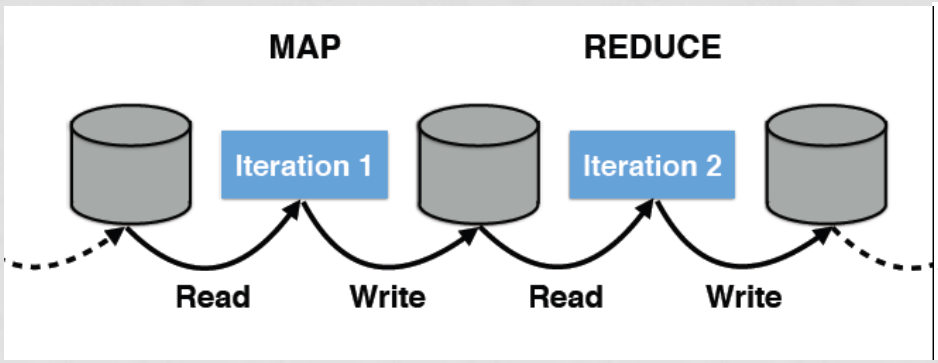
LARGE SCALE MACHINE LEARNING: SPARK

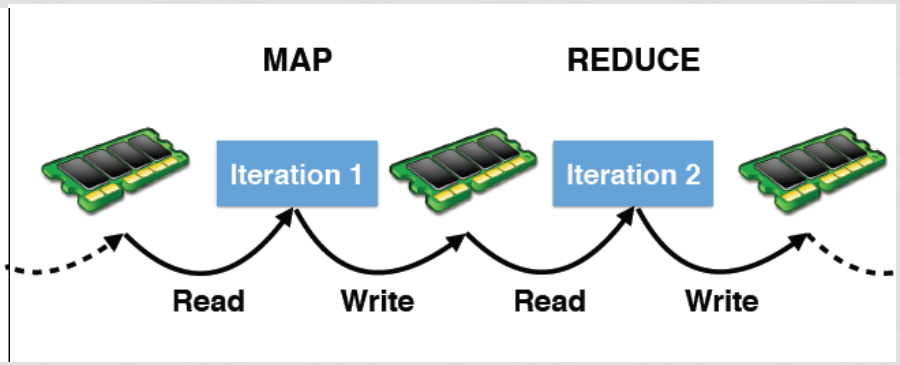
MAPREDUCE / HADOOP LIMITATIONS

- 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

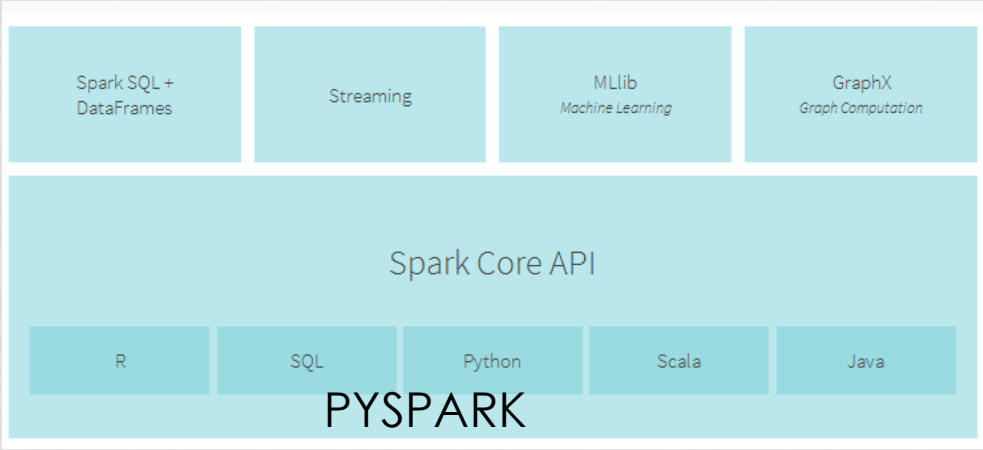


Spark



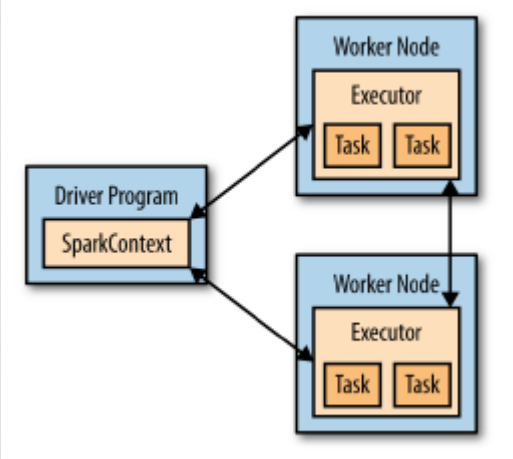
SPARK ECOSYSTEM

- 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
- In any case, Pyspark code and scala code look very similar, so translation is straightforward.



