MINERAÇÃO MASSIVA DE DADOS

Parte 12 – Spark Distributed Processing

Marcial Porto Fernández
marcial@larces.uece.br

Mestrado Acadêmico em Ciência da Computação (MACC)
Universidade Estadual do Ceará (UECE)
Laboratório de Sistemas Digitais (LASID)

Sumário

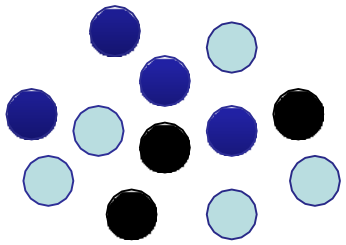
- Recap Map Reduce
- Spark Distributed Architecture
- Anonymous Function
- Spark Operations
- Spark Programming Tips
- Example: Word Count

Sumário

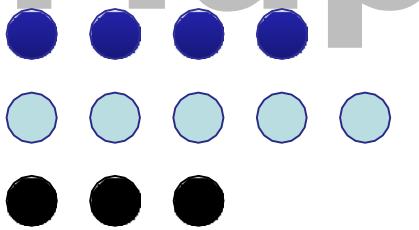
- Recap Map Reduce
- Spark Distributed Architecture
- Anonymous Function
- Spark Operations
- Spark Programming Tips
- Example: Word Count

The MapReduce Paradigm

- Parallel processing paradigm
- Programmer is unaware of parallelism
- Programs are structured into a two-phase execution

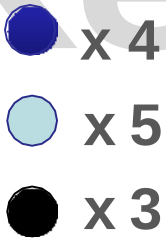


Map



Data elements are classified into categories

Reduce



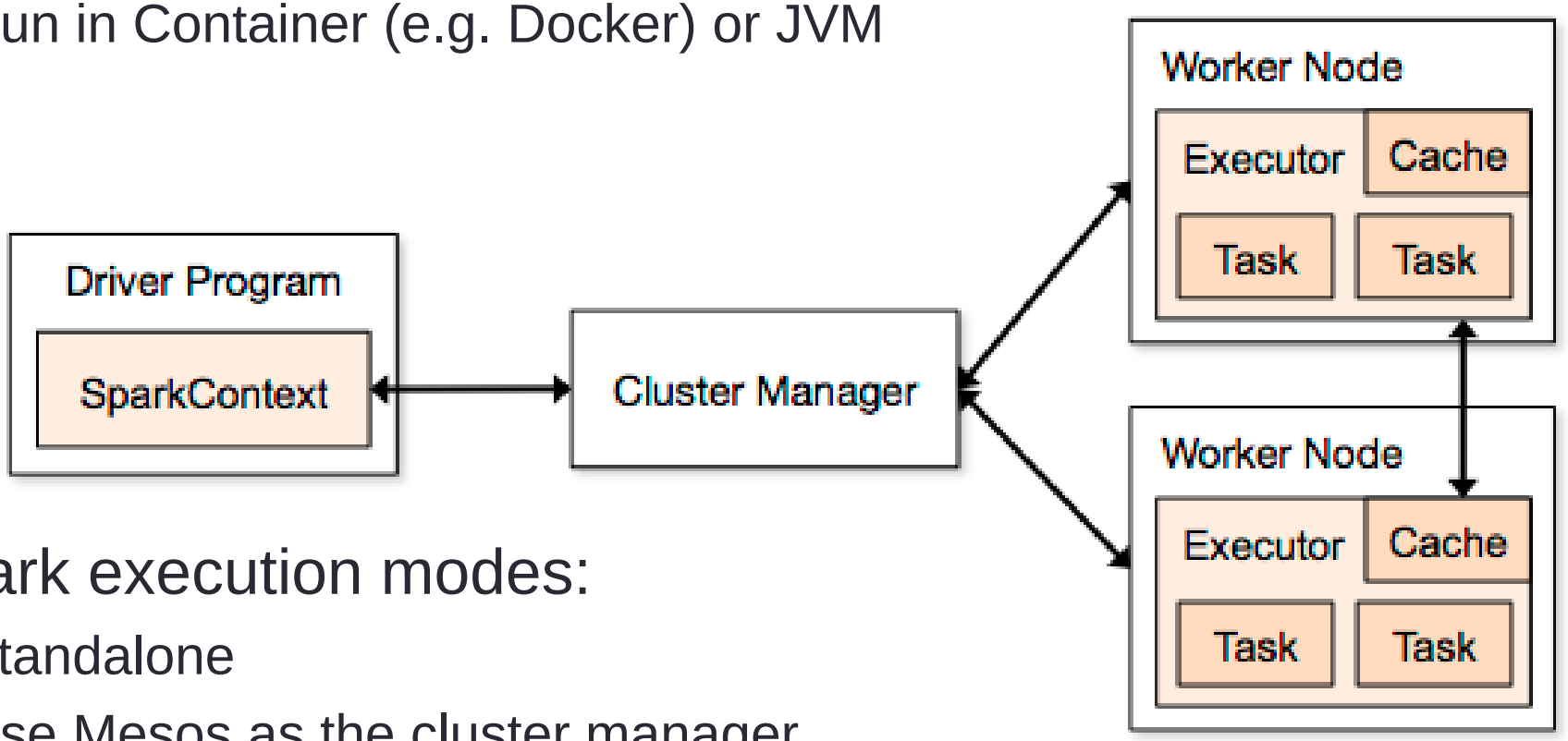
An algorithm is applied to all the elements of the same category

