Ricardo Aler Mur

- The limitations of the MapReduce programming model are explained, and Spark is shown to solve them.
- Basic concepts are introduced, specially the RDD (Resilient Distributed Dataset) and the concept of transformation and action.
- Transformations transform a RDD into another RDD, but its execution is lazy. That means that noting happens when the transformation is applied.
- Only when an action is executed, all the transformations are actually applied and run.
- Some examples of transformations and actions are explained.

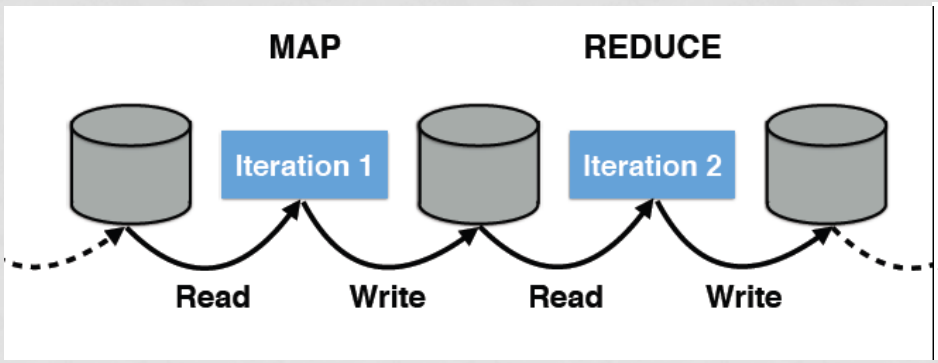
- A more complex RDD is introduced: pair RDDs, where every instance constains a key and a value.
- Some specific transformations for pair RDDs are explained: *reduceByKey and flatMap*.
- It is shown that the MapReduce programming model can be programmed in Spark with Map and ReduceByKey.
- Two of the main Spark libraries for Machine Learning are introduced: Mlib and ML, the latter being the most recent one. Mlib relies on RDDs and the LabelledPoint data type, while ML relies on a more complex data strcture called DataFrame.
- The non-supervised K-means algorithm is explained now within the Spark programming model.

LARGE SCALE MACHINE LEARNING: SPARK

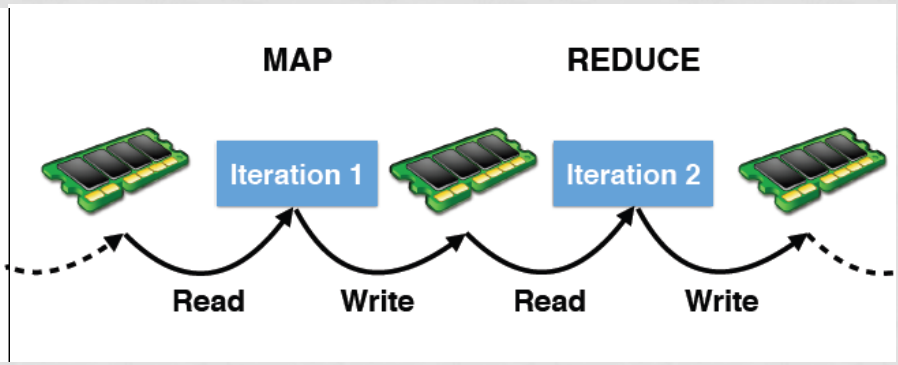
MAPREDUCE / HADOOP LIMITATIONS

- For every map / reduce iteration, MapReduce must save results to disk (and more specifically, to the distributed file system, which involves replication for failure recovery)
- Nowadays, the price of RAM has decreased and it is faster to save results to RAM memory (partially, at least)
- Spark uses some of the ideas of MapReduce, but it is oriented to a more heavy use of RAM

