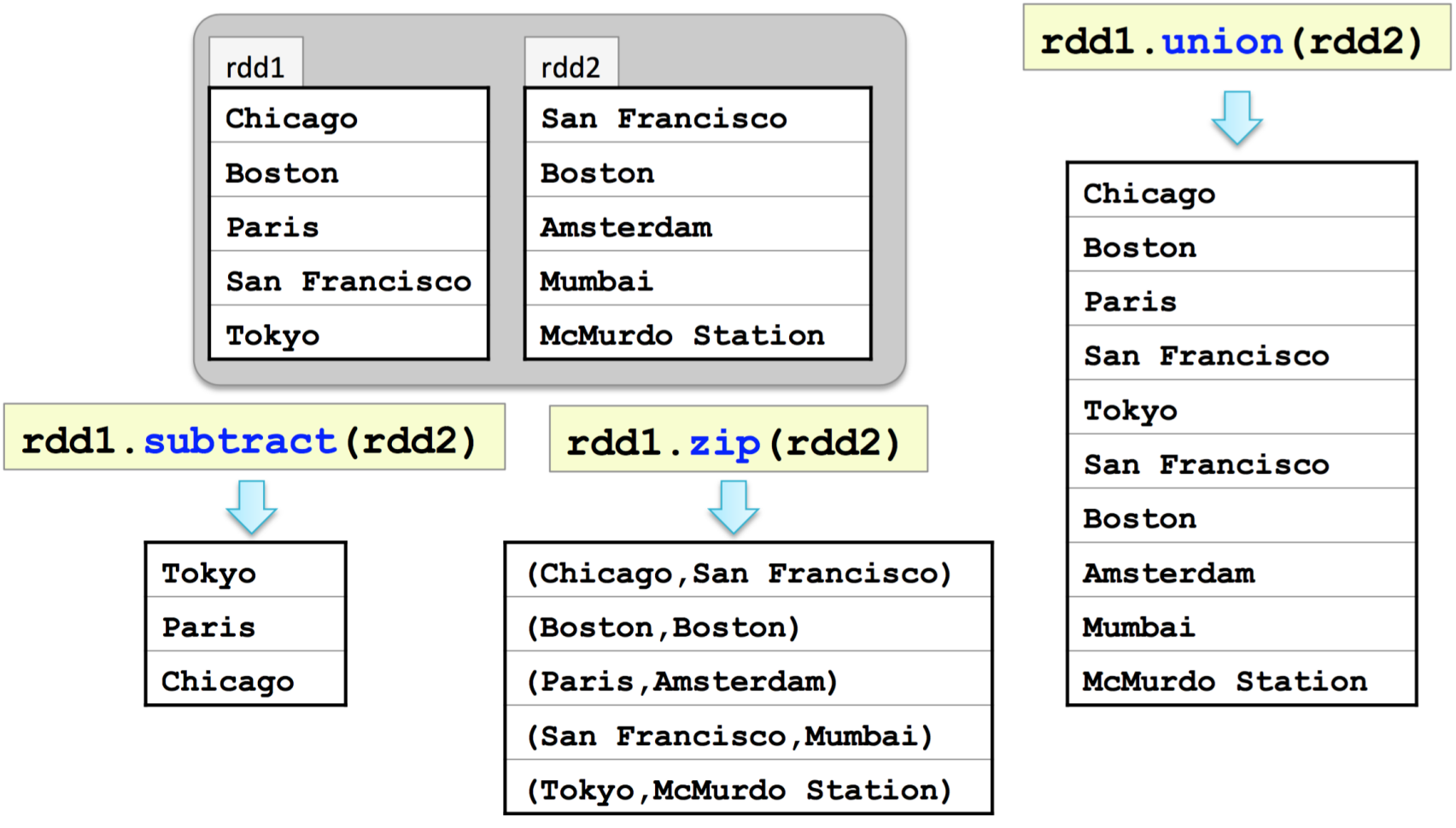
Flatmap() and distinct()

Scala

```
> sc.textFile(file).
  flatMap(line => line.split(' ')).
  distinct()
```



MultiRDD transformation



MultiRDD transformation

- **Other RDD operations**
  - **`first`** – return the first element of the RDD
  - **`foreach`** – apply a function to each element in an RDD
  - **`top(n)`** – return the largest *n* elements using natural ordering
- **Sampling operations**
  - **`sample`** – create a new RDD with a sampling of elements
  - **`takeSample`** – return an array of sampled elements
- **Double RDD operations**
  - Statistical functions, e.g., **`mean`**, **`sum`**, **`variance`**, **`stdev`**

Let's do it !

Please refer to the TP-SPARK-CORE-2.

Key-Value Pair RDD

- **Pair RDDs are a special form of RDD**
  - Each element must be a key-value pair (a two-element tuple)
  - Keys and values can be any type
- **Why?**
  - Use with map-reduce algorithms
  - Many additional functions are available for common data processing needs
    - e.g., sorting, joining, grouping, counting, etc.

Pair RDD

(key1,value1)
(key2,value2)
(key3,value3)
…

Creating PairRDD with map()

Scala

```
> val users = sc.textFile(file) \
    .map(line => line.split('\t')) \
    .map(fields => (fields(0),fields(1)))
```

```
user001\tFred Flintstone
user090\tBugs Bunny
user111\tHarry Potter
…
```

(user001,Fred Flintstone)
(user090,Bugs Bunny)
(user111,Harry Potter)
…

Creating PairRDD with keyBy()

Scala

```
> sc.textFile(logfile) \
    .keyBy(line => line.split(' ')(2))
```

User ID

```
56.38.234.188 - 99788 "GET /KBDOC-00157.html HTTP/1.0" …
56.38.234.188 - 99788 "GET /theme.css HTTP/1.0" …
203.146.17.59 - 25254 "GET /KBDOC-00230.html HTTP/1.0" …
…
```

```
(99788,56.38.234.188 - 99788 "GET /KBDOC-00157.html…)
(99788,56.38.234.188 - 99788 "GET /theme.css…)
(25254,203.146.17.59 - 25254 "GET /KBDOC-00230.html…)
…
```

MapReduce in Spark

- **Map-reduce in Spark works on Pair RDDs**
- **Map phase**
  - Operates on one record at a time
  - "Maps" each record to one or more new records
  - e.g. **`map`**, **`flatMap`**, **`filter`**, **`keyBy`**
- **Reduce phase**
  - Works on map output
  - Consolidates multiple records
  - e.g. **`reduceByKey`**, **`sortByKey`**, **`mean`**

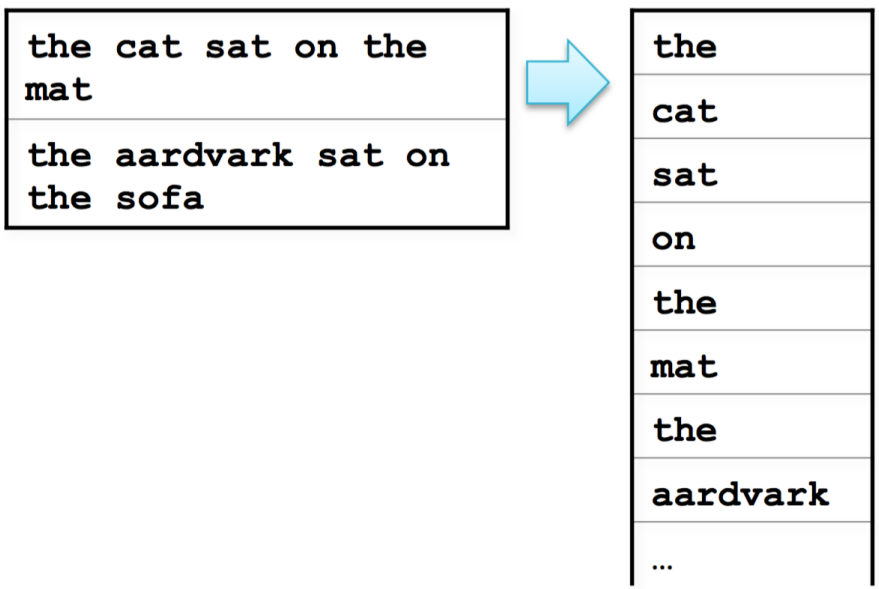
Word Count Example

```
> counts = sc.textFile(file)
```

the cat sat on the mat
the aardvark sat on the sofa

Word Count Example

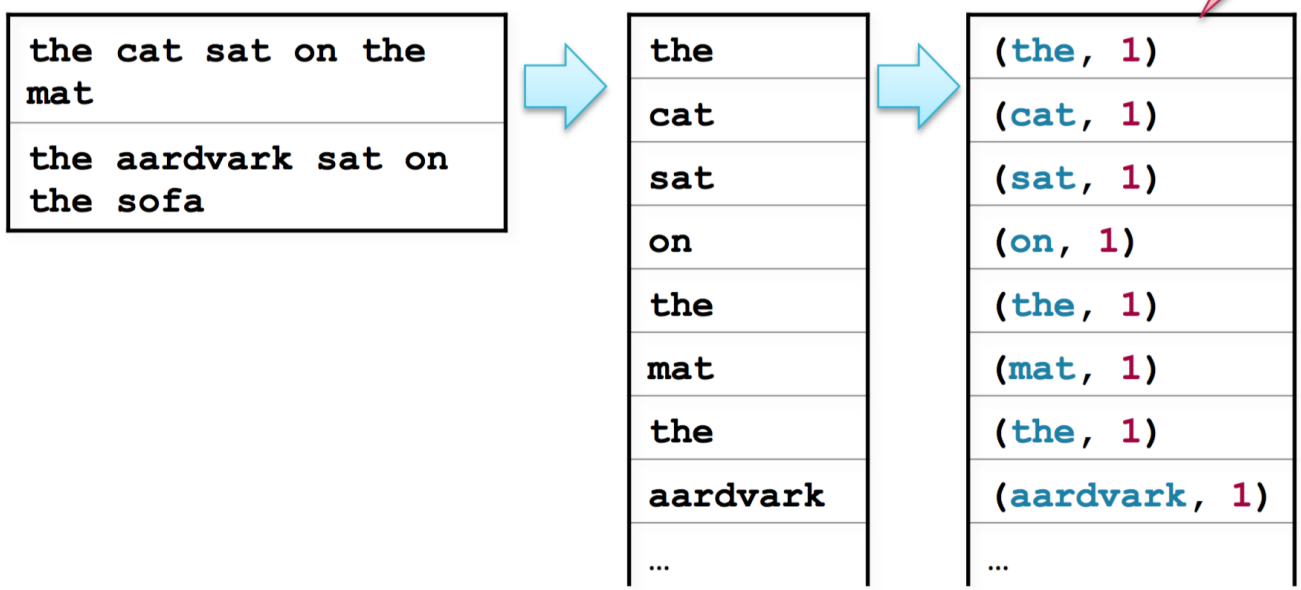
```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split())
```