Sumário

- Recap Map Reduce
- Spark Distributed Architecture
- Anonymous Function
- Spark Operations
- Spark Programming Tips
- Example: Word Count

Spark Application Architecture

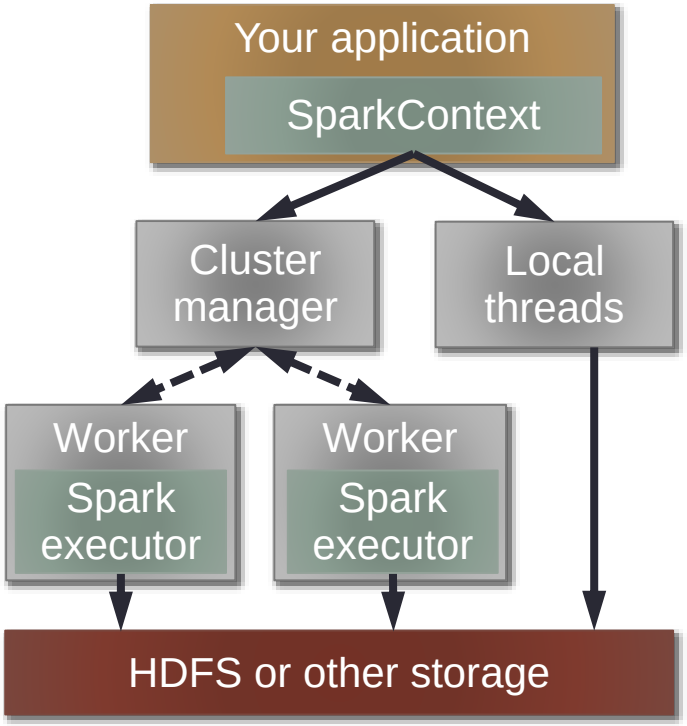
- A Spark application is initiated from a driver program
  - Run in Container (e.g. Docker) or JVM



- Spark execution modes:
  - Standalone
  - Use Mesos as the cluster manager
  - Use YARN as the cluster manager
  - Use Kubernetes as the cluster manager