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



DISTRIBUTED DATA IN SPARK

- They contain **distributed data**, spread across **partitions.**
- **Transformations** on them are carried out in parallel (data parallelism).
- If something goes wrong with one of the workers, Spark recomputes (part of) the transformation automatically
- Distributed data types:
  - **RDD**: Resilient Distributed Dataset (Spark 1.x)
  - **DataFrame**: similar to R dataframes and Python/Pandas dataframes (Spark 2.x)
  - **DataSet**: similar to a dataframe, but with strongly typed columns. Only available in Scala (not in Pyspark)
- From Spark 2.x (July 2016), DataFrame and DataSet are the recommended distributed data structures, but RDDs are still in use.
- There are two machine learning libraries in Spark:
  - MLIB: older, it works with RDDs
  - ML: newer, it works with DataFrames

RDDS (RESILIENT DISTRIBUTED DATASET)

OLD DATA TYPE (~ SPARK 1.X)

- RDDs: they are list of elements, distributed among several partitions. E.g. a list of numbers, or a list of points, a list of strings, …
- [1, 2, 4, 5]
- [(1,2), (3, 4), (1,7)]
- [(“a”, 1), (“b”, 2), (“z”, 7)]

CREATING RDDS

- Note: *sc* is the SparkContext, and it is created when Spark is started (see notebook). It manages the computer cluster.
- 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 computed and 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 stores 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.']
```

RDDS ARE INMUTABLE

- In Python, objects are typically mutable: they can be modified. For instance, in a numeric matrix, we can change the value of position (3,1).
- In Spark, RDDs are inmutable: they cannot be modified. The only way to change an RDD is to replace the old one with a new one.
  - *numbers_rdd.map(lambda x: x**2): this applies the x^2 function, but the result is not stored anywhere (i.e. numbers_rdd is not changed).*
  - *numbers_rdd = numbers_rdd.map(lambda x: x**2): in this case, a numbers_rdd is created, and the old one is destroyed.*

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

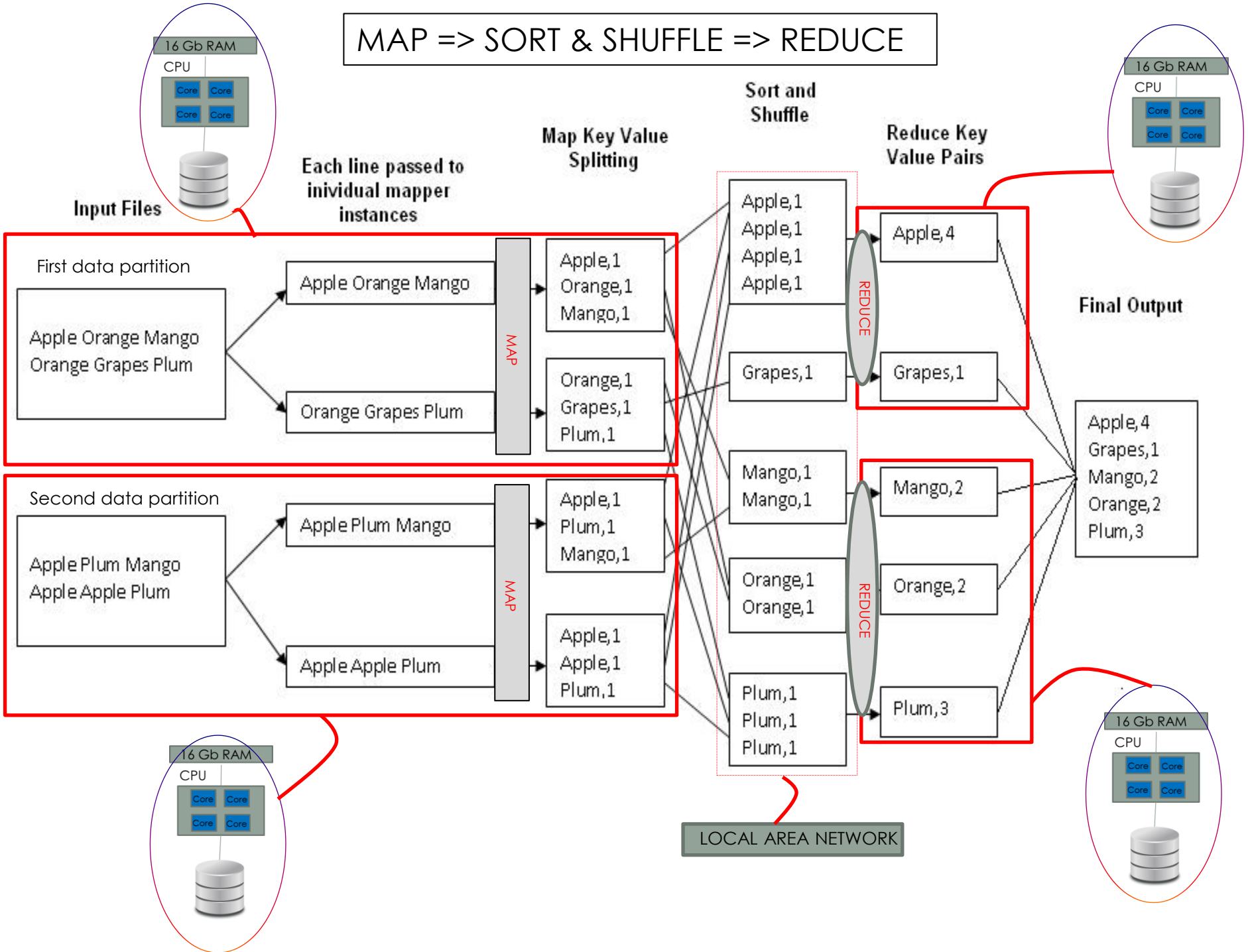
```
sc.stop()
```