Word Count Example

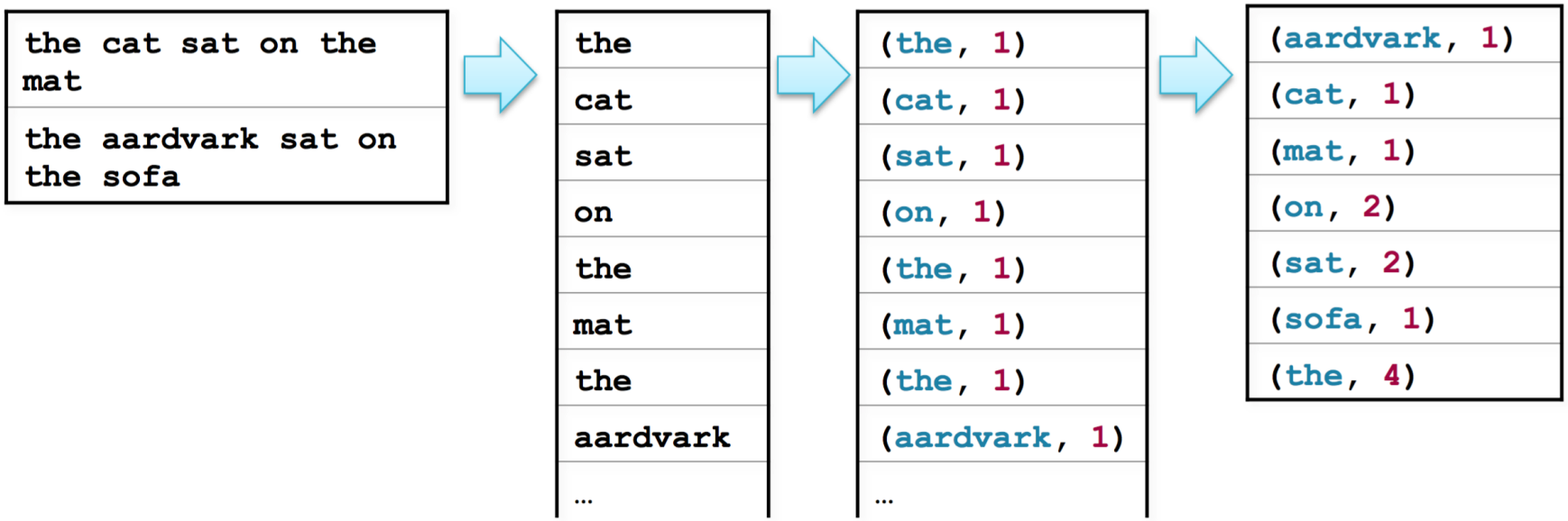
```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1))
```

Key-Value Pairs



Word Count Example

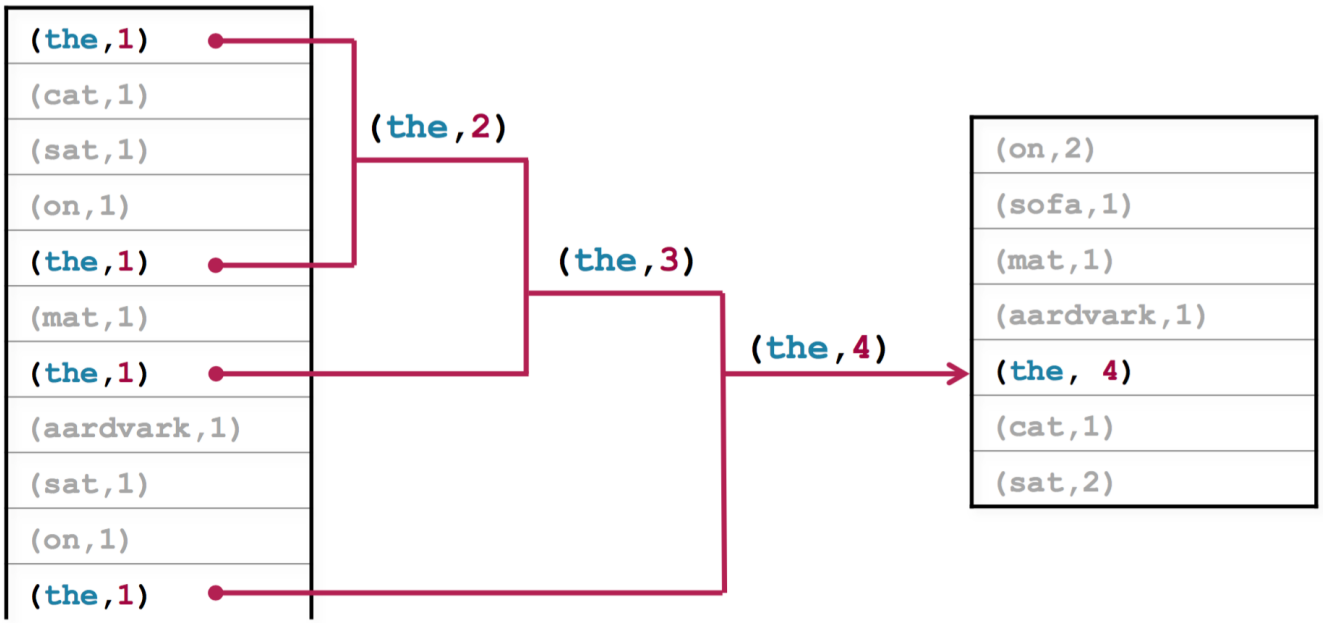
```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1)) \
    .reduceByKey(lambda v1,v2: v1+v2)
```



Word Count Example

- **The function passed to `reduceByKey` combines values from two keys**
  - Function must be binary

```
> counts = sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word,1)) \
  .reduceByKey(lambda v1,v2: v1+v2)
```



Word Count Example

```
> val counts = sc.textFile(file).
  flatMap(line => line.split("\\W")).
  map(word => (word,1)).
  reduceByKey((v1,v2) => v1+v2)
```

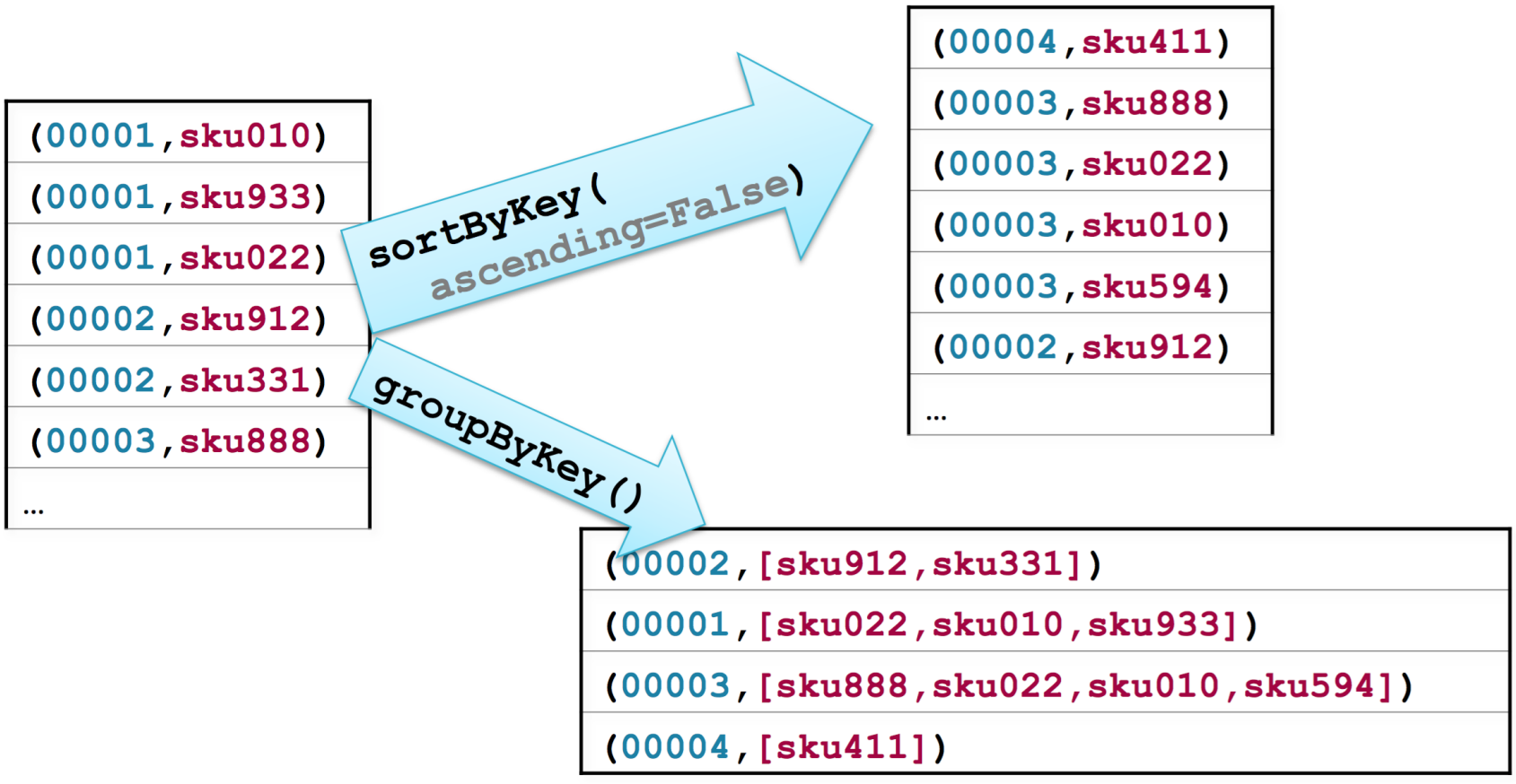
OR

```
> val counts = sc.textFile(file).
  flatMap(_.split("\\W")).
  map((_,1)).
  reduceByKey(_+_)
```