MapReduce

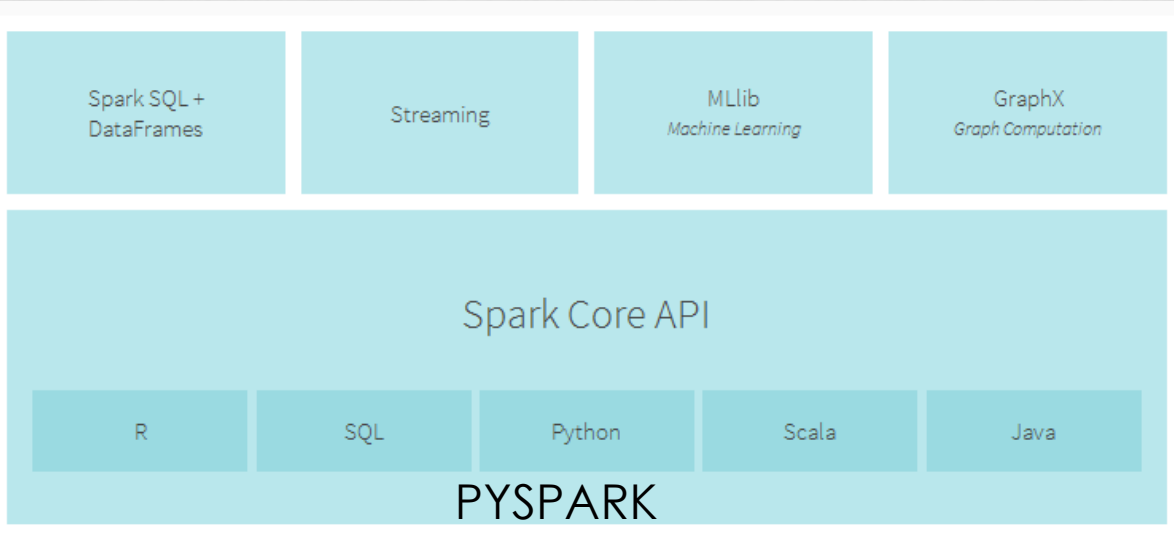


Spark



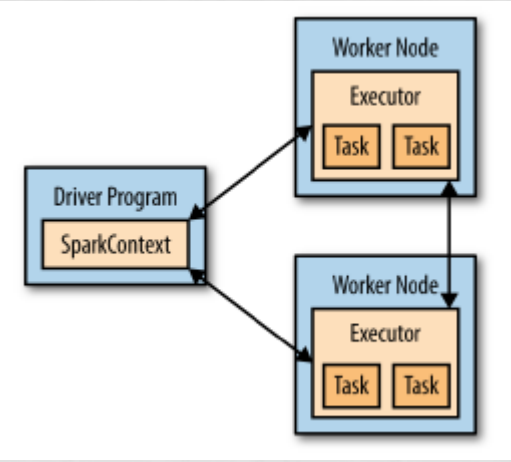
SPARK ECOSYSTEM

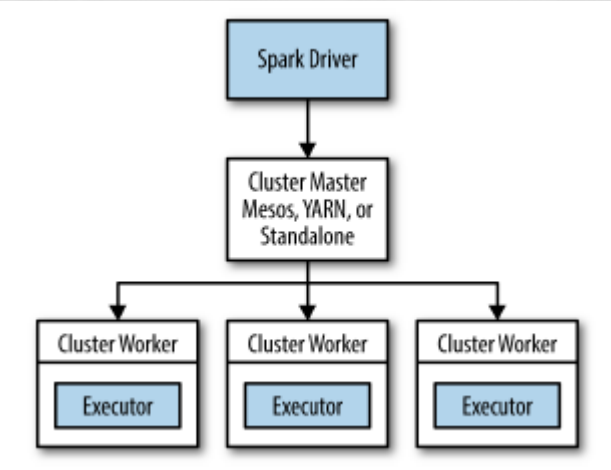
- Spark native language is Scala, but it can be programmed also in Python via the Pyspark package
- Scala is faster, but Pyspark allows to use the Python language and Python libraries



BASIC CONCEPTS

- **Driver:** It runs the main user program. It accesses the Spark environment through a **SparkContext** object.
- **Executor**: it executes tasks spawned from the driver
- **RDD**: Resilient Distributed Dataset:
  - It contains **distributed data**, spread across **partitions**
  - **Transformations** on them are carried out in parallel (data parallelism) (but RDDs themselves are **inmutable**).
  - If something goes wrong with one of the workers, Spark recomputes (part of) the RDD automatically





CREATING RDDS

- Distributing a collection of objects, e.g. a python list
  - *lines_rdd = sc.**parallelize**([1,2,3])*
- Loading an external dataset or file.
  - *lines_rdd = sc.**textFile**('README.md', 4)*
- Transforming an existing RDD
  - *rounded_rdd= numbers_rdd.**map**(round)*

For more information, check the Spark Programming Guide at:
http://spark.apache.org/docs/latest/programming-guide.html

OPERATIONS ON RDDS

- Two types of operations:
  - Transformations: creates a new RDD from a previous one
  - Actions: computes a result based on an existing RDD

- Important: transformations are just recipes, not computations. They are not actually computed until they are needed by an action (called **lazy evaluation**). Thus, results are not loaded into memory until they are actually needed.
- For example, map is a transformation. Collect is an action. Therefore:
  - *rounded_rdd = numbers_rdd.map(round)* does nothing
  - *rounded_numbers = rounded_rdd.collect()* actually computes the result and puts it into the *rounded_numbers* variable

LAMBDA FUNCTIONS

- A lambda function is a quick way of defining a function in python:
- With def:

```
def squared(x):
  return(x**2)

numbers_rdd.map(squared)
```

- With lambda function:

```
numbers_rdd.map(lambda x: x**2)
```

MAIN TRANSFORMATIONS: MAP & FILTER

- map: Reads one element at a time. takes one value, creates a new value:

```
squared_rdd = numbers_rdd.map(lambda x: x**2)
```

- filter: Reads one element at a time. Evaluates each element. Returns the elements that pass the filter()

```
positive_rdd = numbers_rdd.filter(lambda x: x>0)
```