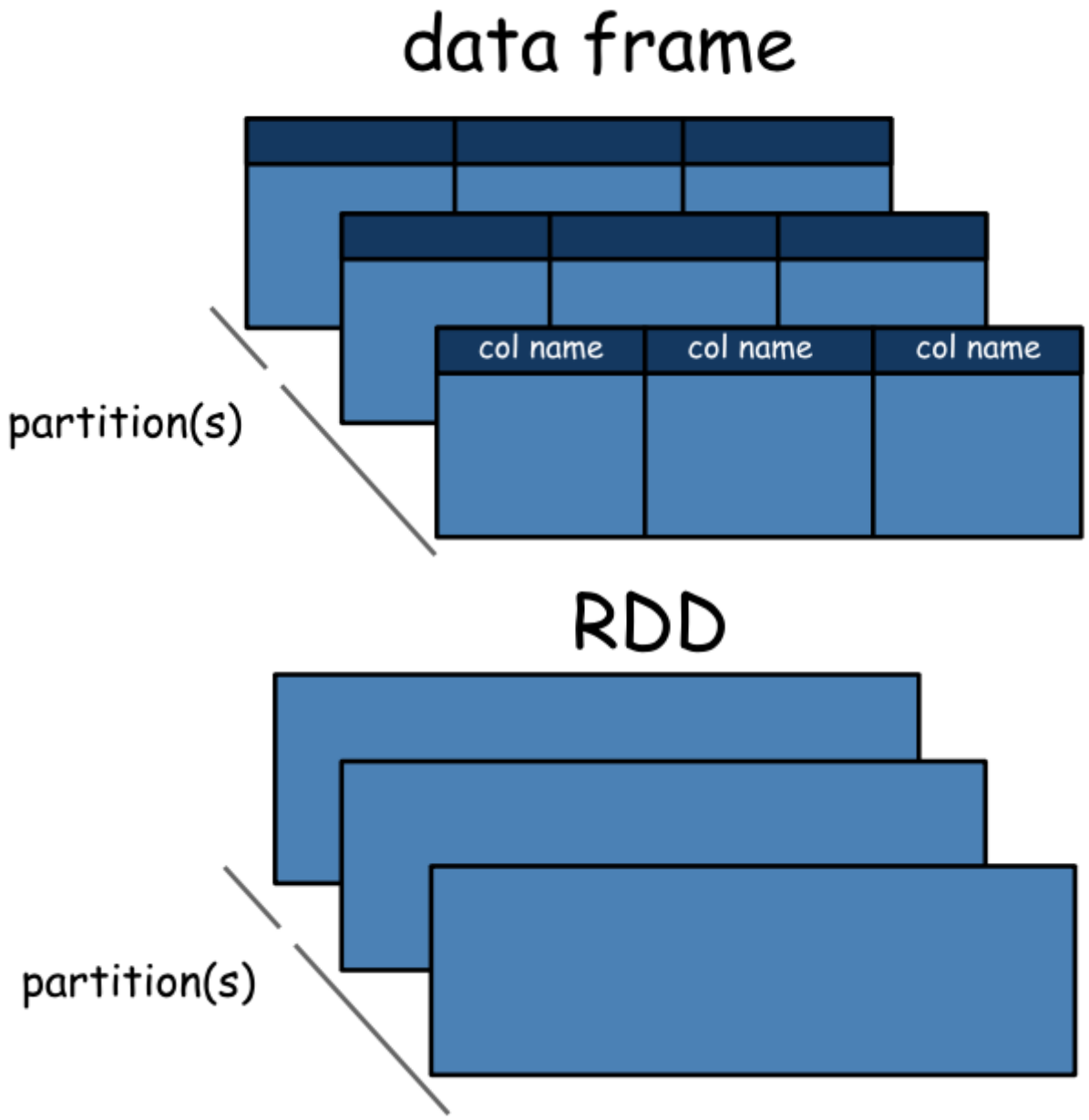
Spark Software Components

- Spark runs as a library in your program (one instance per app)
- Runs tasks locally or on a cluster
  - Standalone deploy, cluster, Mesos or YARN
- Accesses storage via InputFormat API
  - Can use HBase, HDFS, S3, …