Other Pair RDD application

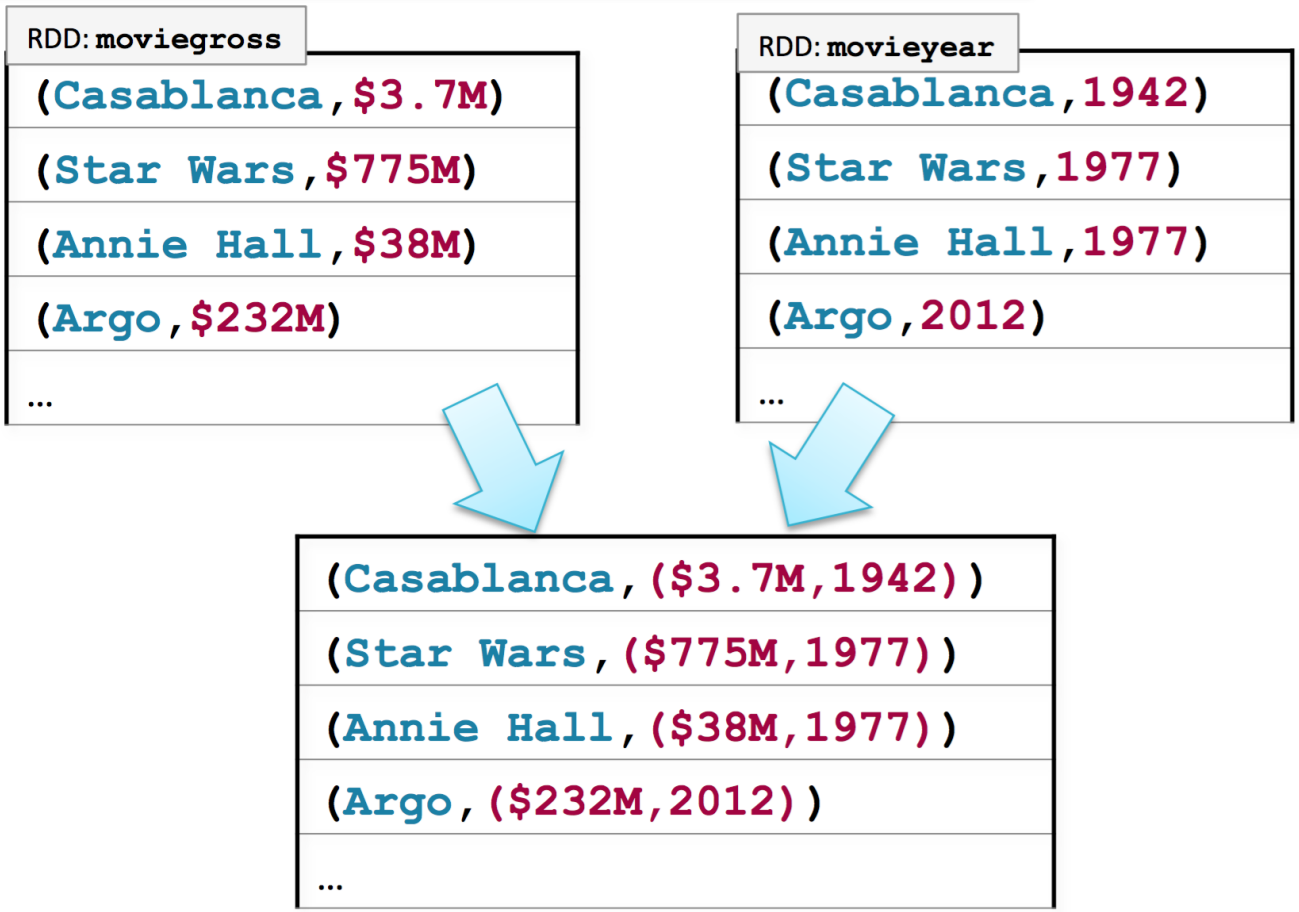
- **In addition to `map` and `reduce` functions, Spark has several operations specific to Pair RDDs**
- **Examples**
  - **`countByKey`** – return a map with the count of occurrences of each key
  - **`groupByKey`** – group all the values for each key in an RDD
  - **`sortByKey`** – sort in ascending or descending order
  - **`join`** – return an RDD containing all pairs with matching keys from two RDDs

sortByKey() and groupByKey() examples



join example

```
> movies = moviegross.join(movieyear)
```



A pattern to join datasets

- **A common programming pattern**
  1. Map separate datasets into key-value Pair RDDs
  2. Join by key
  3. Map joined data into the desired format
  4. Save, display, or continue processing…

PairRDD operations

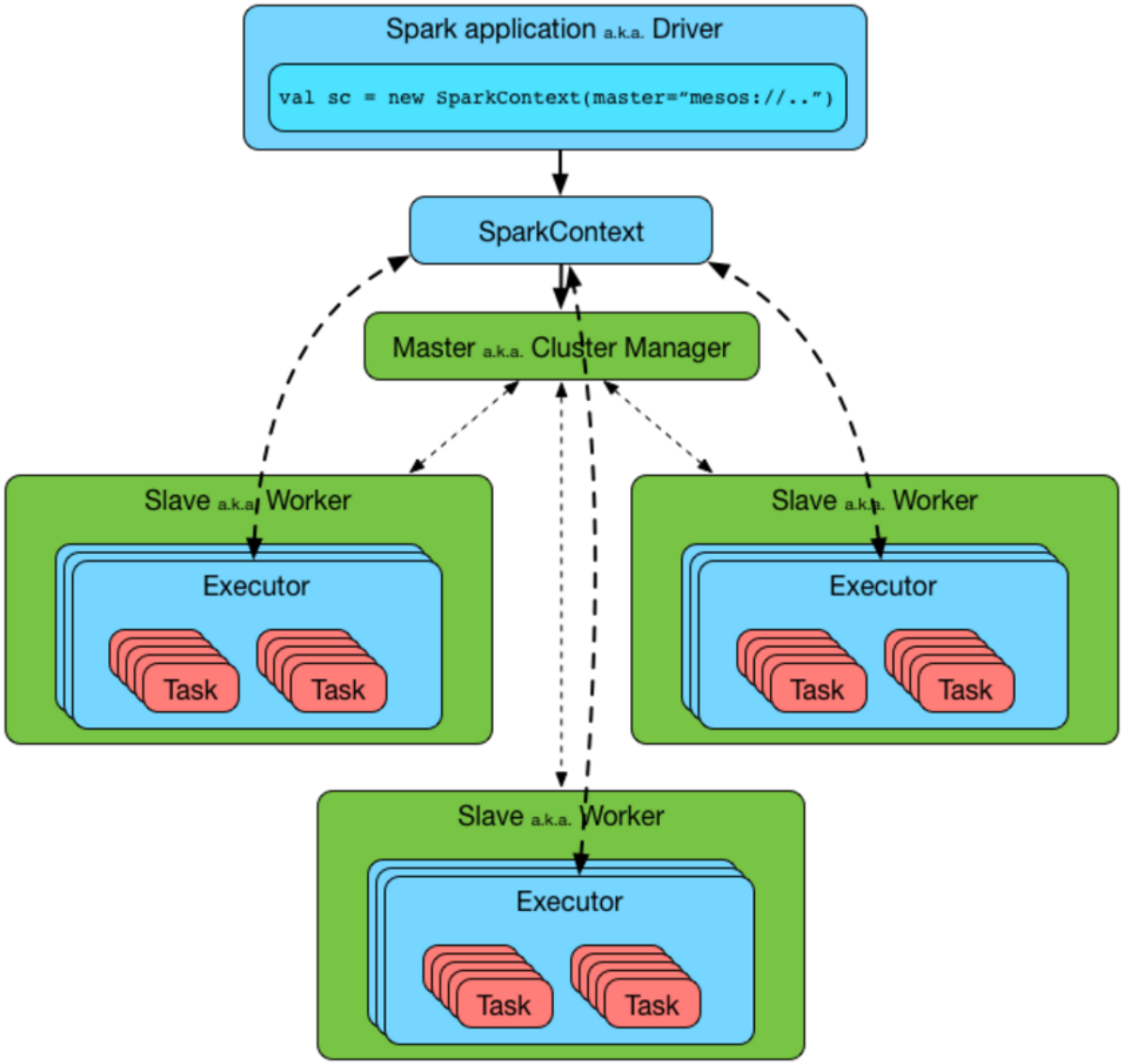
- **Some other pair operations**
  - **`keys`** – return an RDD of just the keys, without the values
  - **`values`** – return an RDD of just the values, without keys
  - **`lookup(`*`key`*`)`** – return the value(s) for a key
  - **`leftOuterJoin`**, **`rightOuterJoin`** , **`fullOuterJoin`** – join, including keys defined in the left, right or either RDD respectively
  - **`mapValues`**, **`flatMapValues`** – execute a function on just the values, keeping the key the same
- **See the `PairRDDFunctions` class Scaladoc for a full list**

Let's do it !

Please refer to the TP-SPARK-CORE-3.

Distributed Data Processing with Spark

Spark Architecture



Spark Architecture

Spark Application = 1 driver program and executors

Driver Program = The process running the main() function of the application and creating the SparkContext

Cluster Manager = allocate resources across applications

Worker = Any node that can run code in the cluster

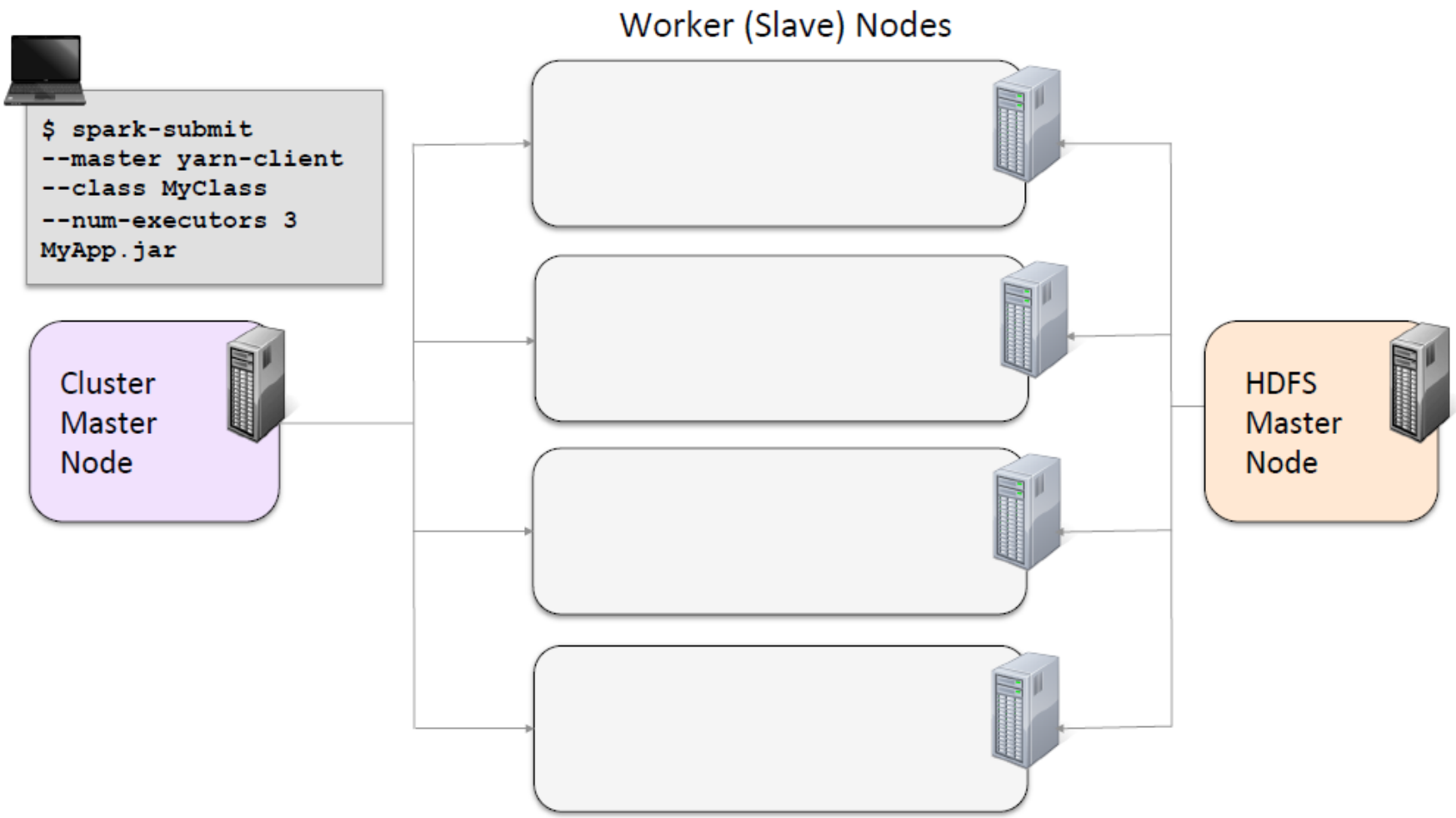
Executor = A process launched on a worker node