- flatMap

TRANSFORMATION: FLATMAP

- flatMap: it applies a function that takes one element from the rdd, but produces a list. The final rdd is flattened

```
def getWords(line):
    return(line.split())
```

```
lines_rdd =  sc.parallelize(lines)
words_rdd = lines_rdd.map(getWords)
print(words_rdd.collect())
```

```
[['In', 'a', 'hole', 'in', 'the', 'ground', 'there', 'lived', 'a',
'hobbit.'], ['Not', 'a', 'nasty,', 'dirty,', 'wet', 'hole,', 'fille
d', 'with', 'the', 'ends', 'of', 'worms', 'and', 'an', 'oozy', 'sme
```



```
words_rdd = lines_rdd.flatMap(getWords)
print(words_rdd.collect())
```

```
['In', 'a', 'hole', 'in', 'the', 'ground', 'there', 'lived', 'a',
'hobbit.', 'Not', 'a', 'nasty,', 'dirty,', 'wet', 'hole,', 'fille
d', 'with', 'the', 'ends', 'of', 'worms', 'and', 'an', 'oozy', 'sme
ll.']
```

ACTIONS

- Actions force Spark to compute transformations on RDD
- Results can be returned to the driver or saved to disk
- Every call to an action recomputes the transformation (but recomputation can be avoided by persisting results to memory or disk)

MAIN ACTIONS: COMPUTING THE RDD (OR PART OF IT)

- *collect()*: retrieves the entire RDD
  - Important: results must fit in the memory of the machine where the driver is running
- *take(n)*: like *collect*, but returns only n elements from the RDD
  - Important: this is not a sample from all the partitions. All elements might come from one or two partitions
- *takeSample()*: like *take*, but takes a random sample from all the partitions
- *top(n), takeOrdered*: like *take*, but the RDD is first ordered and the first n elements are returned

- Note: "take(n)": Spark realizes that only n elements of the RDD are needed, and it will compute only those n elements (if possible)

MAIN ACTIONS: REDUCE

- *reduce()*: Takes a function that takes two elements from the RDD and returns a single value

```
numbers_rdd = sc.parallelize([1,2,3,4,5,6])
print(numbers_rdd
      .reduce(lambda x,y:x+y))
```

```
21
```

- *count()*: counts the number of elements of the RDD

PIPELINES OF TRANSFORMATIONS AND ACTIONS

- Example: filter the even numbers, square them, and add them together: filter, map, reduce
- In python, it is possible to write a command over several lines if they are enclosed within parentheses

```
numbers_rdd = sc.parallelize([1,2,3,4,5,6,7,8,9,10])
result = (numbers_rdd
          .filter(lambda x: x % 2 == 0)
          .map(lambda x: x**2)
          .reduce(lambda x,y: x+y)
         )

print result
```

```
220
```

PERSISTENCE

- The even_rdd RDD is recomputed everytime (one for computing result and another for computing result2)

```
numbers_rdd = sc.parallelize([1,2,3,4,5,6,7,8,9,10])
even_rdd = (numbers_rdd
            .filter(lambda x: x % 2 == 0)
           )

result = (even_rdd
          .collect()
          )
result2 = (even_rdd
          .map(lambda x: x**2)
          .collect()
           )
```

PERSISTENCE

- Storing (persisting) the RDD in memory can be enforced via *persist()*

```
numbers_rdd = sc.parallelize([1,2,3,4,5,6,7,8,9,10])
even_rdd = (numbers_rdd
             .filter(lambda x: x % 2 == 0)
            )

even_rdd.persist()

result = (even_rdd
          .collect()
          )

result2 = (even_rdd
          .map(lambda x: x**2)
          .collect()
           )
```

PERSISTENCE

- Note: *sc.textFile()* or sc.parallelize() do not actually load the memory or carry out the partitioning of data. If we want to do the loading or the partition and persist the result, we must use *persist()*

lines = sc.textFile("The_Hobbit.txt")

lines.persist()

numbers = sc.parallelize([1,2,3,4])

numbers.persist()

RDDS OF KEY/VALUE PAIRS

- **Pair RDDs**: They are standard RDDs, but each element is a tuple of a key and a value: (key, value)

```
pairs_rdd = sc.parallelize([("dog",4), ("bird", 2), ("octopussy", 8),
                            ("ant", 6), ("spider", 8)])
```

```
pairs_rdd
```

```
ParallelCollectionRDD[83] at parallelize at PythonRDD.scala:391
```

OPERATIONS FOR PAIR RDDS

- *reduceByKey*: it is like reduce, but a different reduce is carried out for every key. Note: *reduceByKey* is a transformation (not an action, like *reduce*)

```
groups = [("a", 3), ("b", 2), ("a", 1), ("b", 5)]
rdd = sc.parallelize(groups)
rdd.reduceByKey(lambda x,y: x+y).collect()
```

```
[('a', 4), ('b', 7)]
```

- Other: sortByKey, groupByKey, countByKey
- collectAsMap: collects the pair RDD as a python dictionary