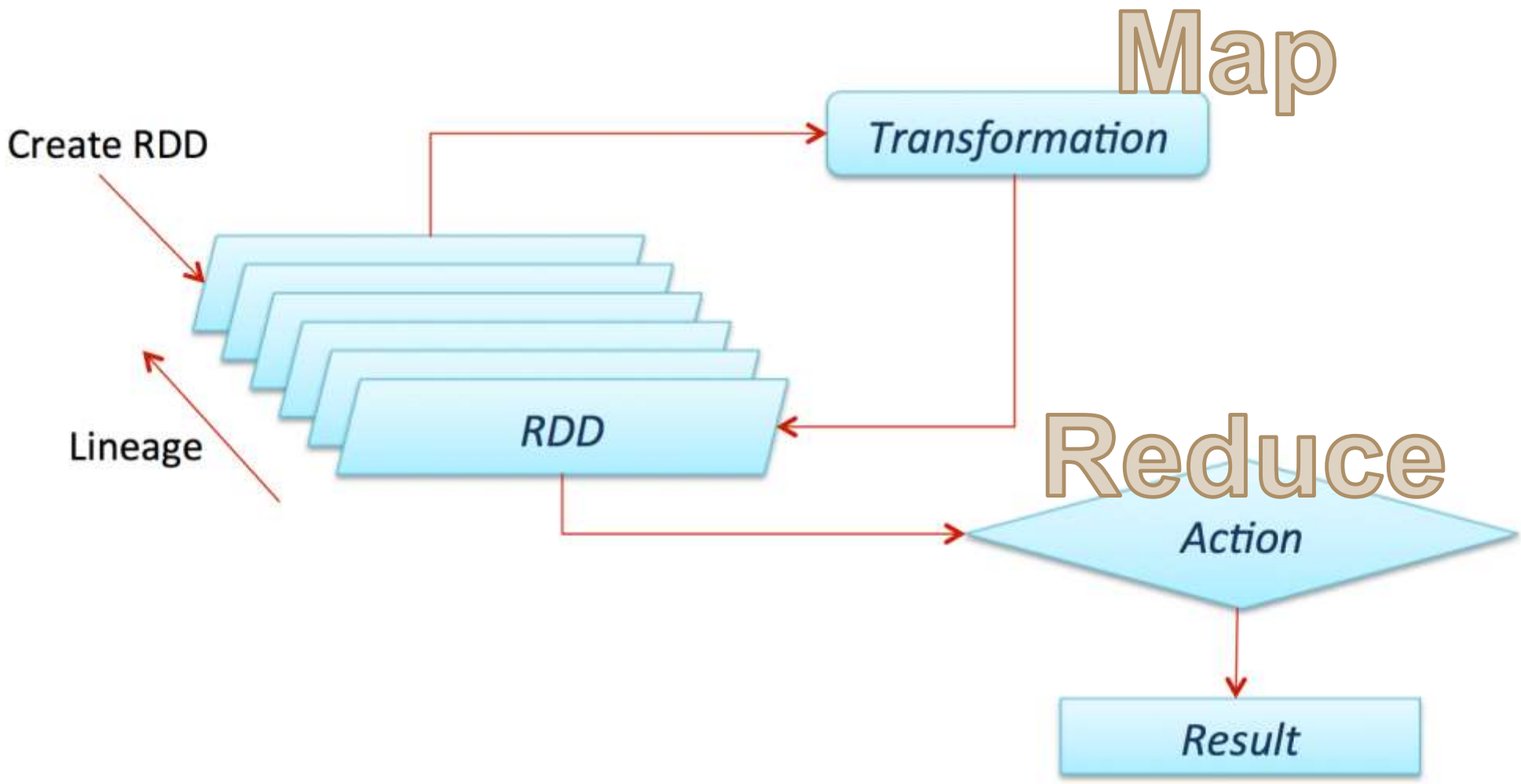


Spark Distributed Architecture

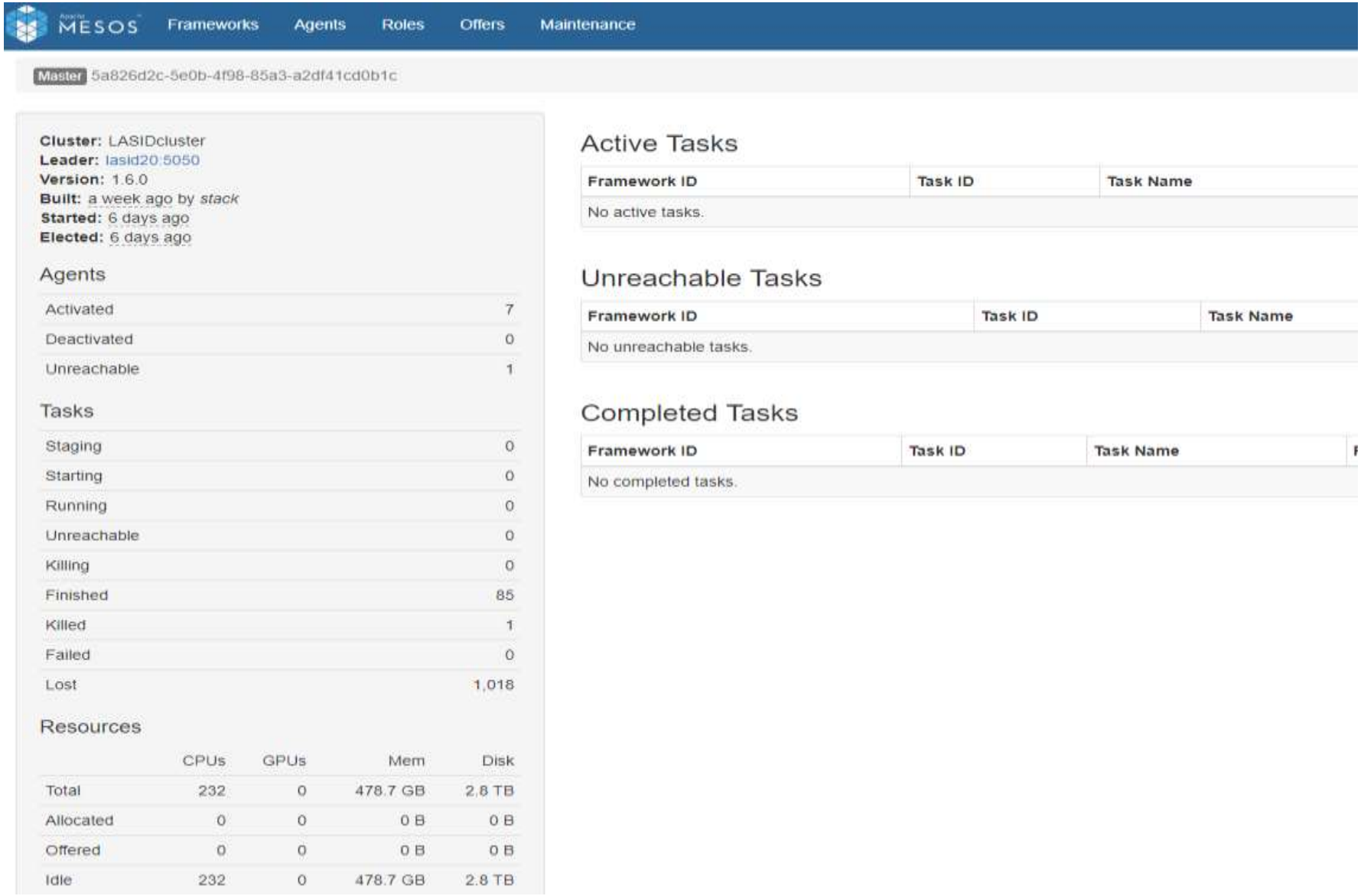
- Data partition
- Transformation Dataframe to RDD
- RDD is basically a Dataframe without collumn label



Spark Distributed Architecture



Mesos Dashboard



Mesos Dashboard

Completed Tasks

Find...

