Data analytics with Apache Spark

Prace Advanced Training Center

Mario Macías Lloret

http://macias.info

@MaciasUPC

http://github.com/mariomac

February 2016

Overview of the course

1. Introduction
2. Apache Spark’s basic concepts
3. Spark SQL
4. Mllib

Objectives of this course

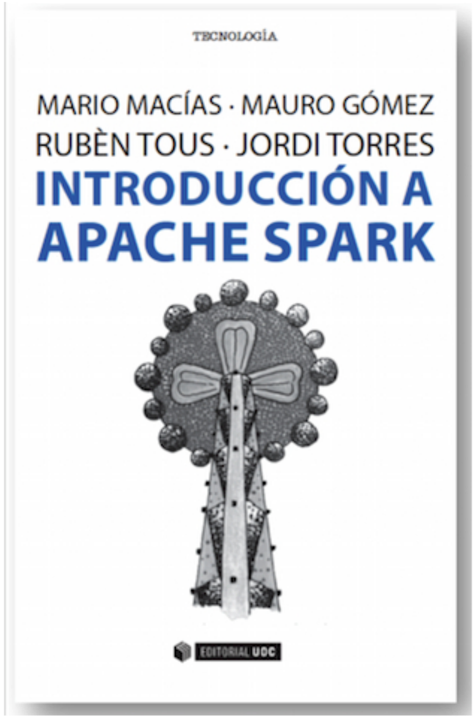
1. To get introduced in the Apache Spark architecture and software ecosystem
2. To learn the basics of some of its core components and libraries
3. Generally speaking, to expand the view about what kind of problems can be solved by means of Big Data frameworks and programming models

Structure of the course

- 3 basic topics are going to be presented
  - Basic architecture and core components
  - Spark SQL
  - Mllib
  - Because of time constraints, other topics won't be covered
    - Stream processing
    - Graph processing
- For each topic, a short introduction will be given, followed by hands-on exercises
  - Learn by doing

Other resources

- Official Spark documentation
  - http://spark.apache.org/docs/latest/
- Books
  - Learning Spark
    - http://spark.apache.org/docs/latest/
  - Introducción a Apache Spark
    - http://www.sparkbarcelona.es/