Task A unit of work that will be sent to one executor

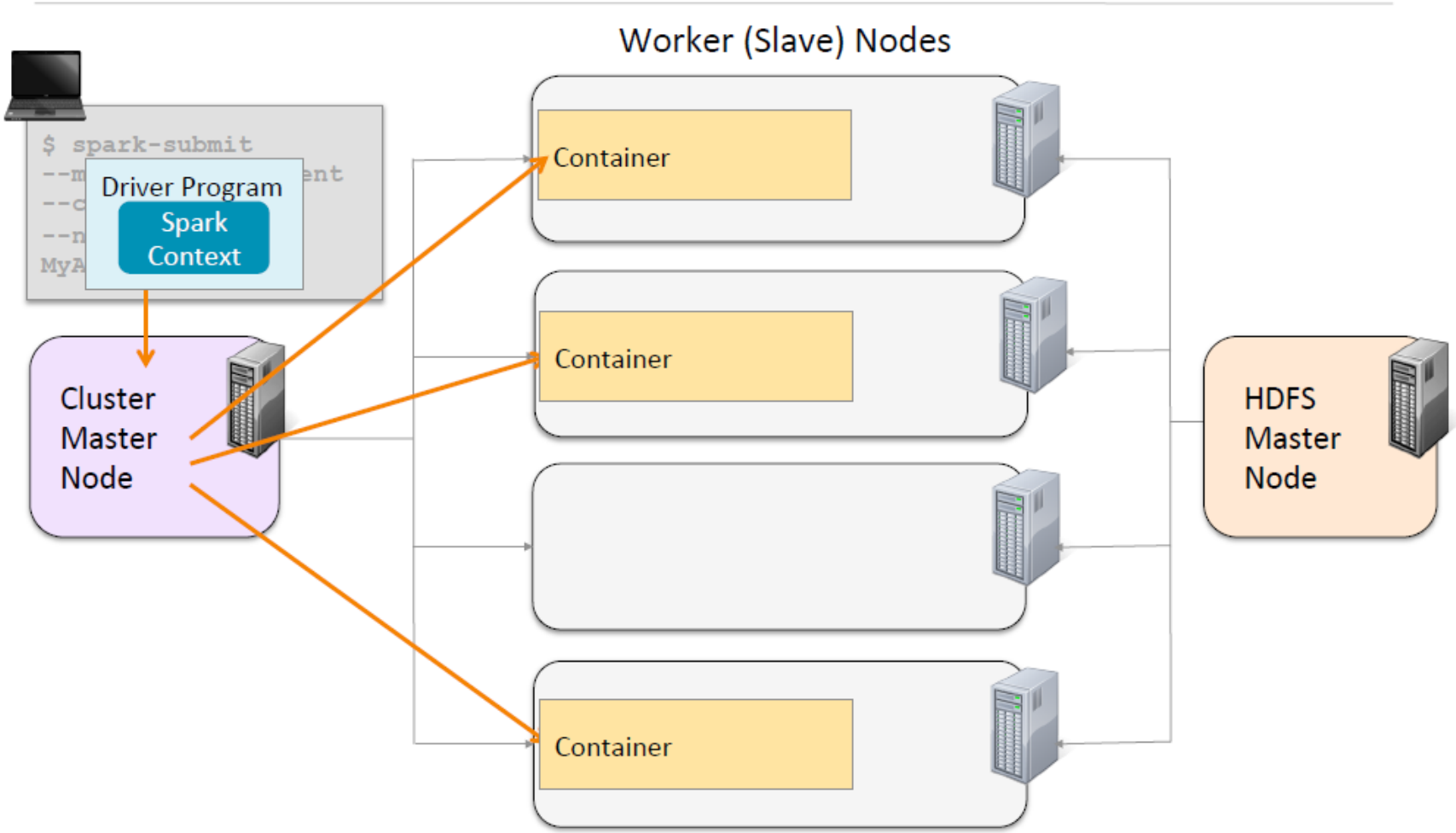
Spark Deployment Modes

	Local	Spark Standalone	YARN Client	YARN Cluster
Driver runs in:	Client	Client	Client	Resource Manager
Cluster Manager	Client	Spark Master	Resource Manager	Resource Manager
Who start executors ?	Client	Spark Slave	Node Manager	Node Manager
Spark Shell ?	Yes	Yes	Yes	No

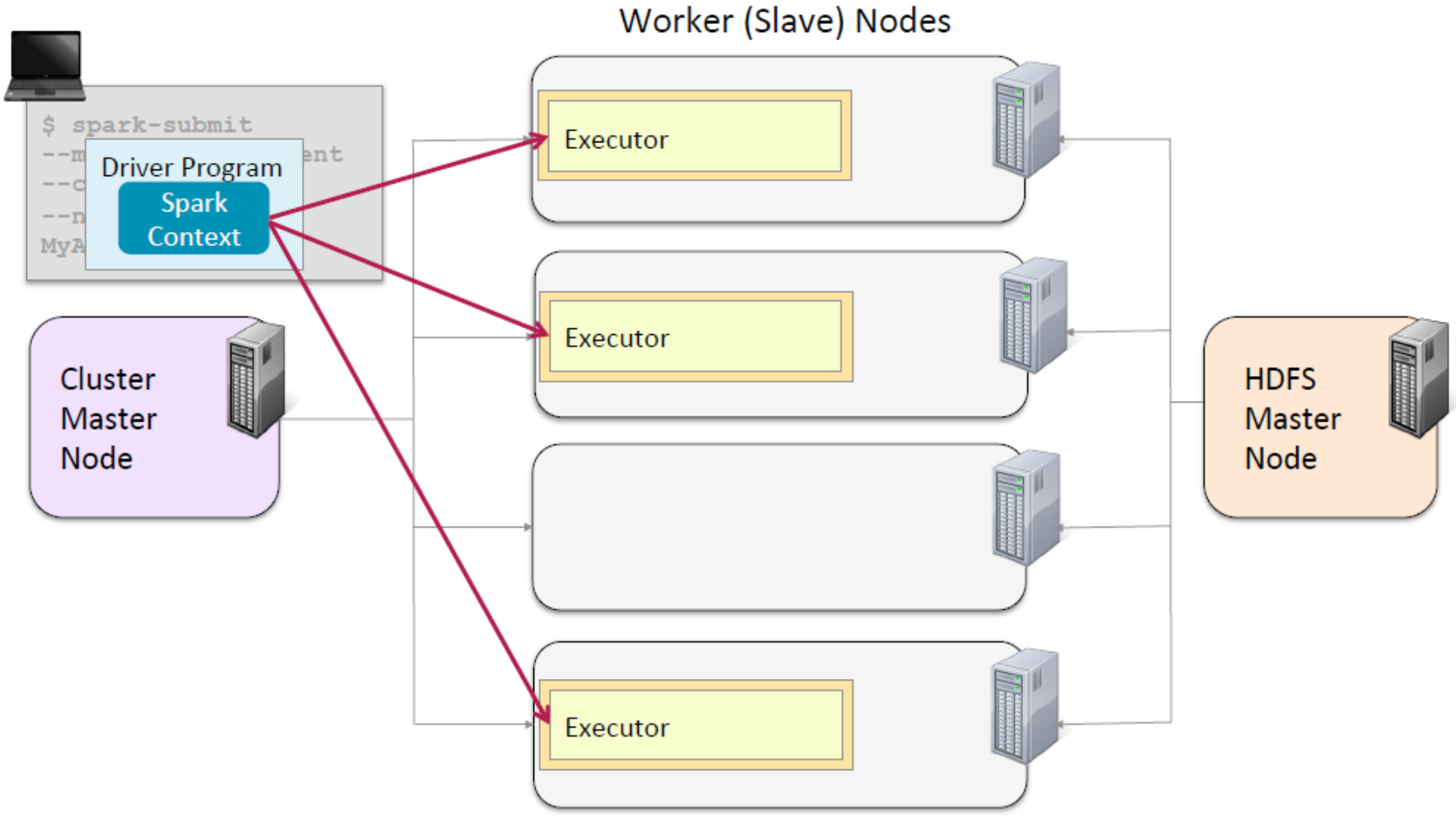
Instance


Worker (Slave) Nodes
$ spark-submit
--master yarn-client
--class MyClass
--num-executors 3
MyApp.jar
Cluster
Master
Node
HDFS
Master
Node