ID ▼	Name	Role	State	Started	Stopped	Host
9	SparkParallel 9	*	LOST		2018-11-29T18:47:19-0300	lasid10
8	SparkParallel 8	*	LOST		2018-11-29T18:47:19-0300	lasid50
7	SparkParallel 7	*	LOST		2018-11-29T18:47:19-0300	lasid70
6	SparkParallel 6	*	LOST		2018-11-29T18:47:19-0300	lasid60
5	SparkParallel 5	*	LOST		2018-11-29T18:47:19-0300	lasid40
4	SparkParallel 4	*	LOST		2018-11-29T18:47:19-0300	lasid50
3	SparkParallel 3	*	LOST		2018-11-29T18:47:19-0300	lasid10
2	SparkParallel 2	*	LOST		2018-11-29T18:47:19-0300	lasid30
12	SparkParallel 12	*	LOST		2018-11-29T18:47:19-0300	lasid30
11	SparkParallel 11	*	LOST		2018-11-29T18:47:19-0300	lasid40
10	SparkParallel 10	*	LOST		2018-11-29T18:47:19-0300	lasid60
1	SparkParallel 1	*	LOST		2018-11-29T18:47:19-0300	lasid70
0	SparkParallel 0	*	FINISHED	2018-11-29T18:47:22-0300	2018-11-29T18:47:31-0300	10.129.64.20

Sumário

- Recap Map Reduce
- Spark Distributed Architecture
- Anonymous Function
- Spark Operations
- Spark Programming Tips
- Example: Word Count

Anonymous function

- **Anonymous functions** originate in the work of Alonzo Church in his invention of the lambda calculus in 1936.
  - Also known as *function literal*, *lambda abstraction*, or *lambda expression*.
- In several programming languages, anonymous functions are introduced using the keyword *lambda*.
- Anonymous functions are often referred to as lambdas or lambda abstractions.
- The first programming language where anonymous functions have been used was Lisp in 1958.

Lambda Architecture

- Nathan Marz came up with the term **Lambda Architecture** for a generic, scalable and fault-tolerant data processing architecture.
- It is *data-processing architecture* designed to handle massive quantities of data by taking advantage of both batch and stream processing methods.
- Like Python def, the lambda creates a function to be called later.
- But it returns the function instead of assigning it to a name.
- In practice, they are used as a way to inline a function definition, or to set a code execution.

Lambda Python Examples

- **lambda** arg1, arg2, ...argN **:** expression using arguments
- **def func**(arg1, arg2, ...argN) **:**
  value = expression using arguments
  return value

```
>>>
>>> def func(x): return x ** 3

>>> print(func(5))
125
>>>
>>> lamb = lambda x: x ** 3
>>> print(lamb(5))
125
>>>
```

```
lines = sc.textFile("data.txt")
lineLengths = lines.map(lambda s: len(s))
totalLength = lineLengths.reduce(lambda a, b: a + b)
```

```
f = lambda x,y: ["PASS",x,y] if x>3 and y<100 else ["FAIL",x,y]
```

```
print(f(4,50))
```

```
['FAIL', 4, 200]
```

Spark Lambda Python Examples

```
lines = sc.textFile("data/cs100/lab1/shakespeare.txt")
```

- Read the lines in a string
  - ['word1 word2 word3','word4 word3 word2']

```
counts = (lines.flatMap(lambda x: x.split(' '))
                .map(lambda x: (x, 1))
                .reduceByKey(lambda x,y : x + y))
```

- Split all words in FlatMap
  - ['word1','word2','word3','word4','word3','word2'].
- Identify all word occurrence in Map
  - [('word1',1),('word2',1),('word3',1),('word4',1),('word3',1),('word2',1)].
- Count the number of occurences in ReduceByKey
  - [('word1',1),('word2',2),('word3',2),('word4',1)].