```
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
Final Output
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
Apple,4
Grapes,1
Mango,2
Orange,2
Plum,3
LOCAL AREA NETWORK

DATAFRAME

CURRENT DISTRIBUTED DATA TYPE (~ SPARK 2.X)

DATAFRAMES

- A Spark dataframe is a distributed dataframe.
- Similar to R dataframes, or Pandas / Python dataframes (that is, two-dimensional matrices) but distributed among partitions (computers)

```
data_sd.show()
```

```
+-----+-------+----+-----+----+-----+-----+-----+------+---+-----+-------+------+-----+
|label|   CRIM|  ZN|INDUS|CHAS|  NOX|   RM|  AGE|   DIS|RAD|  TAX|PTRATIO|     B|LSTAT|
+-----+-------+----+-----+----+-----+-----+-----+------+---+-----+-------+------+-----+
| 24.0|0.00632|18.0| 2.31| 0.0|0.538|6.575| 65.2|  4.09|1.0|296.0|   15.3| 396.9| 4.98|
| 21.6|0.02731| 0.0| 7.07| 0.0|0.469|6.421| 78.9|4.9671|2.0|242.0|   17.8| 396.9| 9.14|
| 34.7|0.02729| 0.0| 7.07| 0.0|0.469|7.185| 61.1|4.9671|2.0|242.0|   17.8|392.83| 4.03|
| 33.4|0.03237| 0.0| 2.18| 0.0|0.458|6.998| 45.8|6.0622|3.0|222.0|   18.7|394.63| 2.94|
| 36.2|0.06905| 0.0| 2.18| 0.0|0.458|7.147| 54.2|6.0622|3.0|222.0|   18.7| 396.9| 5.33|
| 28.7|0.02985| 0.0| 2.18| 0.0|0.458| 6.43| 58.7|6.0622|3.0|222.0|   18.7|394.12| 5.21|
| 22.9|0.08829|12.5| 7.87| 0.0|0.524|6.012| 66.6|5.5605|5.0|311.0|   15.2| 395.6|12.43|
| 22.1|0.12579|12.5| 7.87| 0.0|0.524|6.172| 96.1|5.9505|5.0|311.0|   15.2| 396.9|19.15|
| 16.5|0.21124|12.5| 7.87| 0.0|0.524|5.631|100.0|6.0821|5.0|311.0|   15.2|386.63|29.93|
| 18.9|0.17004|12.5| 7.87| 0.0|0.524|6.004| 85.9|6.5921|5.0|311.0|   15.2|386.71| 17.1|
| 15.0|0.22489|12.5| 7.87| 0.0|0.524|6.377| 94.3|6.3467|5.0|311.0|   15.2|392.52|20.45|
| 18.9|0.11747|12.5| 7.87| 0.0|0.524|6.009| 82.9|6.2267|5.0|311.0|   15.2| 396.9|13.27|
| 21.7|0.09378|12.5| 7.87| 0.0|0.524|5.889| 39.0|5.4509|5.0|311.0|   15.2| 390.5|15.71|
| 20.4|0.62976| 0.0| 8.14| 0.0|0.538|5.949| 61.8|4.7075|4.0|307.0|   21.0| 396.9| 8.26|
| 18.2|0.63796| 0.0| 8.14| 0.0|0.538|6.096| 84.5|4.4619|4.0|307.0|   21.0|380.02|10.26|
| 19.9|0.62739| 0.0| 8.14| 0.0|0.538|5.834| 56.5|4.4986|4.0|307.0|   21.0|395.62| 8.47|
| 23.1|1.05393| 0.0| 8.14| 0.0|0.538|5.935| 29.3|4.4986|4.0|307.0|   21.0|386.85| 6.58|
| 17.5| 0.7842| 0.0| 8.14| 0.0|0.538| 5.99| 81.7|4.2579|4.0|307.0|   21.0|386.75|14.67|