OPERATIONS FOR PAIR RDDS: MAP VS. MAPVALUES

- *map* and *flatMap* can be used, but if we want to maintain the keys, it is better to use *mapValues*, *flatMapValues*

```
groups = [("a", 3), ("b", 2), ("a", 1), ("b", 5)]
rdd = sc.parallelize(groups)
```

```
print(rdd.map(lambda (key,value): (key, 2*value))
         .collect())

print(rdd.mapValues(lambda value: 2*value)
         .collect())
```

```
[('a', 6), ('b', 4), ('a', 2), ('b', 10)]
[('a', 6), ('b', 4), ('a', 2), ('b', 10)]
```

MAPREDUCE AND SPARK

- MapReduce is equivalent to the map / reduceByKey Spark transformations

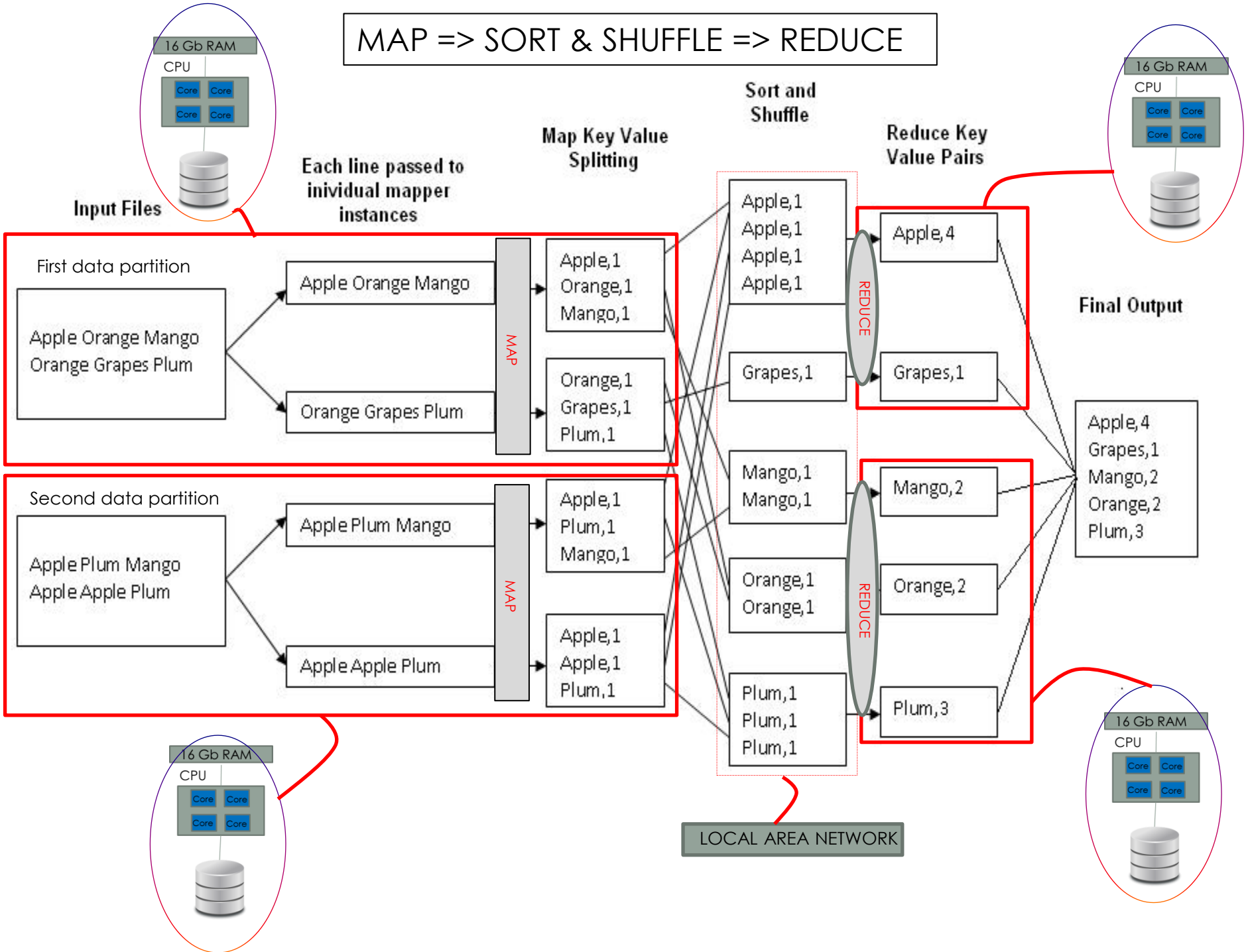
```
sc.stop()
```

```
sc = SparkContext(appName="PythonWordCount")
lines = sc.textFile("The_Hobbit.txt", 1)
counts = (lines.flatMap(lambda x: x.split(' '))
                .map(lambda x: (x, 1))
                .reduceByKey(lambda x,y:x+y)
              )
```

```
output = counts.takeOrdered(10, key=lambda x: -x[1])
for (word, count) in output:
    print("%s: %i" % (word, count))
```

```
the: 5657
and: 4275
: 2834
of: 2474
to: 2035
a: 1866
he: 1502
in: 1356
was: 1352
they: 1104
```

```
sc.stop()
```


