Apache Spark

Lorenzo Di Gaetano

THE CONTRACTOR IS ACTING UNDER A FRAMEWORK CONTRACT CONCLUDED WITH THE COMMISSION

What is Apache Spark?

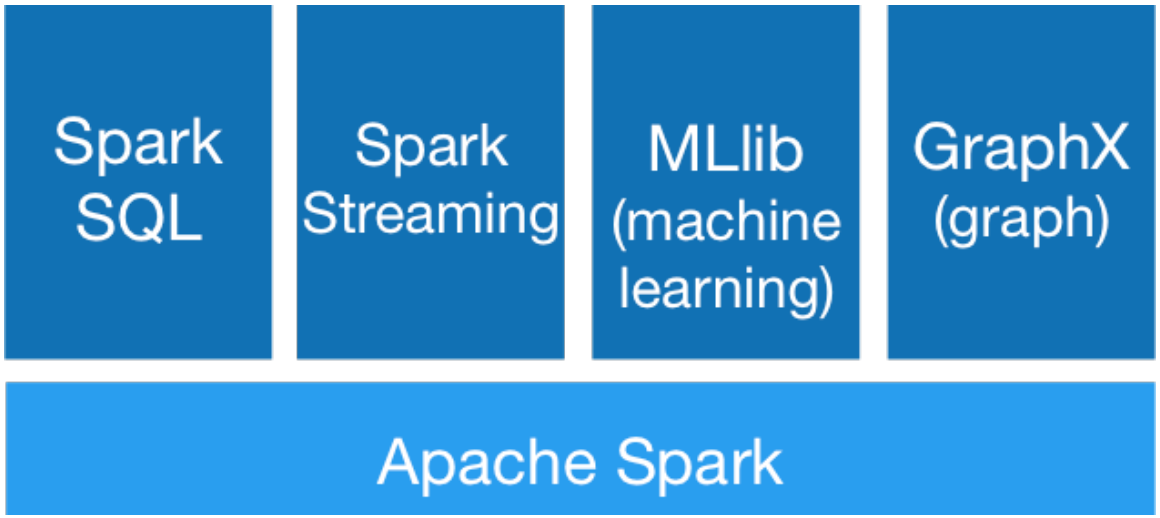
- A general purpose framework for big data processing
- It interfaces with many distributed file systems, such as Hdfs (Hadoop Distributed File System), Amazon S3, Apache Cassandra and many others
- 100 times faster than Hadoop for in-memory computation

Multilanguage API

- You can write applications in various languages
  - **Java**
  - **Python**
  - **Scala**
  - **R**