| 20.2|0.80271| 0.0| 8.14| 0.0|0.538|5.456| 36.6|3.7965|4.0|307.0|   21.0|288.99|11.69|
| 18.2| 0.7258| 0.0| 8.14| 0.0|0.538|5.727| 69.5|3.7965|4.0|307.0|   21.0|390.95|11.28|
+-----+-------+----+-----+----+-----+-----+-----+------+---+-----+-------+------+-----+
only showing top 20 rows
```

CREATING DATAFRAMES

- Note: ***spark*** is the SparkSession (similar to the SparkContext for RDDs, but for dataframes). It handles the computer cluster
- Several ways:
  - Transforming an existing dataframe
  - Loading an external dataset or file in csv format.
    - *data_sd = spark.read.csv("boston.csv",header=True,inferSchema=True)*
  - Creating a Spark dataframe from a Pandas (Python) dataframe:
    - Next slide
  - Creating a Spark dataframe from a Spark RDD
  - …

For more information, check "**Untyped Dataset Operations (aka DataFrame Operations)**": http://spark.apache.org/docs/latest/sql-programming-guide.html#untyped-dataset-operations-aka-dataframe-operations

CREATING A SPARK DATAFRAME FROM A PANDAS/PYTHON DATAFRAME

- Note: the Pandas / Python dataframe is local. The Spark dataframe is distributed.

```
from sklearn.datasets import load_iris
# iris contains a matrix of values
iris = load_iris()
# Now we transform this matrix into a Pandas dataframe
iris_pd = pd.DataFrame(iris.data)
# Finally, we convert it to a Spark dataframe
iris_sd = spark.createDataFrame(iris_pd)
```

```
iris_sd.show()
```

Show is an action that displays the first rows of the distributed dataframe.

```
+---+---+---+---+
|  0|  1|  2|  3|
+---+---+---+---+
|5.1|3.5|1.4|0.2|
|4.9|3.0|1.4|0.2|
|4.7|3.2|1.3|0.2|
|4.6|3.1|1.5|0.2|
|5.0|3.6|1.4|0.2|
```

- printSchema

```
# Prints the schema (columns and their types)
iris_sd.printSchema()
```

```
root
 |-- 0: double (nullable = true)
 |-- 1: double (nullable = true)
 |-- 2: double (nullable = true)
 |-- 3: double (nullable = true)
```

TRANSFORMATION: SELECT

- *Select* is a *transformation* that selects columns or creates new columns.

```
iris_0_sd = iris_sd.select("0")
iris_0_sd.show()
```

```
+---+
|  0|
+---+
|5.1|
|4.9|
|4.7|
|4.6|
|5.0|
|5.4|
```

```
iris_0a_sd = iris_sd.select(iris_sd["0"], (1+iris_sd["0"]).alias("a"))
iris_0a_sd.show()
```

```
+---+---+
|  0|  a|
+---+---+
|5.1|6.1|
|4.9|5.9|
|4.7|5.7|
|4.6|5.6|
```