MAP => SORT & SHUFFLE => REDUCE
16 Gb RAM
CPU
Core
Core
Core
Core
Input Files
Each line passed to individual mapper instances
Map Key Value Splitting
Sort and Shuffle
Reduce Key Value Pairs
First data partition
Apple Orange Mango
Orange Grapes Plum
Apple Orange Mango
Orange Grapes Plum
MAP
Apple,1
Orange,1
Mango,1
Orange,1
Grapes,1
Plum,1
Second data partition
Apple Plum Mango
Apple Apple Plum
Apple Plum Mango
Apple Apple Plum
MAP
Apple,1
Plum,1
Mango,1
Apple,1
Apple,1
Plum,1
Apple,1
Apple,1
Apple,1
Apple,1
Grapes,1
Mango,1
Mango,1
Orange,1
Orange,1
Plum,1
Plum,1
Plum,1
REDUCE
REDUCE
Apple,4
Grapes,1
Mango,2
Orange,2
Plum,3
Final Output
Apple,4
Grapes,1
Mango,2
Orange,2
Plum,3
LOCAL AREA NETWORK

MACHINE LEARNING IN SPARK

- Packages:
  - Mllib (Machine Learning library): common learning algorithms and utilities, including classification, regression, clustering, dimensionality reduction, …
    - Important: it is based on **standard RDDs**, using mainly the ***Labeled point*** data type
  - ML: introduced in 2015. Same as Mllib, but it is based on the recently added **DataFrames** (instead of RDDs). It allows to easily do crossvalidation, grid-search, and ML pipelines (sequences of ML operations). Very recently, another data type has been introduced (**DataSets**), which are basically typed DataFrames.

For more information, check:
http://spark.apache.org/docs/latest/mllib-guide.html

SPARK.MLLIB

- Machine learning in Spark is rapidly changing. Here, we will study only LabeledPoint (ML library), but DataFrames and DataSets will become standard in the near future
- The **LabeledPoint** datatype is a way to represent instances. It is made of two parts: features (input attributes) and label (class, output attribute)

```
from pyspark.mllib.regression import LabeledPoint

# Create a labeled point with a positive label and a dense feature vector.
pos = LabeledPoint(1.0, [1.0, 0.0, 3.0])
```

LABELEDPOINT EXAMPLE

- Let's transform the scikit Iris dataset into LabeledPoints

```
from pyspark.mllib.regression import LabeledPoint
import numpy as np
```

```
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target
data = zip(y,X)
```

```
data_rdd = sc.parallelize(data,4)
```

```
print data_rdd.getNumPartitions()
```

```
4
```

LABELEDPOINT EXAMPLE

Transform the numpy array into a spark labeled points

```
from pyspark.mllib.regression import LabeledPoint
data_rdd = data_rdd.map(lambda x: LabeledPoint(x[0], x[1]))
data_rdd.take(1)
```

```
[LabeledPoint(0.0, [5.1,3.5,1.4,0.2])]
```

```
X_rdd = data_rdd.map(lambda x: x.features)
y_rdd = data_rdd.map(lambda x: x.label)
```

```
print(X_rdd.take(2))
print(y_rdd.take(2))
```

```
[DenseVector([5.1, 3.5, 1.4, 0.2]), DenseVector([4.9, 3.0, 1.4, 0.2])]
[0.0, 0.0]
```

Algorithm *k-means (k)*

1. Initialize the location of the k prototypes k_j

   (usually, randomly)