- In the context of this course we will consider Python

Built-in Libraries



Third party libraries

- Many third party libraries are available
  - http://spark-packages.org/

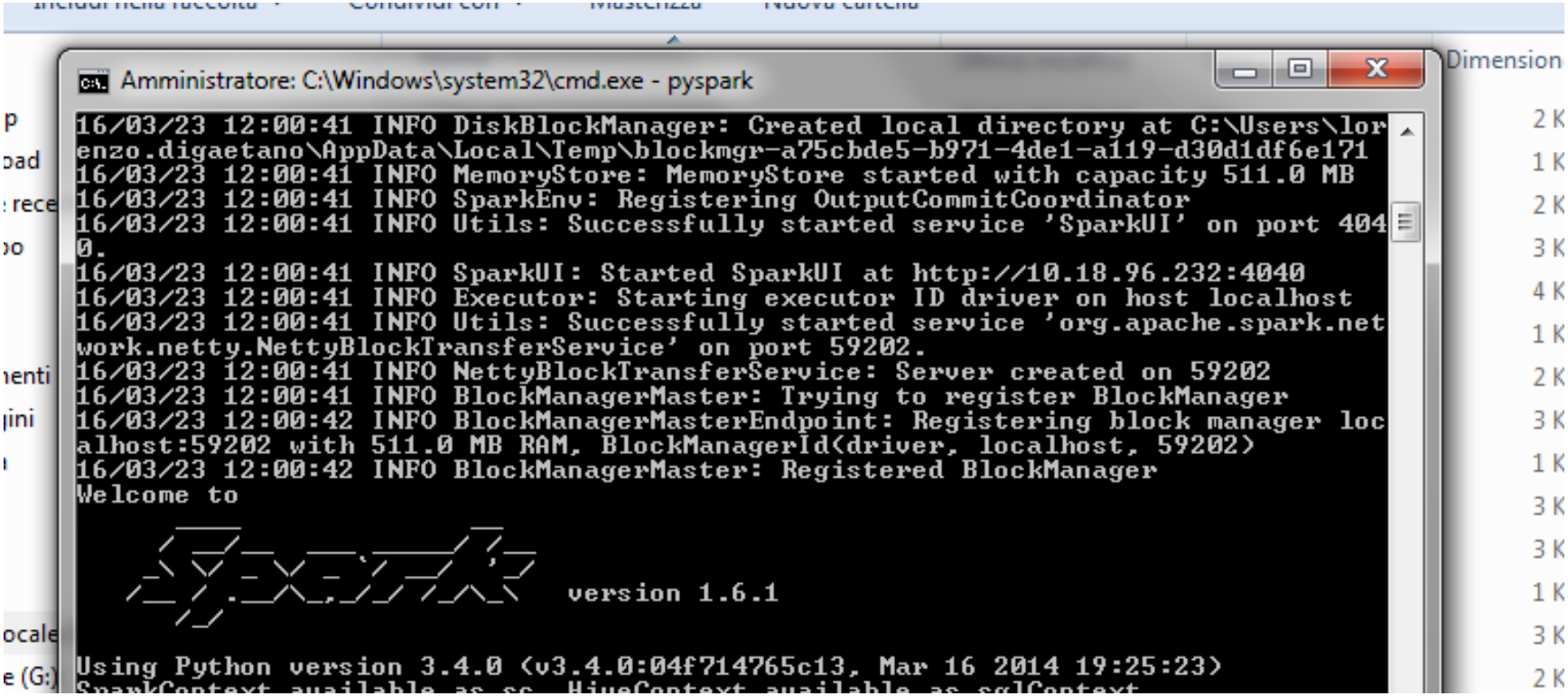
- We used spark-csv in our examples
- We will see later how to use an external jar on our application

Running Spark

- Once you correctly installed spark you can use it in two ways.
  - **spark-submit:** it's the CLI command you can use to launch python spark applications
  - **pyspark:** used to launch an interactive python shell.

European Commission

PySpark up and running!



SparkContext

- Every Spark application starts from the SparkContext
  - **Entry point to the Spark API**

- Using spark-submit you have to manually create a *SparkContext* object in your code
- Using the pyspark interactive shell a SparkContext is automatically available in a variable called *sc*

What is a Spark Application?

- A Spark application is made of a *Driver Program*
- The *Driver Program* runs the the main function which executes parallel operations on the cluster

RDD

- Spark works on RDD – Resilient Distributed Dataset
- A RDD is a collection of elements partitioned in every cluster node. Spark operates in parallel on them
- Spark programs consist in performing operations on RDDs

How to create a RDD

- There are three ways:
  - Reading an external dataset on the filesystem
  - From data in memory (i.e. parallelizing a pre-existent collection on the driver program)
  - From another RDD

RDD from external datasets

- You can create RDD from various kind of external datasets like local filesystem, HDFS, Cassandra etc...
- For example we can read a text file, obtaining a collection of lines:

```
rdd = sc.textFile("textfile.txt")
```

- The level of parallelism is given by the number of partitions in which the file is split on the file system

RDD from collections

- You can create a RDD from a collection using parallelize() method on SparkContext

```
data = [1,2,3,4,5]
rdd = sc.parallelize(data)
```

- Then you can operate in parallel on rdd

Operations on RDD

- There are two types of operations:
  - **Transformations:** Take an existing RDD and return another RDD
  - **Actions:** Take an existing RDD and return a value

Transformations

- For example, *map* is a transformation that takes all elements of the dataset, pass them to a function and returns another RDD with the results

```
resultRDD = originalRDD.map(myFunction)
```