TRANSFORMATIONS: FILTER

- Filter is a transformation that selects the rows in the dataframe that fulfill the condition.

```
iris_bis_sd = iris_sd.filter((iris_sd[0] > 4.5) & (iris_sd[1]<2.5))
iris_bis_sd.show()
```

```
+---+---+---+---+
|  0|  1|  2|  3|
+---+---+---+---+
|5.5|2.3|4.0|1.3|
|4.9|2.4|3.3|1.0|
|5.0|2.0|3.5|1.0|
|6.0|2.2|4.0|1.0|
|6.2|2.2|4.5|1.5|
|5.5|2.4|3.8|1.1|
|5.5|2.4|3.7|1.0|
|6.3|2.3|4.4|1.3|
|5.0|2.3|3.3|1.0|
|6.0|2.2|5.0|1.5|
+---+---+---+---+
```

TRANSFORMATIONS: WITHCOLUMN

- Adds a new column

```
iris_wc_sd = iris_sd.withColumn('larger2', iris_sd["0"]>4.5)
iris_wc_sd.show()
```

```
+---+---+---+---+-------+
|  0|  1|  2|  3|larger2|
+---+---+---+---+-------+
|5.1|3.5|1.4|0.2|   true|
|4.9|3.0|1.4|0.2|   true|
|4.7|3.2|1.3|0.2|   true|
|4.6|3.1|1.5|0.2|   true|
|5.0|3.6|1.4|0.2|   true|
|5.4|3.9|1.7|0.4|   true|
|4.6|3.4|1.4|0.3|   true|
|5.0|3.4|1.5|0.2|   true|
|4.4|2.9|1.4|0.2|  false|
|4.9|3.1|1.5|0.1|   true|
|5.4|3.7|1.5|0.2|   true|
|4.8|3.4|1.6|0.2|   true|
|4.8|3.0|1.4|0.1|   true|
|4.3|3.0|1.1|0.1|  false|
|5.8|4.0|1.2|0.2|   true|
|5.7|4.4|1.5|0.4|   true|
|5.4|3.9|1.3|0.4|   true|
|5.1|3.5|1.4|0.3|   true|
|5.7|3.8|1.7|0.3|   true|
|5.1|3.8|1.5|0.3|   true|
+---+---+---+---+-------+
only showing top 20 rows
```

TRANSFORMATIONS: GROUPBY

- It splits the dataframe into as many groups as values in "larger2" (2 in this case: true and false).
- Count() is also a transformation that counts the number of rows in each of the groups.

```
iris_g_sd = iris_wc_sd.groupBy("larger2")
iris_g_sd.count().show()
```

```
+-------+-----+
|larger2|count|
+-------+-----+
|   true|  145|
|  false|    5|
+-------+-----+
```

PERSISTENCE

- It stores the dataframe into memory. Next time it is used, it won't be recomputed.

```
iris_wc_sd.cache()
```

```
DataFrame[0: double, 1: double, 2: double, 3: double, larger2: boolean]
```

ACTION: COLLECT

- Collects the spark distributed dataframe into a local Python list.

```
import pprint
# iris_sd is a Spark DISTRIBUTE dataframe
# iris_python is a LOCAL python data list
print "iris_sd is of type: {}".format(type(iris_sd))
iris_python = iris_sd.collect()
print("iris_python is of type: {}".format(type(iris_python)))
pprint.pprint(iris_python[0:3])
```

```
iris_sd is of type: <class 'pyspark.sql.dataframe.DataFrame'>
iris_python is of type: <type 'list'>
[Row(0=5.1, 1=3.5, 2=1.4, 3=0.2),
 Row(0=4.9, 1=3.0, 2=1.4, 3=0.2),
 Row(0=4.7, 1=3.2, 2=1.3, 3=0.2)]
```

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

MACHINE LEARN ING WITH MLLIB

RDD-BASED (~ SPARK 1.X)