Sumário

- Recap Map Reduce
- Spark Distributed Architecture
- Anonymous Function
- Spark Operations
- Spark Programming Tips
- Example: Word Count

Spark Operations

- Spark support two types of operations over RDD:
- **Transformations** are operations that are performed on an RDD and which yield a new RDD containing the result.
  - Ex: map, filter, join, union, and so on
- **Actions** are operations that return a value after running a computation on an RDD.
  - Ex: reduce, count, first, and so on
- Transformations are “lazy”, meaning that they do not compute their results right away.
  - Instead, they just “remember” the operation to be performed to the dataset (e.g., file) to which the operation is to be performed.
- The transformations are only actually computed when an action is called and the result is returned to the driver program.
- This design enables Spark to run more efficiently.
  - If a big file was transformed in various ways and passed to first action, Spark would only process and return the result once, rather than the entire file.

Spark Operations

- Now, to intuitively get the difference between these two, consider some of the most common transformations are:
  - map(), filter(), flatMap(), sample(), randomSplit(), coalesce() and repartition()
- Some of the most common actions are:
  - reduce(), collect(), first(), take(), count() and saveAsHadoopFile().
- **Transformations** are lazy operations on a RDD that create one or many new RDDs.
- **Actions** produce non-RDD values, they return a result set, a number, a file, …

Main Spark Transformations

- map(func): Return a new distributed dataset formed by passing each element of the source through a function func.
- flatMap(func): Similar to map(func) but func return a sequence rather than a single item (“flattening”).

```
rdd = sc.parallelize([2, 3, 4])
rdd.flatMap(lambda x: range(1, x)).collect()
Output:
[1, 1, 2, 1, 2, 3]


rdd.map(lambda x: range(1, x)).collect()
Output:
[[1], [1, 2], [1, 2, 3]]
```

Main Spark Transformations

- filter(func): Return a new dataset formed by selecting those elements of the source on which func returns true
- union(otherDataset): Return a new dataset that contains the union of the elements in the source dataset and the argument.
- intersection(otherDataset): Return a new RDD that contains the intersection of elements in the source dataset and the argument.
- distinct([numTasks])): Return a new dataset that contains the distinct elements of the source dataset
- join(otherDataset, [numTasks]): When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.

Main Spark Actions

- reduce(func): Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
- collect(): Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
- count(): Return the number of elements in the dataset.

Remember: *Actions* cause calculations to be performed; *transformations* just set things up (lazy evaluation)

Spark Operations

- **Transformations** (create a new RDD)

map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func*returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.

Spark Operations

- **Transformations** (create a new RDD)

sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numPartitions*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numPartitions argument to set a different number of tasks.

Spark Operations

- **Transformations** (create a new RDD)

reduceByKey(*func*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numPartitions*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.
join(*otherDataset*, [*numPartitions*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.

Spark Operations

- **Transformations** (create a new RDD)

cogroup(*otherDataset*, [*numPartitions*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called groupWith.
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(*command*, *[envVars]*)	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(*numPartitions*)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
repartition(*numPartitions*)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
repartitionAndSortWithinPartitions(*partitioner*)	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery.

Spark Operations

- **Actions** (return results to driver program)

reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.

Spark Operations

- **Actions** (return results to driver program)

takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(*path*)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(*path*) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).

Spark Operations

- **Actions** (return results to driver program)

saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded usingSparkContext.objectFile().
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. **Note**: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior.

Sumário

- Recap Map Reduce
- Spark Distributed Architecture
- Anonymous Function
- Spark Operations
- Spark Programming Tips
- Example: Word Count

Create SparkContext

- Main entry point to Spark functionality
- Created for you in Spark shells as variable sc
- SparkContext is a subgroup of SparkSession.
- SparkContext is pure RDD, without labels, to be processed in parallel.

Creating RDDs

```
# Turn a local collection into an RDD
sc.parallelize([1, 2, 3])

# Load text file from local FS, HDFS, or S3
sc.textFile(“file.txt”)
sc.textFile(“directory/*.txt”)
sc.textFile(“hdfs://namenode:9000/path/file”)

# Convert Dataframe in RDD
sc = df.rdd

# Read CSV to RDD (via Dataframe)
sc = spark.read.csv(“filename.csv”).rdd
```

Basic Transformations

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

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

# Keep elements passing a predicate
even = squares.filter(lambda x: x % 2 == 0) # => {4}

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

Range object (sequence of numbers 0, 1, …, x-1)

Basic Actions

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

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

# Return 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”)
```

Some Key-Value Operations

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

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

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

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

`reduceByKey()` also automatically implements combiners on the map side

Multiple Datasets

```
visits = sc.parallelize([(“index.html”, “1.2.3.4”),
                         (“about.html”, “3.4.5.6”),
                         (“index.html”, “1.3.3.1”)])

pageNames = sc.parallelize([(“index.html”, “Home”), (“about.html”, “About”)])

visits.join(pageNames)
# (“index.html”, (“1.2.3.4”, “Home”))
# (“index.html”, (“1.3.3.1”, “Home”))
# (“about.html”, (“3.4.5.6”, “About”))

visits.cogroup(pageNames)
# (“index.html”, (Seq(“1.2.3.4”, “1.3.3.1”), Seq(“Home”)))
# (“about.html”, (Seq(“3.4.5.6”), Seq(“About”)))
```

Controlling the Level of Parallelism

- All the action operations take an optional second parameter for number of tasks