Actions

- For example, count is an action. It returns the number of elements in the rdd

```
c= rdd.count()
```

- *reduce* is another action. It aggregates all elements of the RDD using a function and returns the result to the driver program

```
result = rdd.reduce(function)
```

Lazy Execution

- Operations are not processed until an action is performed

- A sequence of trasformations can be called on a RDD but the actual processing takes place when an action is called

Passing functions to Spark

- There are three ways
  - **lambda expressions:** used to pass simple expressions which return a value (anonymous functions)
  - **Local function definition:** for more complex code
  - **Functions inside a module**

Example: lambda functions

```
sc = SparkContext()
upper = sc.textFile("textFile.txt") \
        .map(lambda s: s.toUpper())
```

map(f) applies the function f to every single record in the RDD and returns the new transformed RDD

Example: named functions

```
def upcase(s):
       return s.toUpper()

sc = SparkContext()
upper = sc.textFile("textFile.txt") \
       .map(upcase)
```

Getting and printing elements

- *rdd.collect()* is an action that retrieves all elements in an RDD
  - **May cause an out of memory error for big datasets!**
- *rdd.take(n)* is an action that returns the first n elements of the rdd
- *rdd.take(n).foreach(println)* prints the first *n* elements of the rdd

Making RDD persistent

- Spark can persist (or cache) a dataset in memory during operations
- If you persist an RDD, every node stores the partitions it elaborates in memory and reuses them in other actions which will run faster
- You can persist an RDD using the *persist()* or *cache()* methods

Removing data

- Spark perfoms a sort of garbage collection and automatically deletes old data partitions
- We can manually remove an RDD using the *unpersist()* method