SPARK.MLLIB

- In order to use the RDD-based MLLIB library, the RDD must be a list of **LabeledPoint**s
- The **LabeledPoint** datatype is a way to represent instances. It is made of two parts: label (class, output attribute) and features (input attributes).

```
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])
```

- Example of an RDD for training:

[**LabeledPoint**(1.0, **DenseVector**(1.0, 0.0, 3.0)),
LabeledPoint (2.0, **DenseVector**(0.5, 0.7, 1.8)),
LabeledPoint (0.8, **DenseVector**(0.1, 0.5, 2.1))]

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]
```

MACHINE LEARNING WITH ML

DATAFRAME-BASED (~ SPARK 2.X)

The following cells prepare the dataframe for ML use

The algorithms in Spark ML library need a dataframe with just two columns: the first one (typically named *features*) must contain a matrix with the input attributes, the second one must contain the output attribute (typically named *label*). In order to do that, we must go back to the version of the dataframe in the old RDD format (*.rdd*), and transform it into a list of tuples (label, input_features). In this notebook, we assume that all input attributes are real numbers. Spark forces to represent vectors of real numbers as *dense vectors*.

```
from pyspark.ml.linalg import DenseVector
data_rdd = data_sd.rdd.map(lambda x: (x[0], DenseVector(x[1:])))
data_rdd.take(2)
```

This is how the first rows of the dataframe look like.

```
data_sd.show(truncate=False)
```

```
+-----+-----------------------------------------------------------------------------+
|label|features                                                                     |
+-----+-----------------------------------------------------------------------------+
|21.6 |[0.02731,0.0,7.07,0.0,0.469,6.421,78.9,4.9671,2.0,242.0,17.8,396.9,9.14]     |
|28.7 |[0.02985,0.0,2.18,0.0,0.458,6.43,58.7,6.0622,3.0,222.0,18.7,394.12,5.21]     |
|18.9 |[0.17004,12.5,7.87,0.0,0.524,6.004,85.9,6.5921,5.0,311.0,15.2,386.71,17.1]   |
|20.4 |[0.62976,0.0,8.14,0.0,0.538,5.949,61.8,4.7075,4.0,307.0,21.0,396.9,8.26]     |
|17.5 |[0.7842,0.0,8.14,0.0,0.538,5.99,81.7,4.2579,4.0,307.0,21.0,386.75,14.67]     |
|19.6 |[0.85204,0.0,8.14,0.0,0.538,5.965,89.2,4.0123,4.0,307.0,21.0,392.53,13.83]   |
|13.9 |[0.84054,0.0,8.14,0.0,0.538,5.599,85.7,4.4546,4.0,307.0,21.0,303.42,16.51]   |
|21.0 |[1.00245,0.0,8.14,0.0,0.538,6.674,87.3,4.239,4.0,307.0,21.0,380.23,11.98]    |
|13.1 |[1.15172,0.0,8.14,0.0,0.538,5.701,95.0,3.7872,4.0,307.0,21.0,358.77,18.35]   |
|21.0 |[0.08014,0.0,5.96,0.0,0.499,5.85,41.5,3.9342,5.0,279.0,19.2,396.9,8.77]      |
|26.6 |[0.12744,0.0,6.91,0.0,0.448,6.77,2.9,5.7209,3.0,233.0,17.9,385.41,4.84]      |
|19.3 |[0.17142,0.0,6.91,0.0,0.448,5.682,33.8,5.1004,3.0,233.0,17.9,396.9,10.21]    |
|19.4 |[0.21977,0.0,6.91,0.0,0.448,5.602,62.0,6.0877,3.0,233.0,17.9,396.9,16.2]     |
|23.4 |[0.04981,21.0,5.64,0.0,0.439,5.998,21.4,6.8147,4.0,243.0,16.8,396.9,8.43]    |
|31.6 |[0.01432,100.0,1.32,0.0,0.411,6.816,40.5,8.3248,5.0,256.0,15.1,392.9,3.95]   |
|16.0 |[0.17171,25.0,5.13,0.0,0.453,5.966,93.4,6.8185,8.0,284.0,19.7,378.08,14.44]  |
|23.5 |[0.03584,80.0,3.37,0.0,0.398,6.29,17.8,6.6115,4.0,337.0,16.1,396.9,4.67]     |
|20.9 |[0.12816,12.5,6.07,0.0,0.409,5.885,33.0,6.498,4.0,345.0,18.9,396.9,8.79]     |
```