



European
Commission

Apache Spark

Lorenzo Di Gaetano

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

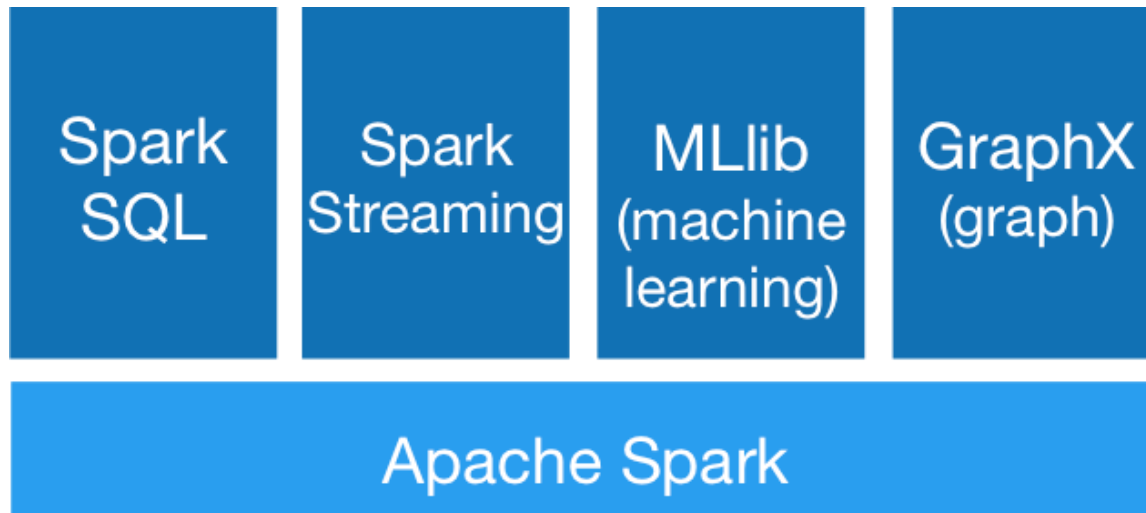
What is Apache Spark?

- 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

Getting started with Spark!

- Spark can run on Hadoop, on the cloud and can be also be installed as a standalone application on your PC
- Spark comes pre-installed in all the major Hadoop distributions
- We will consider the stand-alone installation in this course

Prerequisites

- Python (for using pySpark) and Java must be installed.
- All environment variables must be correctly set.

Local installation on Windows (1/2)

- Download Spark at <https://spark.apache.com/downloads.html>
- Download “Pre-built for Hadoop 2.6 and later”
- Unzip it
- Set the SPARK_HOME environment variable to point where you unzipped Spark
- Add %SPARK_HOME%/bin to your PATH

Local installation on Windows (2/2)

- Download <http://public-repo-1.hortonworks.com/hdp-win-alpha/winutils.exe>
- Move it in %SPARK_HOME%/bin
- Create an environment variable:
HADOOP_HOME=%SPARK_HOME%
- Now test it: open a terminal and launch pySpark

PySpark up and running!

```
Amministratore: C:\Windows\system32\cmd.exe - pyspark
16/03/23 12:00:41 INFO DiskBlockManager: Created local directory at C:\Users\lor
enzo.digaetano\AppData\Local\Temp\blockmgr-a75cbde5-b971-4de1-a119-d30d1df6e171
16/03/23 12:00:41 INFO MemoryStore: MemoryStore started with capacity 511.0 MB
16/03/23 12:00:41 INFO SparkEnv: Registering OutputCommitCoordinator
16/03/23 12:00:41 INFO Utils: Successfully started service 'SparkUI' on port 404
0.
16/03/23 12:00:41 INFO SparkUI: Started SparkUI at http://10.18.96.232:4040
16/03/23 12:00:41 INFO Executor: Starting executor ID driver on host localhost
16/03/23 12:00:41 INFO Utils: Successfully started service 'org.apache.spark.net
work.netty.NettyBlockTransferService' on port 59202.
16/03/23 12:00:41 INFO NettyBlockTransferService: Server created on 59202
16/03/23 12:00:41 INFO BlockManagerMaster: Trying to register BlockManager
16/03/23 12:00:42 INFO BlockManagerMasterEndpoint: Registering block manager loc
alhost:59202 with 511.0 MB RAM, BlockManagerId(driver, localhost, 59202)
16/03/23 12:00:42 INFO BlockManagerMaster: Registered BlockManager
Welcome to
Spark version 1.6.1
Using Python version 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014 19:25:23)
SparkContext available as sc, HiveContext available as sqlContext
```

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.

Running Spark

- 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 Filesystem
- A RDD is a collection of elements partitioned in every cluster node. Spark operates in parallel on them

RDD

- Every RDD is created from a file on Hadoop filesystem
- They can be made persistent in memory

How to create a RDD

- There are two ways:
 - Parallelizing a pre-existent collection on the driver program
 - Reading an external dataset on the filesystem

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

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")
```

Operations on RDD

- There are two types of operations:
 - **Transformations:** Take an existing dataset and return another dataset
 - **Actions:** Take an existing dataset 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, *reduce* is an action. It aggregates all elements of the RDD using a function and returns the result to the driver program

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

Passing functions to Spark

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

Example: word counting

```
def countWords(s):  
    words = s.split(" ")  
    return len(words)  
  
sc = SparkContext(...)  
sc.textFile("textFile.txt").map(countWords)
```

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

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

Printing elements

- When working on a single machine we can simply use `rdd.foreach(println)`
- But when we work in cluster mode, the `println` will be executed on the executor `stdout`, so we will not see anything on the driver node!
- Instead, we can `rdd.take(n).foreach(println)` to print the first n elements of the collection, in order to be sure we will not run out of memory.

Removing data

- Spark performs a sort of garbage collection and automatically deletes old data partitions
- We can manually remove an RDD using the *unpersist()* 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([\n    StructField("field1", IntegerType(), True),\n    StructField("field2", FloatType(), True), \n    StructField("field3", TimestampType(), True), \n    StructField("field4", StringType(), True) \n])
```

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

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

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>