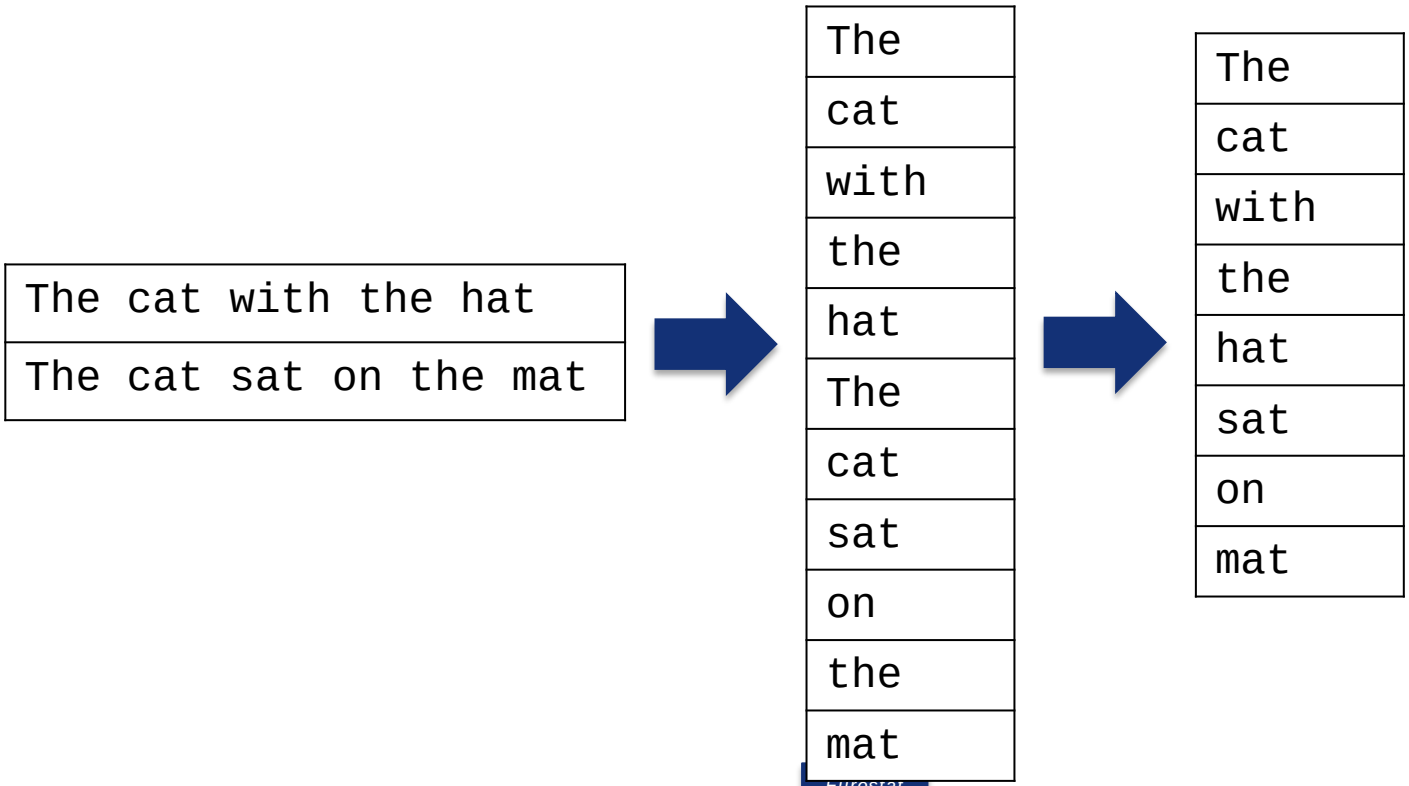
RDD Operations

- Single-RDD transformations
  - **flatMap:** maps one element to multiple elements
  - **distinct:** removes duplicates
  - **sortBy:** sort by the given function

- Multi-RDD transformation
  - **union:** creates a new RDD with all elements of the two initial RDDs
  - **intersection:** creates a new RDD with common elements only
  - **zip:** pair elements of the two RDDs in order (not a join!)

Example: flatMap and distinct

```
sc.textFile("textFile.txt") \
        .flatMap(lambda s: s.split(" ")).distinct
```



Key-value Pairs

- Some special operations in Spark works only on RDDs in a special format
- Key-Value pairs are RDDs where each element is a two-element tuple
- Key and value can be of any type, including complex types (i.e. they can also be tuples…)
- Common data processing operations can be applied only to Key-Value RDDs

Creating Key-Value RDDs

- Getting data in key-value format is the first step before applying pair-only functions
- Normally done with a map operation

```
sc.textFile("textFile.txt") \
        .map(lambda s: s.split("\t")) \
        .map(lambda row: (row[0],row[1]) )
```

001\tJohn\t35
002\tAlice\t41
003\tBob\t72
004\tCarla\t54

(001,”John”)
(002,”Alice”)
(003,”Bob”)
(004,”Carla”)

Creating Key-Value RDDs

- Key and/or value can be composite

```
sc.textFile("textFile.txt") \
        .map(lambda s: s.split("\t")) \
        .map(lambda row: (row[0],(row[1),row[2])) )
```

001\tJohn\t35
002\tAlice\t41
003\tBob\t72
004\tCarla\t54

(001,(”John”,35))
(002,(”Alice”,41))
(003,(”Bob”,72))
(004,(”Carla”,54))

Map-Reduce in Spark

- Map-Reduce can be implemented in Spark using pair RDDs
- More flexible than standard Map-Reduce because different operations can be used
- Map phase – maps one input pair to one or more output pairs
  - **Input (k,v) -> Output (k1,v1)**
  - **map, flatMap, filter, keyBy …**
- Reduce phase – consolidates multiple records obtained from map
  - **Input (v1,v2) -> Output (v3)**
  - **reduceByKey, sortByKey, …**

Example: word counting
with lambda functions

```
sc = SparkContext()
counts = sc.textFile("textFile.txt") \
       .flatMap(lambda s: s.split(" ")) \
       .map(lambda x: (x,1)) \
       .reduceByKey(lambda v1,v2: v1+v2)

output =counts.collect()
```

functions passed to reduceByKey must be commutative and associative

Example: word counting
with local function definition

```
import re

def extractWords(s):
        re.sub("[^a-zA-Z0-9 ]" , "" , s)
        words = s.toUpper().split(" ")
        return words;

sc = SparkContext()
counts = sc.textFile("textFile.txt") \
        .flatMap(extractWords) \
        .map(lambda x: (x,1)) \
        .reduceByKey(lambda v1,v2: v1+v2)

output =counts.collect()
```