```
words.reduceByKey(lambda x, y: x + y, 5)
words.groupByKey(5)
visits.join(pageViews, 5)
```

Using Local Variables

- External variables you use in a closure will automatically be shipped to the cluster:

```
query = raw_input("Enter a query:")
pages.filter(lambda x: x.startswith(query)).count()
```

- Some caveats:
  - Each task gets a new copy (updates aren't sent back)
  - Variable must be Serializable (Java/Scala) or Pickle-able (Python)
  - Don't use fields of an outer object (ships all of it!)

Example: K-Means

- Map read data and Split values

```
$ cat data/mllib/kmeans_data.txt
0.0 0.0 0.0
0.1 0.1 0.1
0.2 0.2 0.2
9.0 9.0 9.0
9.1 9.1 9.1
9.2 9.2 9.2
```

```
from pyspark import SparkContext
from pyspark.mllib.clustering import KMeans, KMeansModel
from numpy import array
from math import sqrt

sc = SparkContext(appName = "K-Means")

# Load and parse the data
data = sc.textFile("data/mllib/kmeans_data.txt")
parsedData = data.map(lambda line: array(map(float,
    line.split())))
```

Example: K-Means

- Train in one process
- Test dataset distributed

```
11 # Build the model (cluster the data)
12 clusters = KMeans.train(parsedData, 2, maxIterations=10,
13         runs=10, initializationMode="random")
14
15 # Evaluate clustering by computing WCSS
16 def error(point):
17     center = clusters.centers[clusters.predict(point)]
18     return sqrt(sum([x**2 for x in (point - center)]))
19
20 WCSS = parsedData.map(error).reduce(lambda x, y: x + y)
21 print("Within Set Sum of Squared Error = " + str(WCSS))
22
23 # Save and load model
24 clusters.save(sc, "myModelPath")
25 sameModel = KMeansModel.load(sc, "myModelPath")
```

Distributed Spark Keys Points

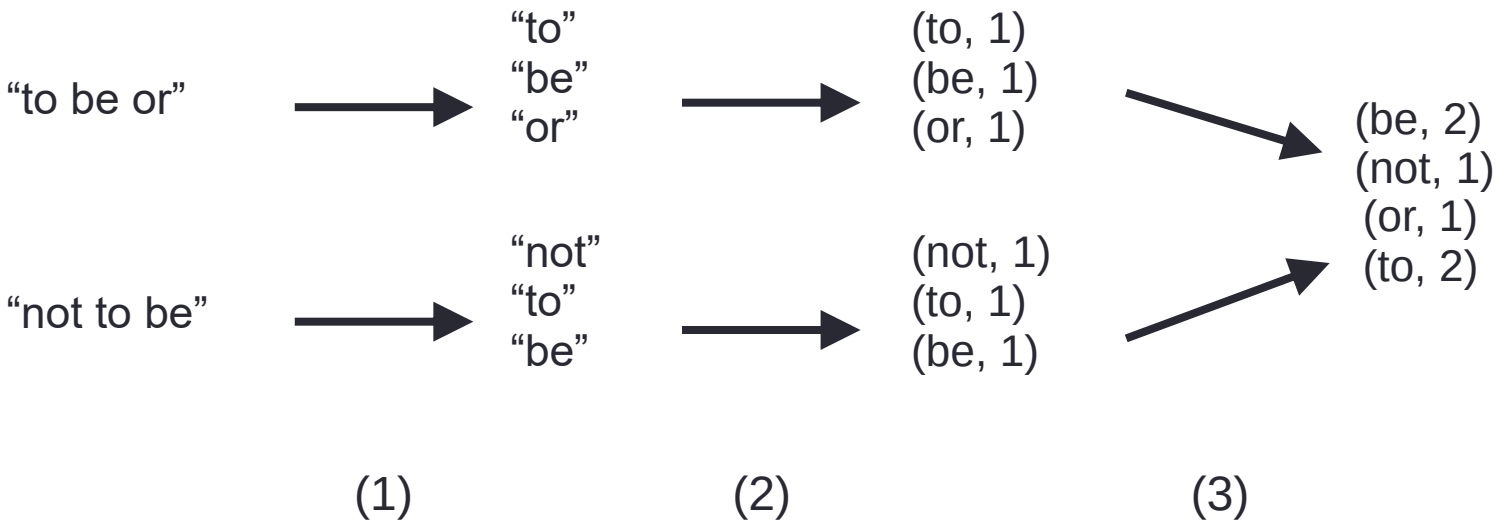
- Data should be RDD format
  - Spark doesn’t need headers...
  - Data should be divided in parts
- Isolate Transform function (Map) and Action functions (Reduce)
  - The parallelism is automatically (“automagically”?)
- If you use many computers, you should use a cluster management

Sumário

- Recap Map Reduce
- Spark Distributed Architecture
- Anonymous Function
- Spark Operations
- Spark Programming Tips
- Example: Word Count

Example: Word Count