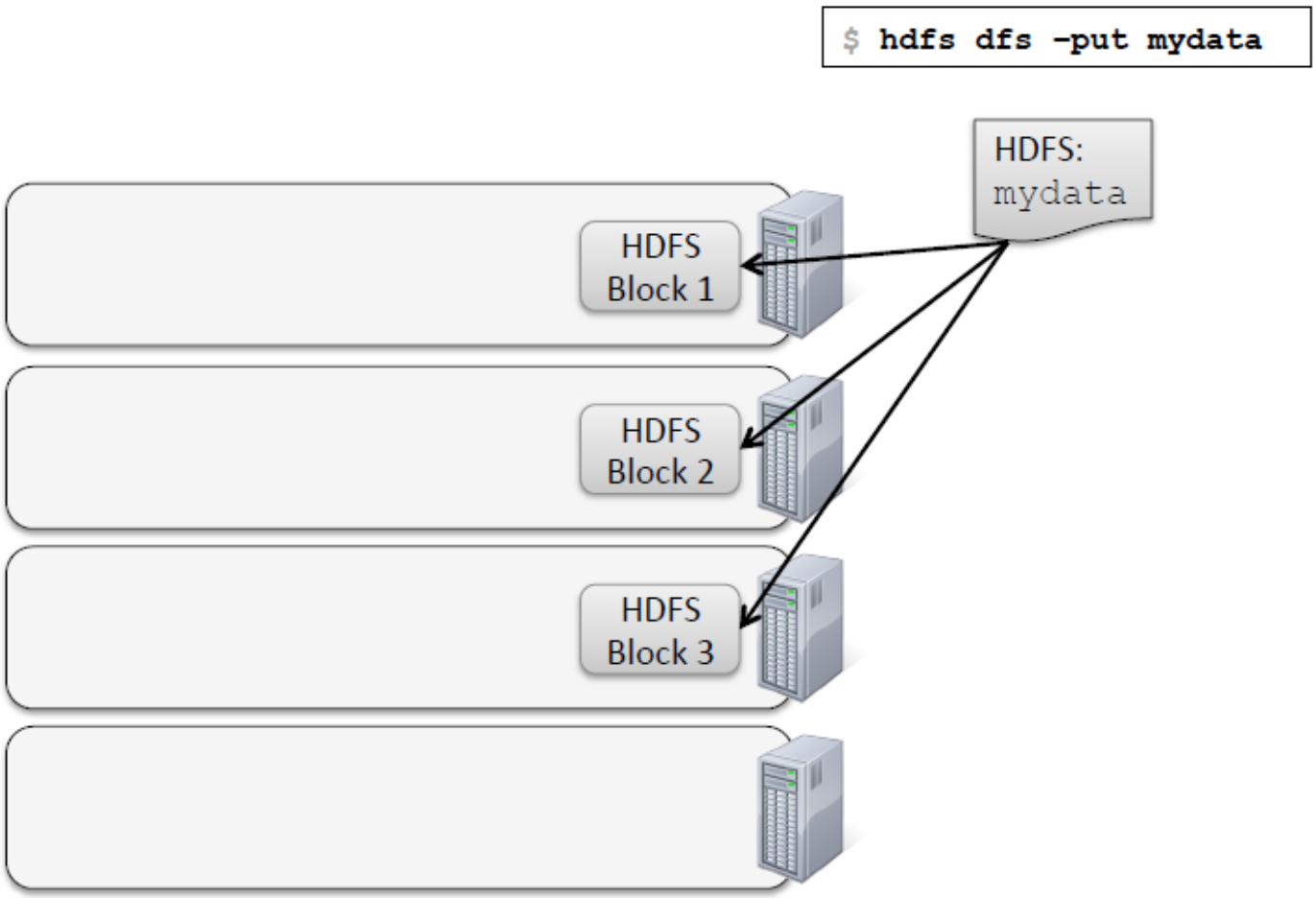
Instance


Worker (Slave) Nodes
$ spark-submit
Driver Program
Spark Context
Cluster Master Node
Container
Container
Container
HDFS Master Node

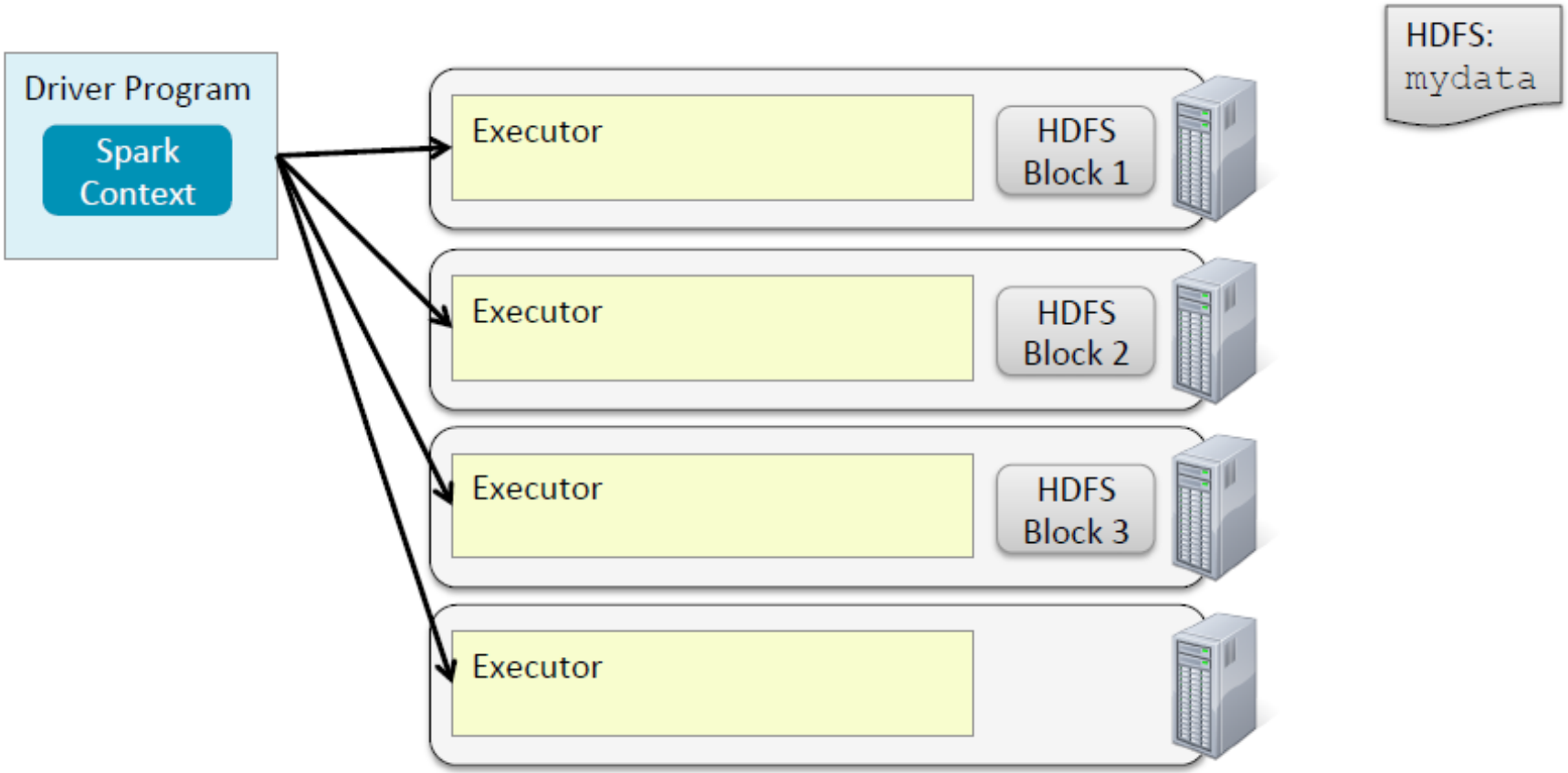
Instance


Worker (Slave) Nodes
$ spark-submit
Driver Program
Spark Context
Executor
Executor
Executor
Cluster Master Node
HDFS Master Node