Other Key-Value RDD Operations

- **join**: starts from two RDDs and returns an RDD containing all pairs with matching keys
- **sortByKey**: returns a new RDD with elements ordered by key
- **groupByKey**: returns a new RDD with all elements sharing a common key grouped in a same element

Example: groupByKey

```
sc.textFile("textFile.txt") \
        .map(lambda s: s.split("\t")) \
        .map(lambda row: (row[0],row[1]) ) \
        .groupByKey
```

001\tJohn
002\tAlice
001\tBob
004\tCarla
002\tMaria
002\tTom

(001,[”John”, ”Bob”])
(002,[”Alice”, ”Maria”, ”Tom”])
(004,[”Carla”])

Shared Variables

- We always must keep in mind that when we pass a function to a spark operation, this function is executed on separate cluster nodes. Every node receives a COPY of the variable inside the function

- Every change to the local value of the variable is not propagated to the driver program

Shared Variables

- To solve the problem spark offers two types of shared variables:
  - **Broadcast variables**
  - **Accumulators**

Broadcast variables

- Instead of creating a copy of the variable for each machine, broadcast variables allows the programmer to keep a cached read-only variable in every machine

- We can create a broadcast variable with the *SparkContext.broadcast(var)* method, which returns the reference of the broadcast variable

Accumulators

- Accumulators are used to keep some kind of shared «counter» across the machines. It's a special variable which can be «added»
- We can create an accumulator with *SparkContext.accumulator(var)* method
- Once the accumulator is created we can add values to it with the *add()* method

SparkSQL and DataFrames

- SparkSQL is the spark module for structured data processing
- DataFrame API is one of the ways to interact with SparkSQL

DataFrames

- A DataFrame is a collection of data organized into columns
- Similar to tables in relational databases
- Can be created from various sources: structured data files, Hive Tables, external db, csv etc…

Creating a DataFrame

- Given a SparkContext (sc), the first thing to do is to create a SQLContext

```
sqlContext = SQLContext(sc)
```

- Then read the data, for example in JSON format:

```
df = sqlContext.read.json('file.json')
```

Creating a DataFrame from csv file

- We now see how we can load a csv file into a DataFrame using an external library called Spark csv: https://github.com/databricks/spark-csv

- We need to download two files: *spark-csv_2.11-1.4.0.jar* and *commons-csv-1.2.jar*

Creating a DataFrame from csv file

- When launching our code with spark-submit we have to add external jars to the classpath, assuming we put the jars into *lib* directory of our project:

```
spark-submit --jars lib/spark-csv_2.11-1.4.0.jar,lib/commons-csv-1.2.jar myCode.py
```

Creating a DataFrame from csv file

- Then we read our file into a DataFrame

```
df = sqlContext.read.format('com.databricks.spark.csv').options(header='true',
nullValue='NULL').load(myCsvFile, inferschema = ‘true’)
```

- Note the *inferschema = true* option. With this option activated, Spark tries to guess the format of every field
- We can specify the schema manually creating a *StructType* array

Creating a DataFrame from csv file

- Create the schema structure

```
customSchema = StructType([ \
          StructField("field1", IntegerType(), True),
          StructField("field2", FloatType(), True), \
          StructField("field3", TimestampType(), True), \
          StructField("field4", StringType(), True) \
])
```

- And then pass it to our load function as an option

```
df = sqlContext.read.format('com.databricks.spark.csv').options(header='true',
nullValue='NULL').load(myFile, schema = customSchema)
```

Queries on dataframes

- All queries on dataFrames return another dataFrame
- Example of query methods:
  - **select –** returns a DF with one or more columns from the original one
  - **distinct –** returns a DF with distinct elements from the original one
  - **join –** joins the original DF with another one
  - **limit –** returns the original DF with only the first n rows
  - **filter –** returns the original DF with only the rows matching a provided condition
- Queries can be chained.