```
lines = sc.textFile(“hamlet.txt”)
counts = lines.flatMap(lambda line: line.split(“ ”)) \   (1)
              .map(lambda word: (word, 1)) \             (2)
              .reduceByKey(lambda x, y: x + y)           (3)
```



Example: Word Count

Aula 12: Spark Parallel Processing

```
import findspark
findspark.init()

from pyspark import SparkContext
from pyspark.sql import SparkSession
import time

%env MESOS_NATIVE_JAVA_LIBRARY=/usr/local/lib/libmesos.so

#book_file = "/data/textdata/books/english/Dracula.txt"
book_file = "/data/textdata/books/english/War_and_Peace.txt"
```

```
env: MESOS_NATIVE_JAVA_LIBRARY=/usr/local/lib/libmesos.so
```

Example: Word Count

Word Count Serial

```
# Create local Spark session
spark = SparkSession.builder\
        .appName("SparkSerial")\
        .master("local") \
        .getOrCreate()

# create the Spark Context
sc = spark.sparkContext
```

Reading Data

```
# Read de book file
text_file = sc.textFile('file://'+ book_file)
```

Example: Word Count

```
start_time = time.time()
counts = text_file.flatMap(lambda line: line.split(' ')) \
              .map(lambda word: (word, 1)) \
              .reduceByKey(lambda a, b: a + b)

#counts.saveAsTextFile("hdfs://...")
counts.toDF(["Word", "Count"]).sort("Count",ascending=False).show(5)

print("--- Execution time: %s seconds ---" % (time.time() - start_time))
```

```
+----+-----+
|Word|Count|
+----+-----+
| the|31704|
| and|20564|
|    |16769|
|  to|16320|
|  of|14855|
+----+-----+
only showing top 5 rows

--- Execution time: 5.566906929016113 seconds ---
```

Example: Word Count

Word Count Parallel

```
spark = SparkSession.builder\
        .appName("SparkParallel")\
        .master("mesos://zk://10.129.64.20:2181/mesos") \
        .config("spark.executor.uri", "http://10.129.64.20/physdata/spark-2.3.1-bin-hadoop2.7.tgz")\
        .getOrCreate()

#         .master("mesos://zk://10.129.64.20:2181/mesos") \
#         .master("mesos://10.129.64.20:5050") \
#         .master("mesos://zk://10.129.64.20:2181,10.129.64.10:2181,10.129.64.30:2181/mesos") \

# Enable Arrow-based columnar data transfers
#spark.conf.set("spark.sql.execution.arrow.enabled", "true")
spark.conf.set("spark.submit.deployMode", "cluster")
spark.conf.set("spark.driver.supervise", "true")
spark.conf.set("spark.executor.memory", "4g")

# create the Spark Context
sc = spark.sparkContext
```

Example: Word Count

Word Count one data partition

```
# Read de book file
text_file = sc.textFile('file://'+ book_file)
```

```
start_time = time.time()
counts = text_file.flatMap(lambda line: line.split(' ')) \
              .map(lambda word: (word, 1)) \
              .reduceByKey(lambda a, b: a + b)

#counts.saveAsTextFile("hdfs://...")
counts.toDF(["Word", "Count"]).sort("Count",ascending=False).show(5)

print("--- Execution time: %s seconds ---" % (time.time() - start_time))
```

```
+----+-----+
|Word|Count|
+----+-----+
| the|31704|
| and|20564|
|    |16769|
|  to|16320|
|  of|14855|
+----+-----+
only showing top 5 rows

--- Execution time: 10.740650653839111 seconds ---
```

Example: Word Count

Word Count five data partitions

```
# Read de book file
part = 5
text_file_part = sc.textFile('file://'+ book_file, part)
```

```
start_time = time.time()
counts = text_file_part.flatMap(lambda line: line.split(' ')) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)

#counts.saveAsTextFile("hdfs://...")
counts.toDF(["Word", "Count"]).sort("Count",ascending=False).show(5)

print("--- Execution time: %s seconds ---" % (time.time() - start_time))
```

```
+----+-----+
|Word|Count|
+----+-----+
| the|31704|
| and|20564|
|    |16769|
|  to|16320|
|  of|14855|
+----+-----+
only showing top 5 rows

--- Execution time: 0.8180429935455322 seconds ---
```

OBRIGADO !

marcial@larces.uece.br
http://marcial.larces.uece.br