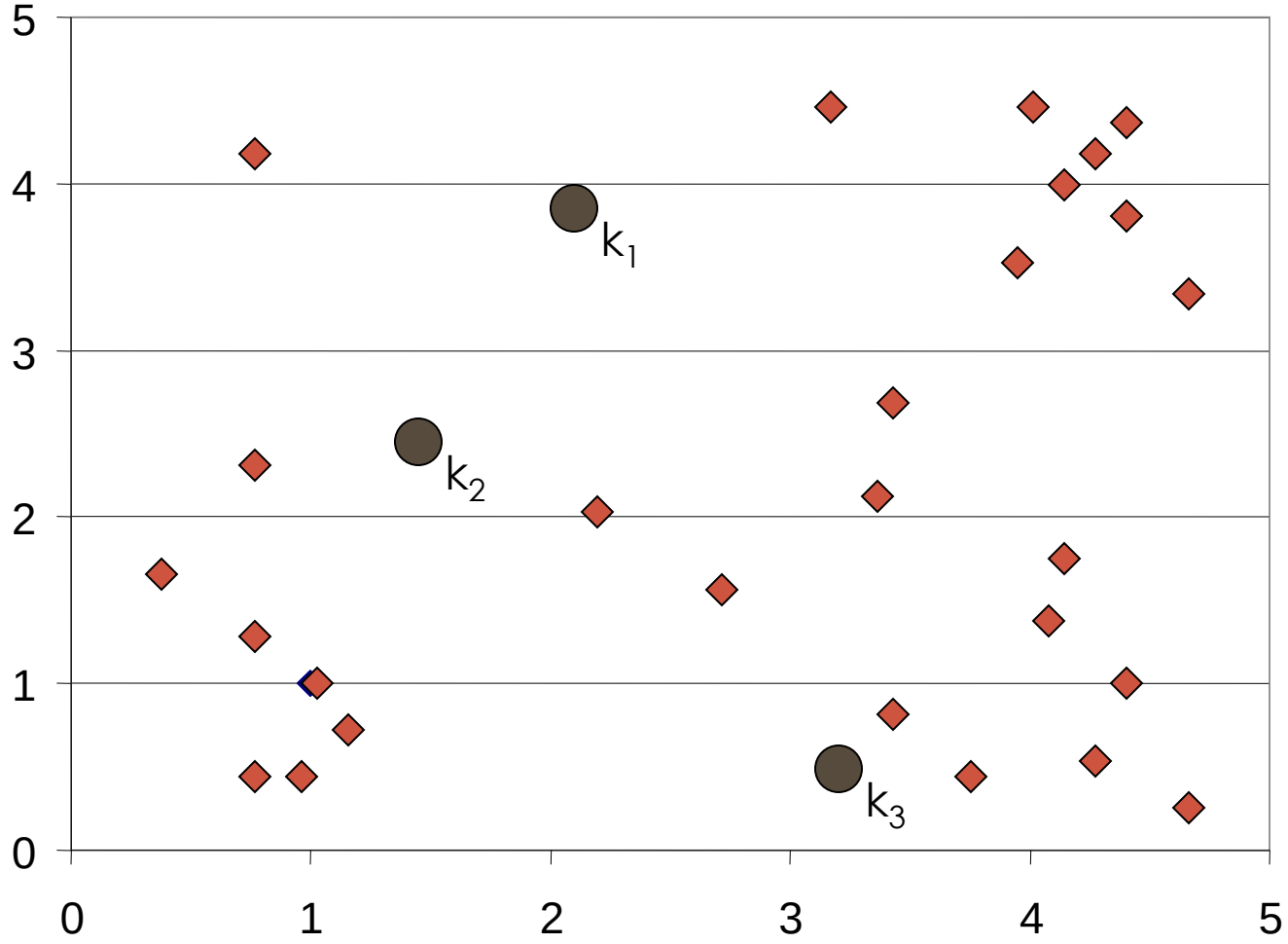
2. Assign each instance x_i to its closest prototype

   (usually, closeness = Euclidean distance).

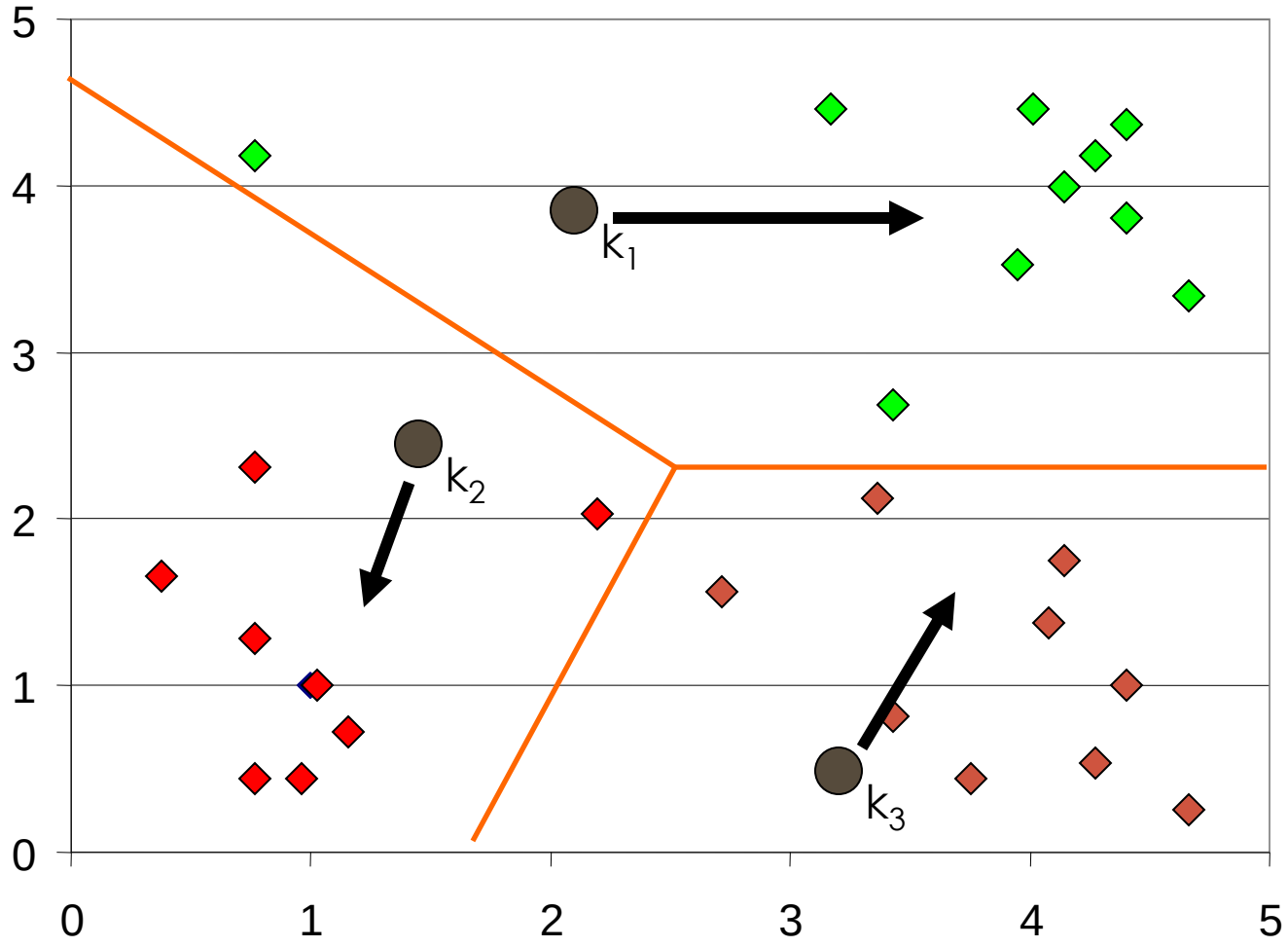
3. Update the location of prototypes k_j as the average of the instances x_i assigned to each cluster.