Spark and HDFS locality



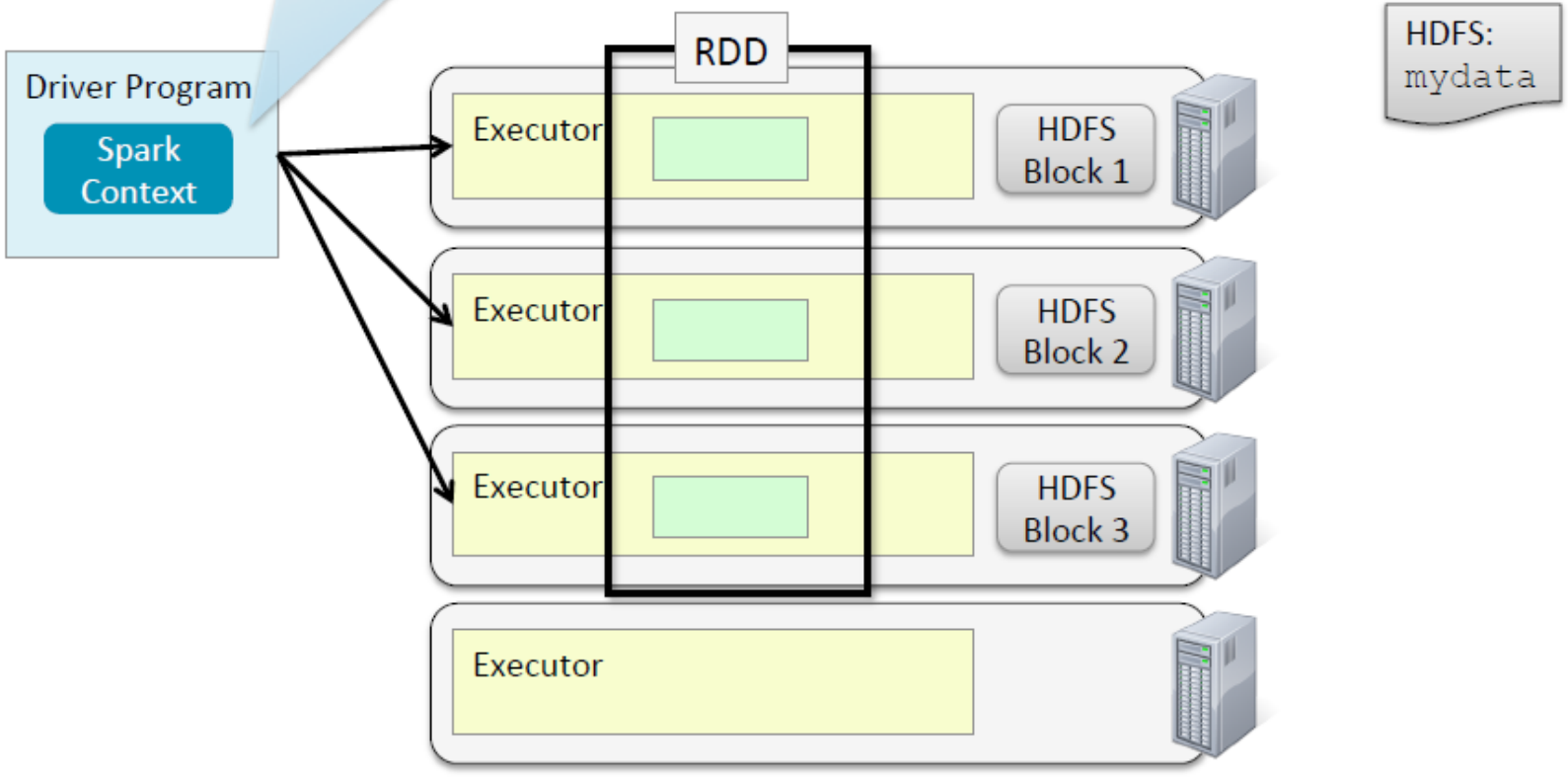
Spark and HDFS locality



Spark and HDFS locality

```
sc.textFile("hdfs://…mydata").collect()
```

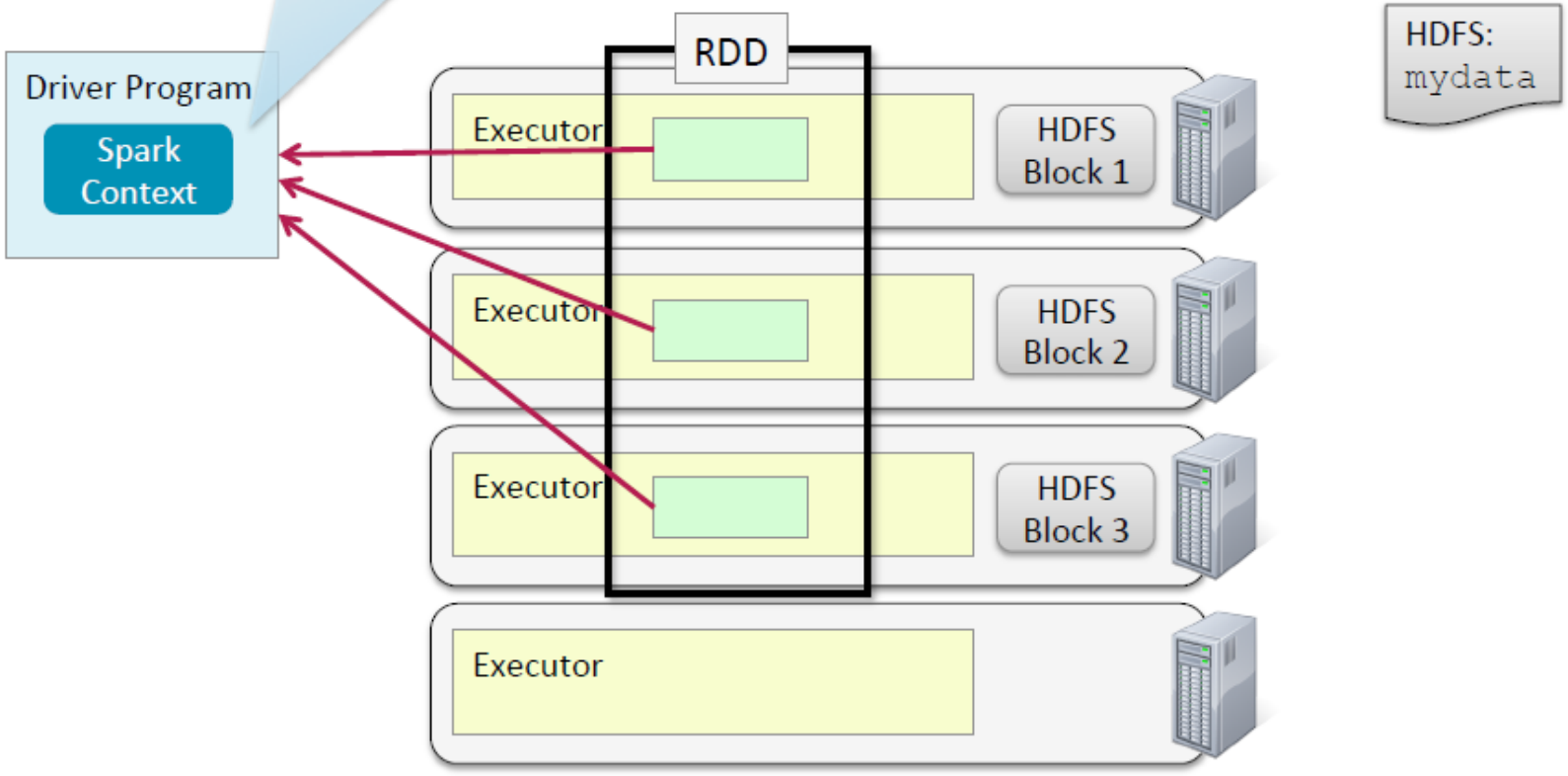
By default, Spark partitions file-based RDDs by block. Each block loads into a single partition.



Spark and HDFS locality

```
sc.textFile("hdfs://…mydata").collect()
```

Data is distributed across executors until an action returns a value to the driver

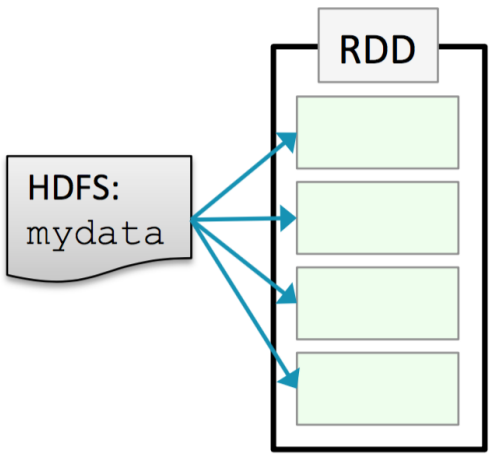


Parallel operations on partitions

- **RDD operations are executed in parallel on each partition**
  - When possible, tasks execute on the worker nodes where the data is in memory
- **Some operations preserve partitioning**
  - e.g., **`map`**, **`flatMap`**, **`filter`**
- **Some operations repartition**
  - e.g., **`reduce`**, **`sort`**, **`group`**

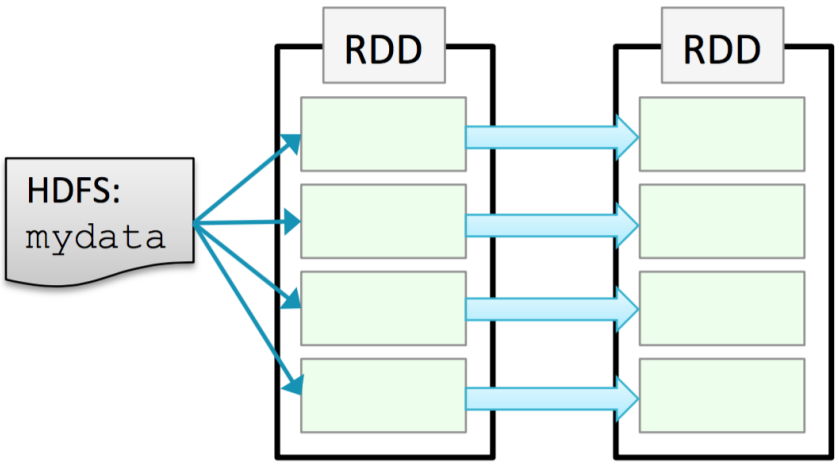
Parallel operations on partitions

```
> avglens = sc.textFile(file)
```



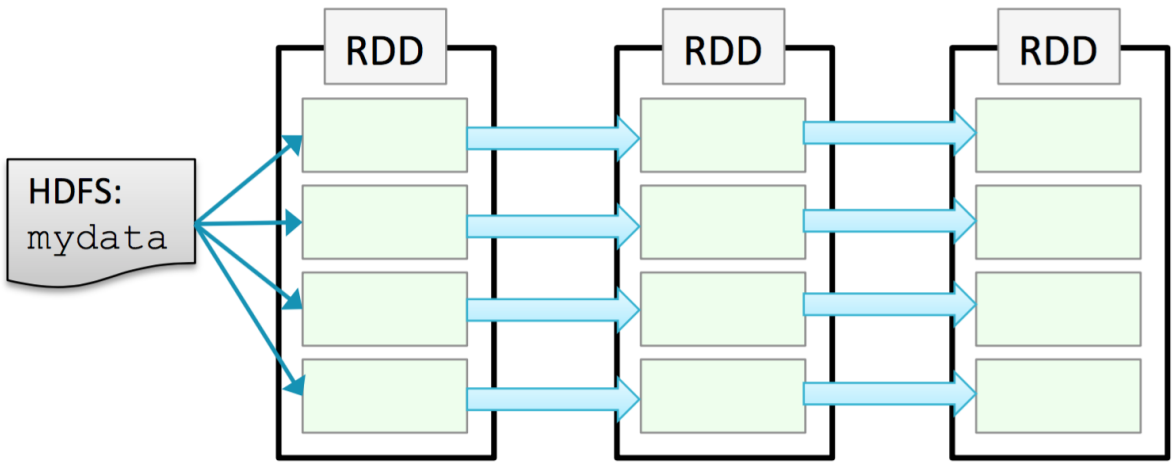
Parallel operations on partitions

```
> avglens = sc.textFile(file) \
  .flatMap(lambda line: line.split())
```



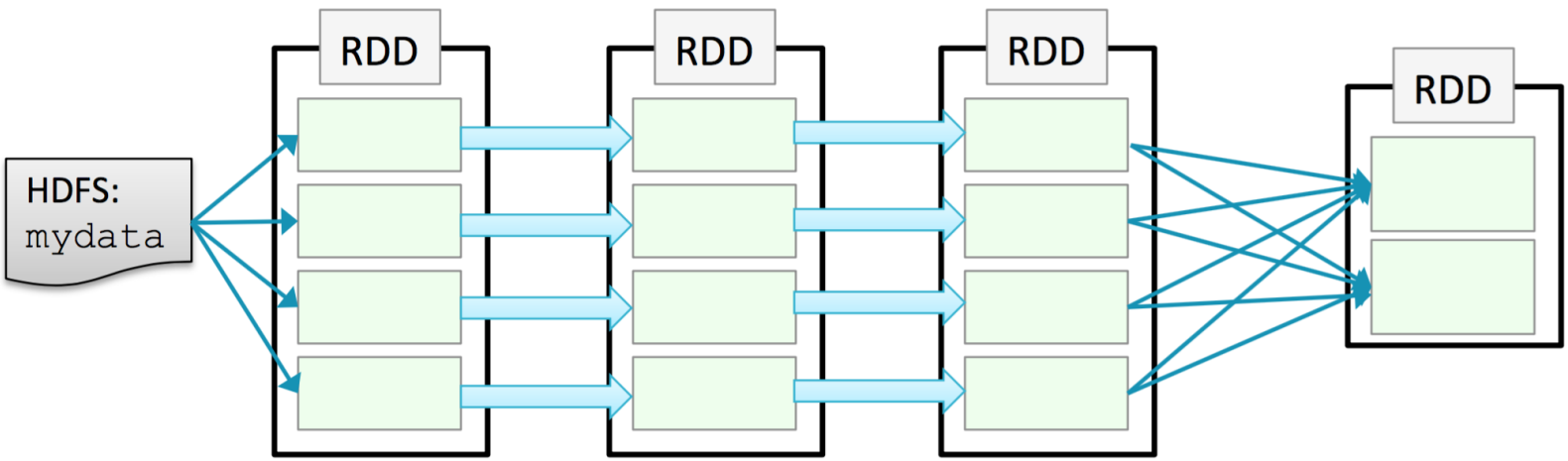
Parallel operations on partitions

```
> avglens = sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word[0],len(word)))
```