Dataframes and RDD

- Dataframes are built on RDD
  - RDD contains row objects
  - We can use the rdd method to get the rdd from DF

```
myRDD = myDF.rdd
```

myDF

id	name	age
1	John Doe	30
2	Patrick Obre	23
3	George Fish	45
4	Alex Web	19
5	Angela Goss	22

myRDD

Row[1,John Doe,30]

Row[2,Patrick Obre,23]

Row[3,George Fish,45]

Row[4,Alex Web,19]

Row[5,Angela Goss,22]

Extracting data from Rows

```
myRDD = myDF.map(lambda row:(row.id,row.name))
myRDDbyID = myRDD.groupByKey()
```

id	name	age
1	John Doe	30
2	Patrick Obre	23
3	George Fish	45
3	Alex Web	19
5	Angela Goss	22

(1,John Doe)
(2,Patrick Obre)
(3,George Fish)
(3,Alex Web)
(5,Angela Goss)

(1,[John Doe])	
(2,[Patrick Obre])	
(3,[George Fish, Alex Web])	
(5,Angela Goss)	

Example operations on DataFrames

- To show the content of the DataFrame
  - `df.show()`
- To print the Schema of the DataFrame
  - `df.printSchema()`
- To select a column
  - `df.select('columnName').show()`
- To filter by some parameter
  - `df.filter(df['columnName'] > N).show()`

A complete example: group and avg

- We have a table like this

```
+----------+------+-------+--------------+------------------+-----------+-------+
|codice_pdv|prdkey|   week|vendite_valore|vendite_confezioni|flag_sconto| sconto|
+----------+------+-------+--------------+------------------+-----------+-------+
|      4567|730716|2015009|   190.8500061|             196.0|        0.0|0.98991|
|      4567|730716|2013048|   174.6000061|             153.0|       null|   null|
|      4567|730716|2015008|   160.6000061|             165.0|        0.0|0.98951|
|      4567|730716|2015006|  171.92999268|             176.0|        0.0|0.99329|
|      4567|730716|2015005|  209.47999573|             213.0|        0.0|1.00447|
+----------+------+-------+--------------+------------------+-----------+-------+
```

- We want to group by prdkey and calculate the average vendite_valore for every group

Preparing the schema

```
sc = SparkContext("local", "PrezzoMedio")

sqlContext = SQLContext(sc)

customSchema = StructType([ \
    StructField("codice_pdv", IntegerType(), True), \
    StructField("prdkey", IntegerType(), True), \
    StructField("week", IntegerType(), True), \
    StructField("vendite_valore", FloatType(), True), \
    StructField("vendite_confezioni", FloatType(), True), \
    StructField("data_ins", TimestampType(), True), \
    StructField("num_riga", FloatType(), True), \
    StructField("flag_sconto", FloatType(), True), \
    StructField("sconto", FloatType(), True), \
    StructField("idricefile", FloatType(), True), \
    StructField("iddettzipfile", FloatType(), True), \
    StructField("uteins", StringType(), True), \
    StructField("datamod", TimestampType(), True), \
    StructField("utemod", StringType(), True) \
    ])
```

Read data and do the job

```
myCsvFile = "C:\TransFatt1000.csv"

df = sqlContext.read.format('com.databricks.spark.csv').options(header='true',
nullValue='NULL').load(myCsvFile, schema = customSchema)

t0 = time()

averageForProductKey = df.groupBy("prdkey").avg("vendite_valore").collect()

tt = time() - t0

print(averageForProductKey)

print ("Query performed in {:03.2f} seconds.".format(tt))
```

With parquet table on HDFS

```
myParquetTable = "hdfs://bigdata-mnn.hcl.istat.it:8020///user/hive/warehouse/scanner_data.db/trans_fatt_p«

df = sqlContext.read.load(myParquetTable, schema = customSchema)

t0 = time()

averageForProductKey = df.groupBy("prdkey").avg("vendite_valore").collect()

tt = time() - t0

print(averageForProductKey)

print ("Query performed in {:03.2f} seconds.".format(tt))
```

References

- http://spark.apache.org/
- http://spark.apache.org/sql/
- http://spark.apache.org/docs/latest/sql-programming-guide.html