4. Go to 2, until clusters do not change

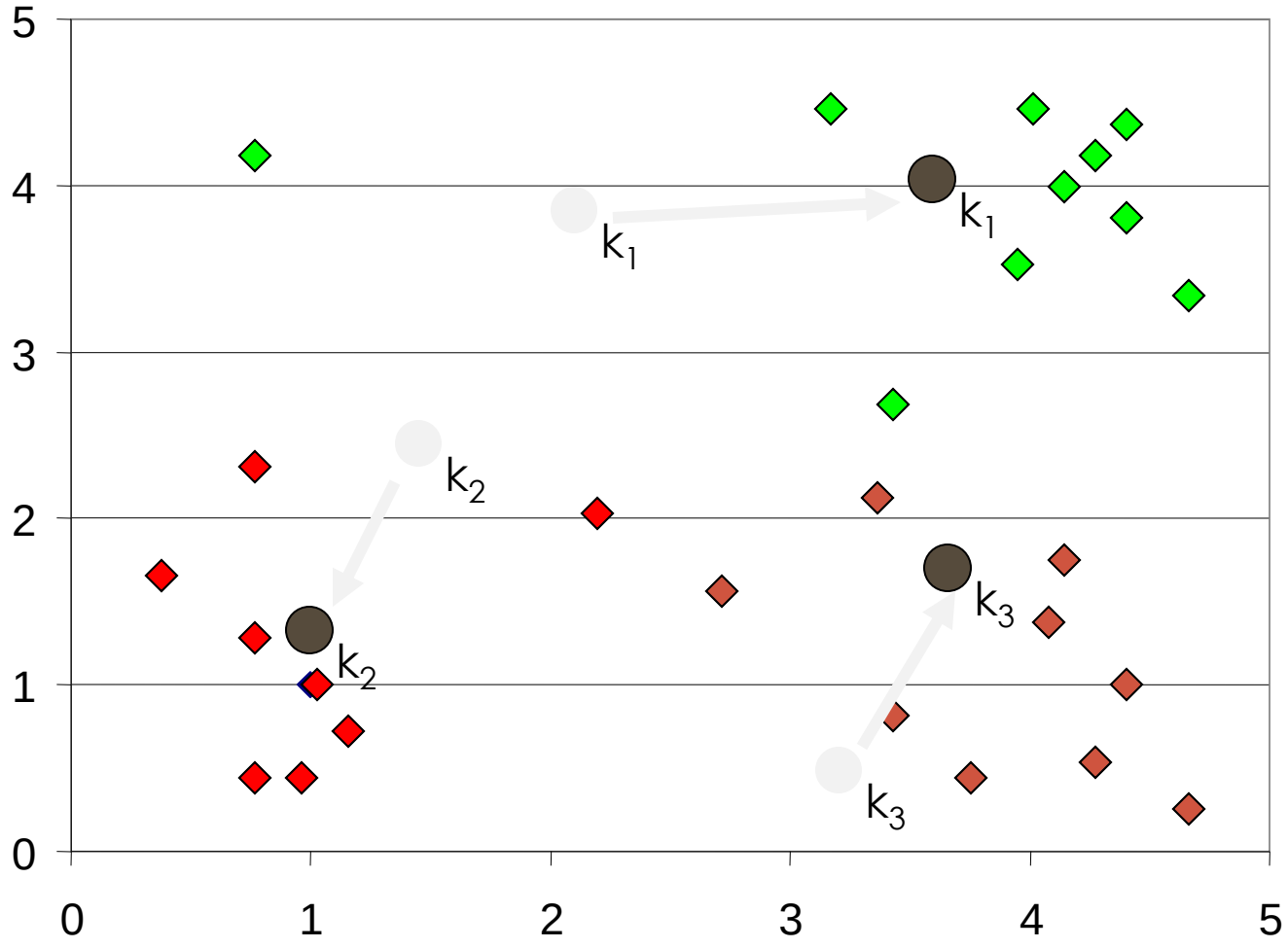
RANDOM INITIALIZATION OF PROTOTYPES



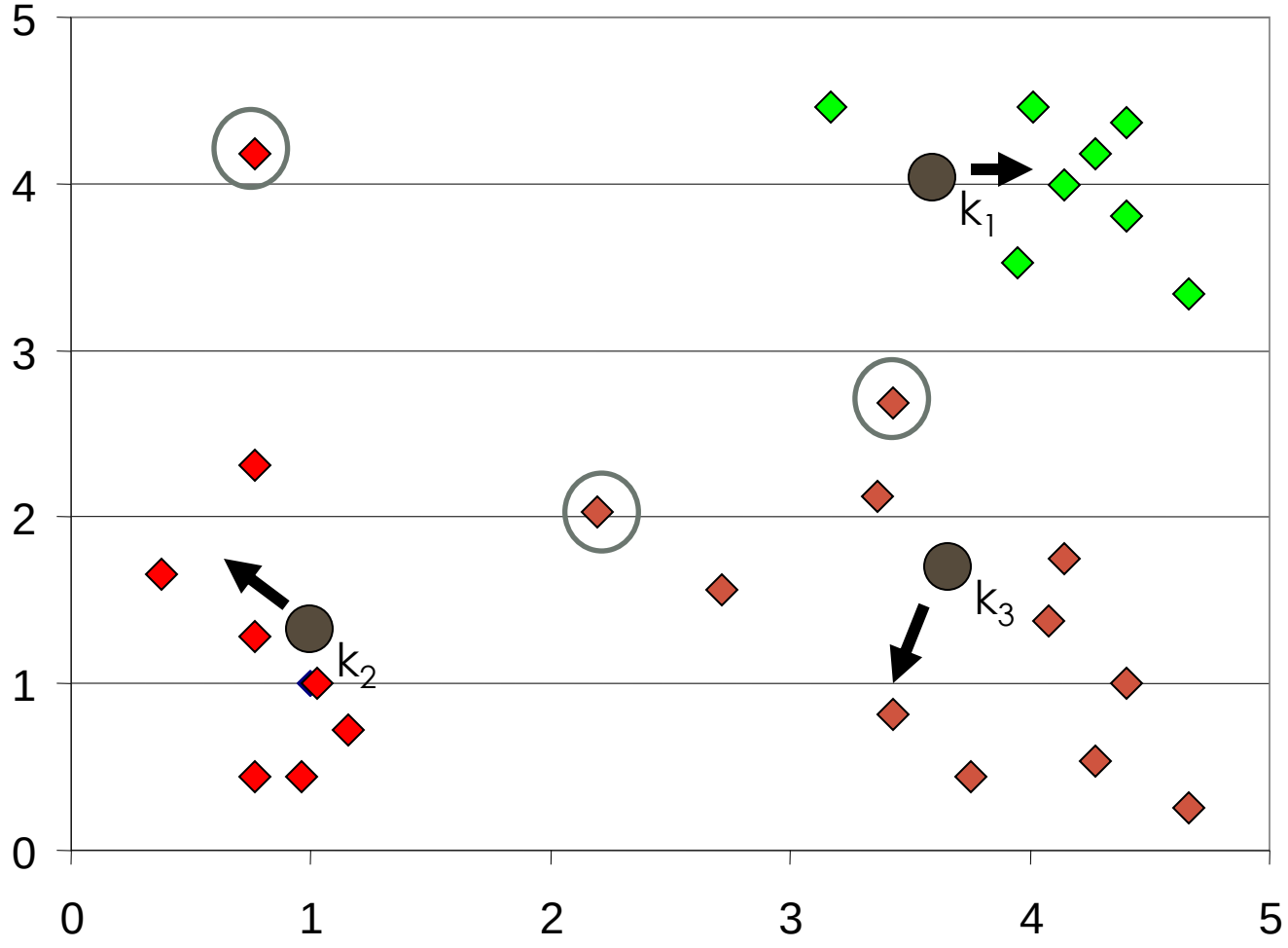
ASSIGNING INSTANCES TO CLOSEST PROTOTYPE



UPDATE PROTOTYPES (AVERAGE)



ASSIGNING INSTANCES TO CLOSEST PROTOTYPE



UPDATE PROTOTYPES (AVERAGE)