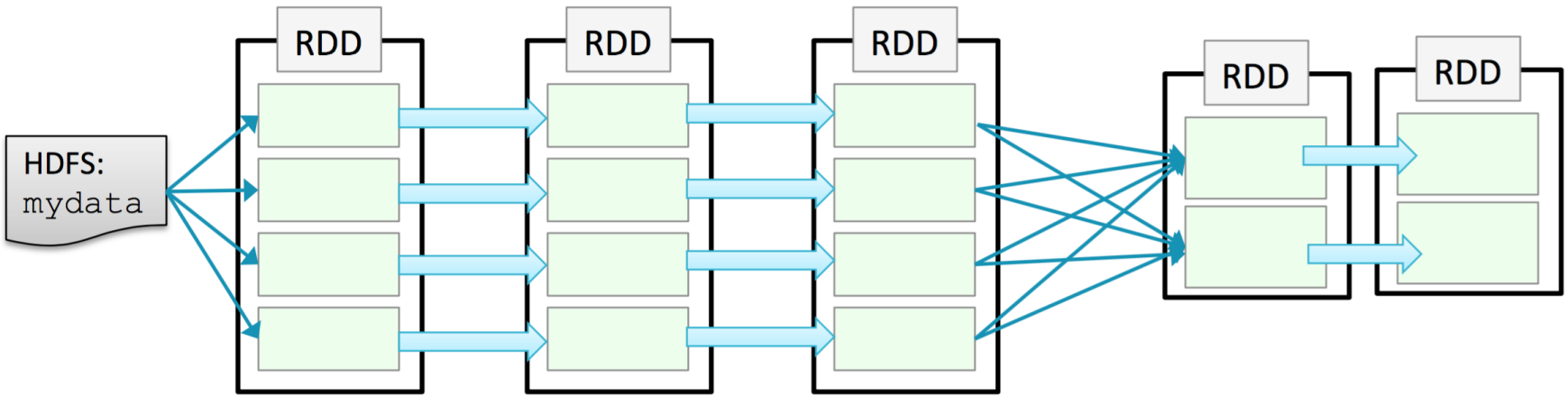
Parallel operations on partitions

```
> avglens = sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word[0],len(word))) \
  .groupByKey()
```



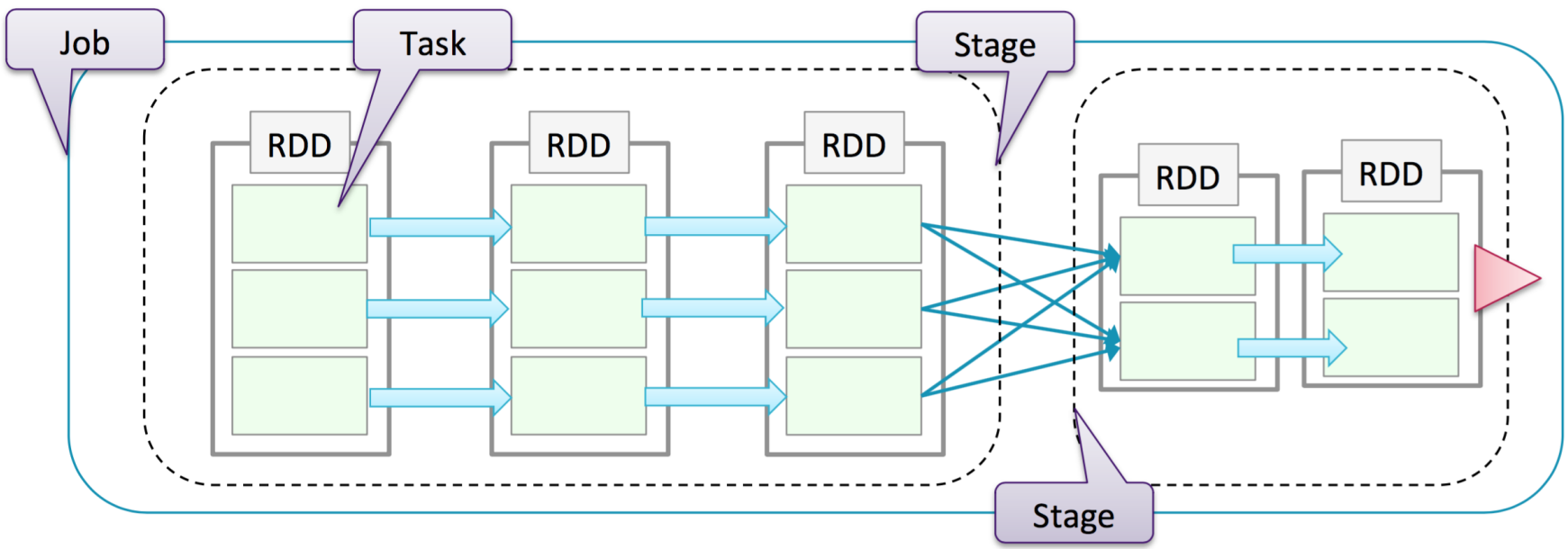
Parallel operations on partitions

```
> avglens = sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word[0],len(word))) \
  .groupByKey() \
  .map(lambda (k, values): \
     (k, sum(values)/len(values)))
```



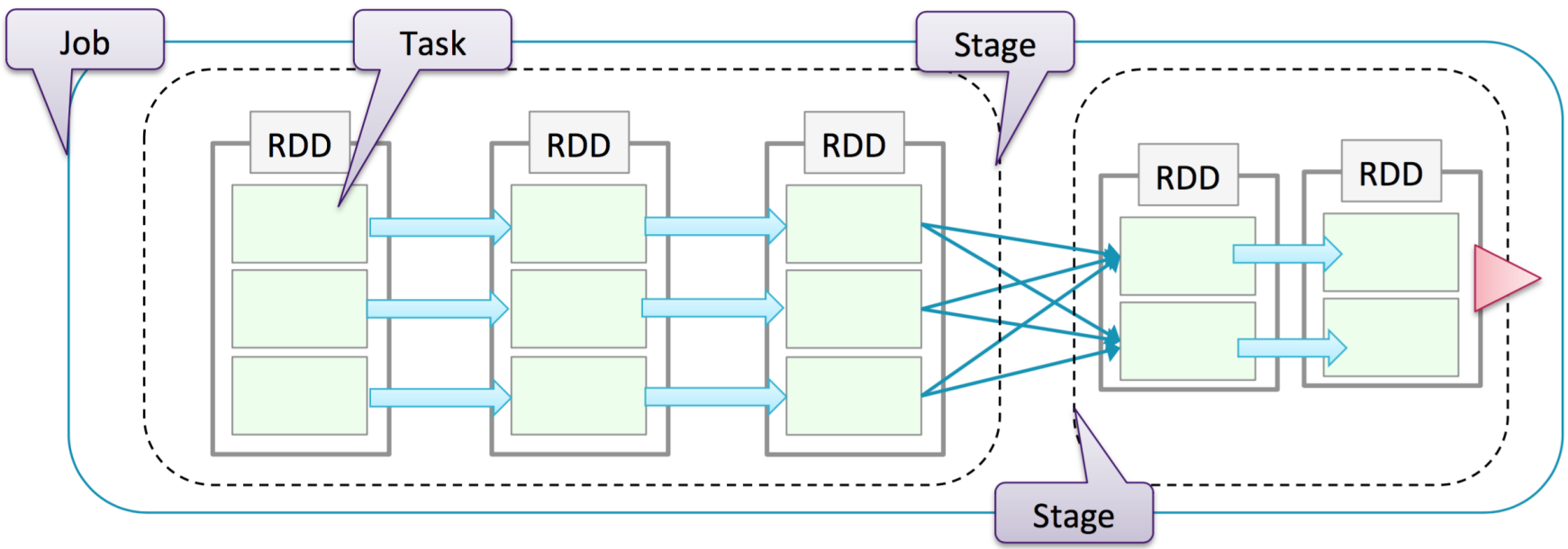
Spark Terminology

- **Job** : a set of tasks executed as a result of an action



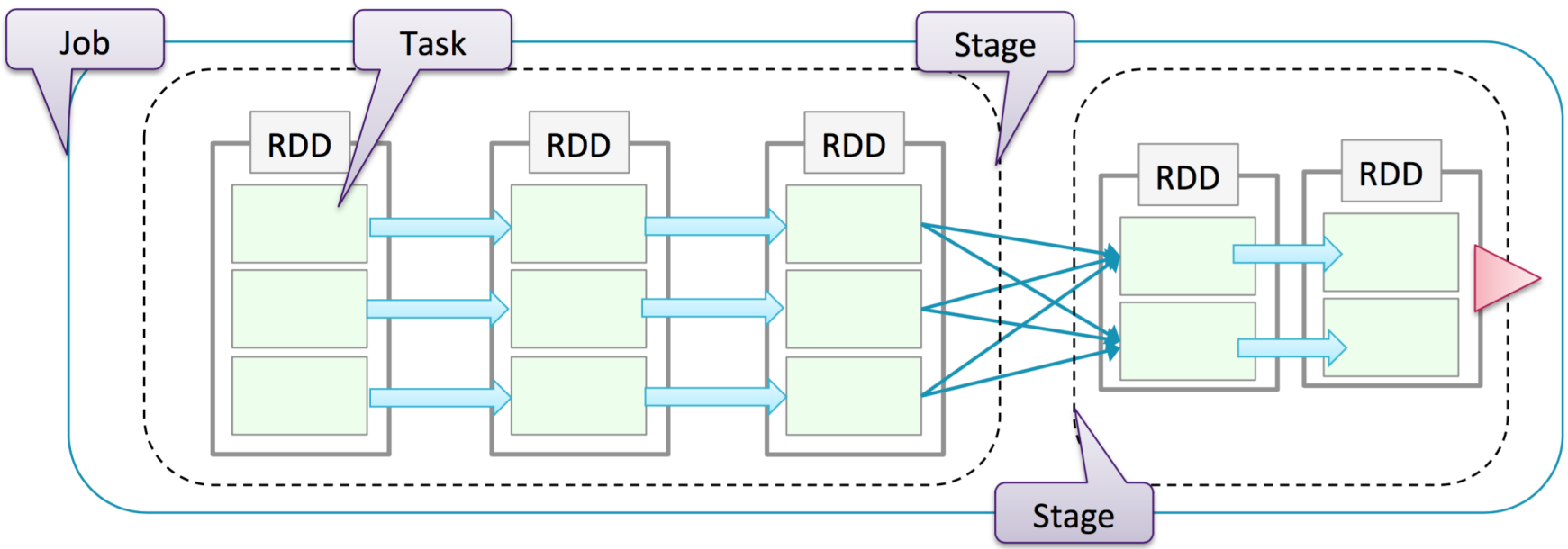
Spark Terminology

- **Job** : a set of tasks executed as a result of an action
- **Stage** : a set of tasks in a job that can be executed in parallel



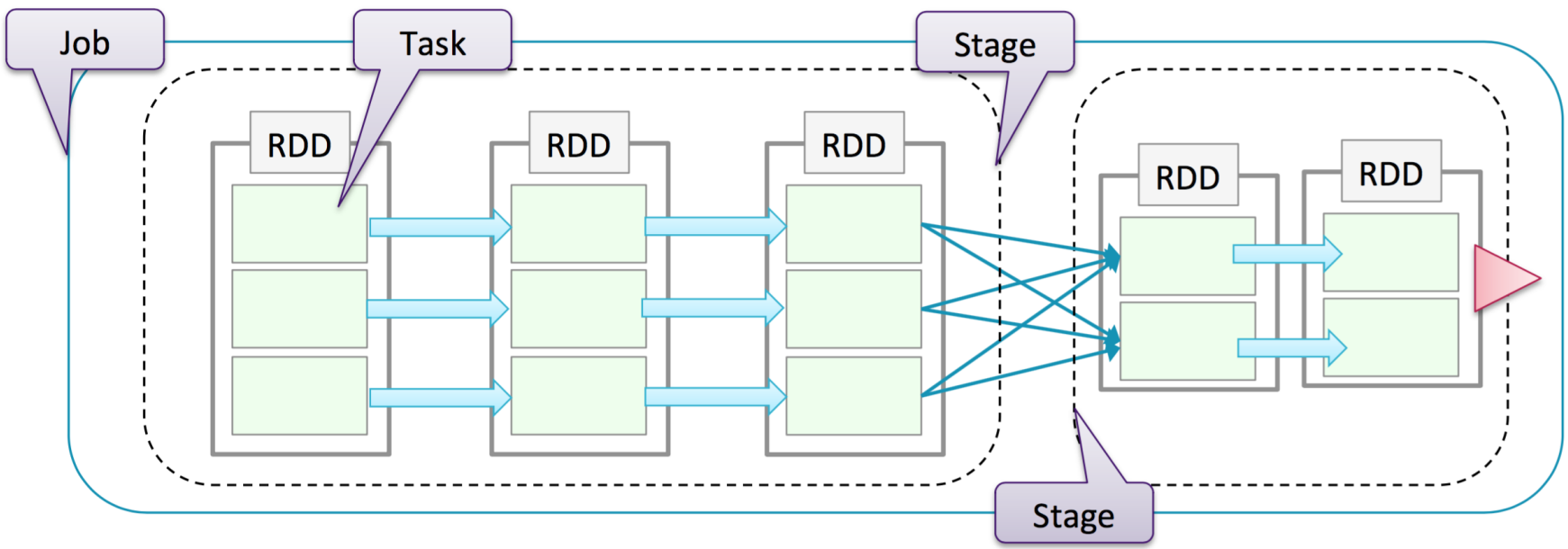
Spark Terminology

- **Job** : a set of tasks executed as a result of an action
- **Stage** : a set of tasks in a job that can be executed in parallel
- **Task** : an individual unit of work sent to one executor



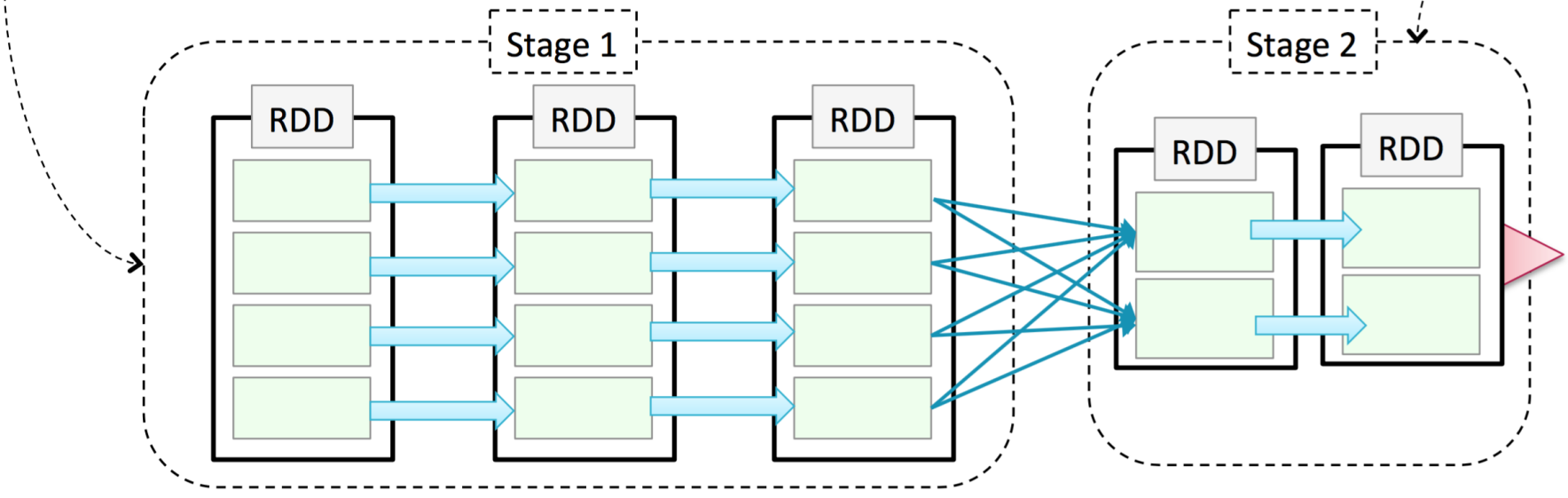
Spark Terminology

- **Job** : a set of tasks executed as a result of an action
- **Stage** : a set of tasks in a job that can be executed in parallel
- **Task** : an individual unit of work sent to one executor
- **Application** – can contain any number of jobs managed by a single driver



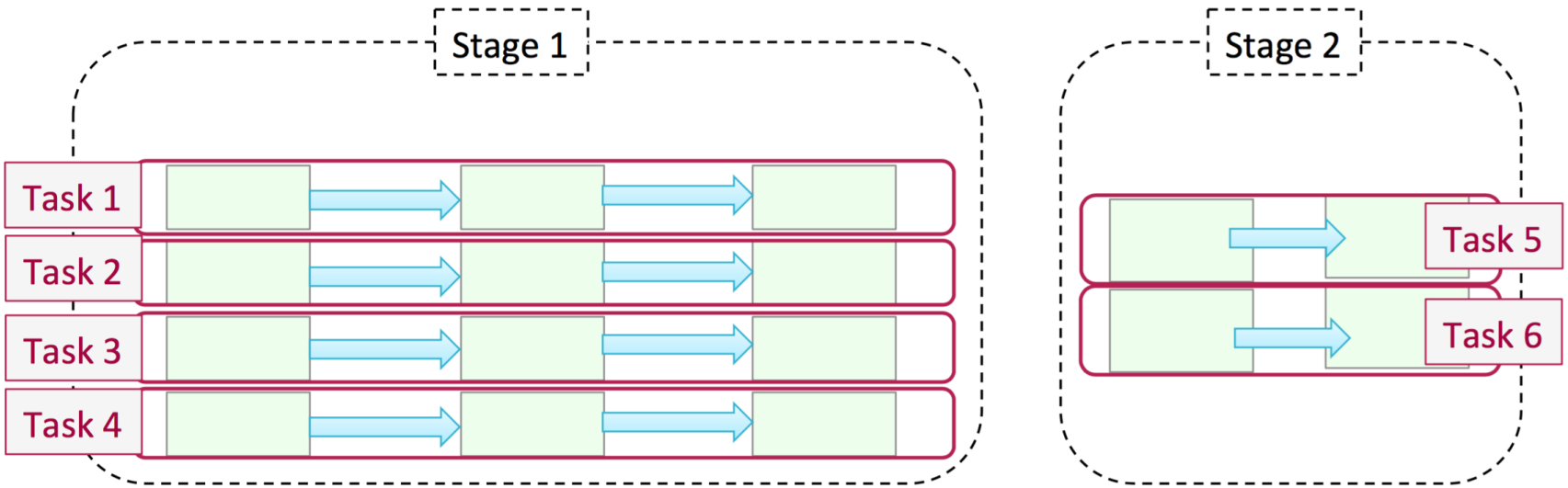
Stages and Tasks

```
> val avglens = sc.textFile(myfile).
    flatMap(line => line.split("\\W")).
    map(word => (word(0),word.length)).
    groupByKey().
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))

> avglens.saveAsTextFile("avglen-output")
```



Stages and Tasks

```
> val avglens = sc.textFile(myfile).
    flatMap(line => line.split("\\W")).
    map(word => (word(0),word.length)).
    groupByKey().
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))

> avglens.saveAsTextFile("avglen-output")
```



Let's do it !

Please refer to the Hands'On Exercise Manual.
Hands :On Exercise : Write and Run a Spark Application

SparkSQL

What is SparkSQL ?

- Higher level of abstraction

What is SparkSQL ?

- Higher level of abstraction
- A distributed collection of data organized into named columns

What is SparkSQL ?

- Higher level of abstraction
- A distributed collection of data organized into named columns
- Query data using SQL

What is SparkSQL ?

- Higher level of abstraction
- A distributed collection of data organized into named columns
- Query data using SQL
- Constructed from data file, hive table, external DB, existing RDD

Date Frames

movie	firstname	name	year
intouchables	omar	sy	2013
inception	leonardo	dicaprio	2011
star wars	samuel	jackson	1996

```
val castsDF = casts.toDF
castsDF.printSchema
castsDF.show
castsDF.filter("movie = 'inception'").select('firstname, 'name).show
castsDF.select('firstname, ‘name).distinct.show
castsDF.groupBy('year).count().show()
```

Spark Streaming

What is Spark Streaming ?

- Spark API for **Real Time processing**

What is Spark Streaming ?

- Spark API for **Real Time processing**
- Low latency & high throughput

What is Spark Streaming ?

- Spark API for **Real Time processing**
- Low latency & high throughput
- Uses fast batch processing

What is Spark Streaming ?

- Spark API for **Real Time processing**
- Low latency & high throughput
- Uses fast batch processing
- Can combine with classic RDD