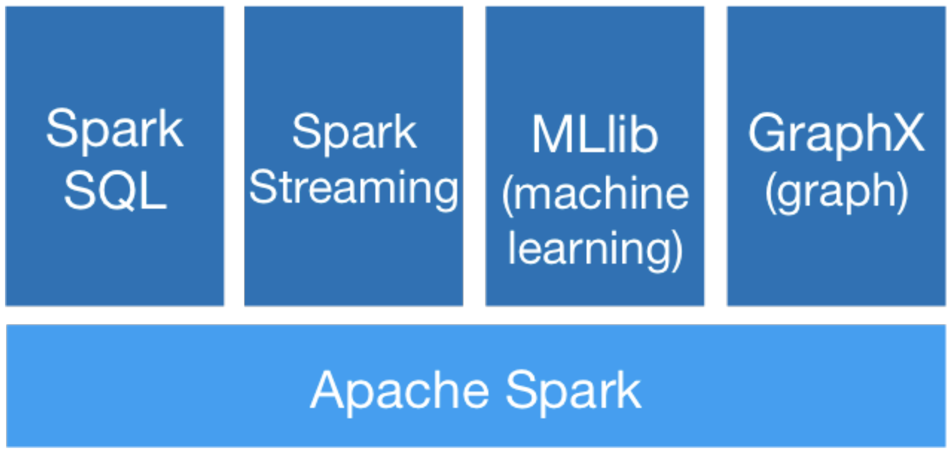


Overview of the course

1. Introduction
2. **Apache Spark's basic concepts**
3. Spark SQL
4. Mllib

What is Apache Spark

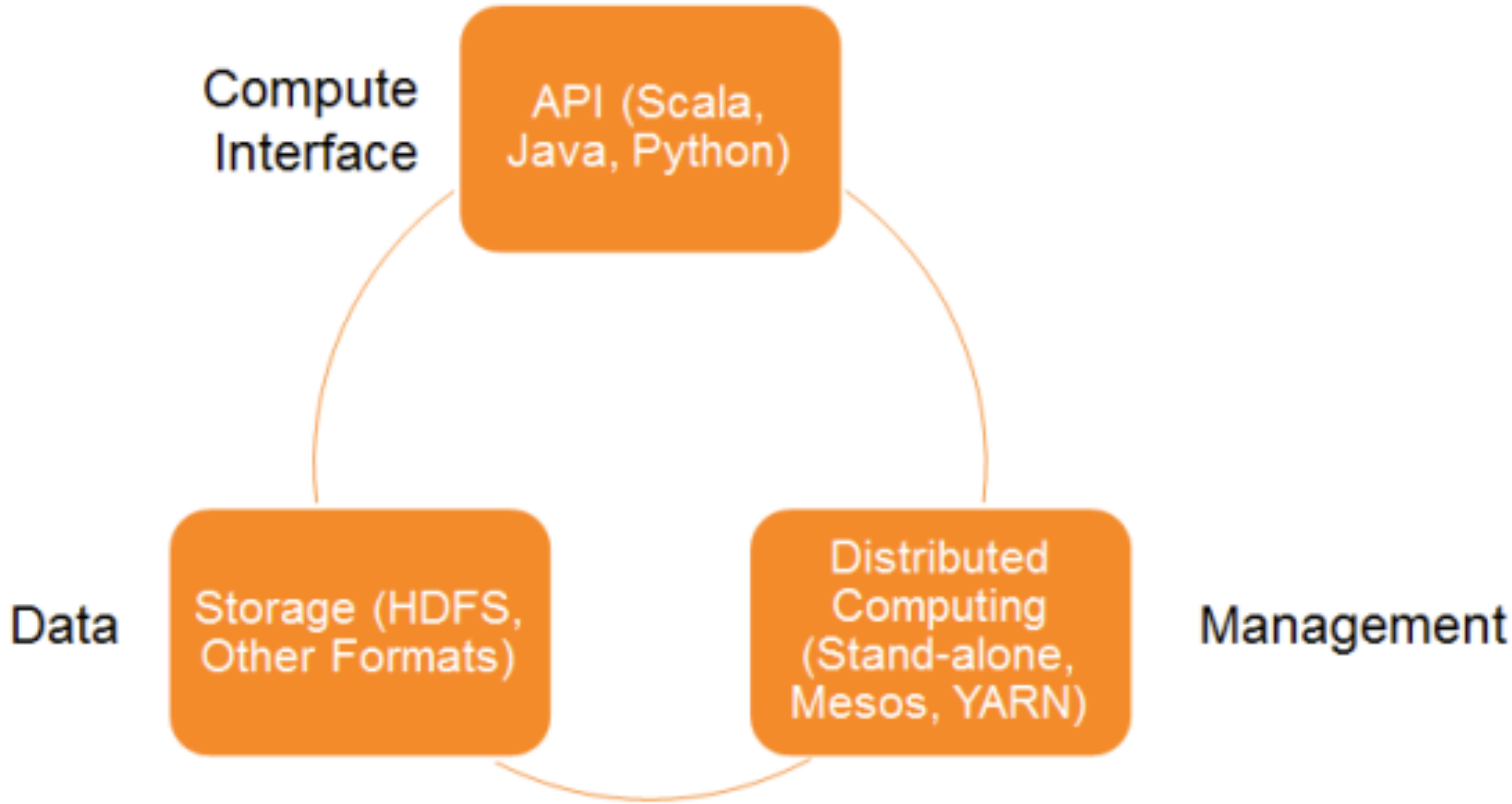
- Cluster computing framework
- Set of programming models and libraries
- Suited for data-intensive applications and Machine Learning problems
- 5 main components
  - Spark Core and RDDs
  - Spark SQL
  - Spark Streaming
  - Mllib
  - GraphX



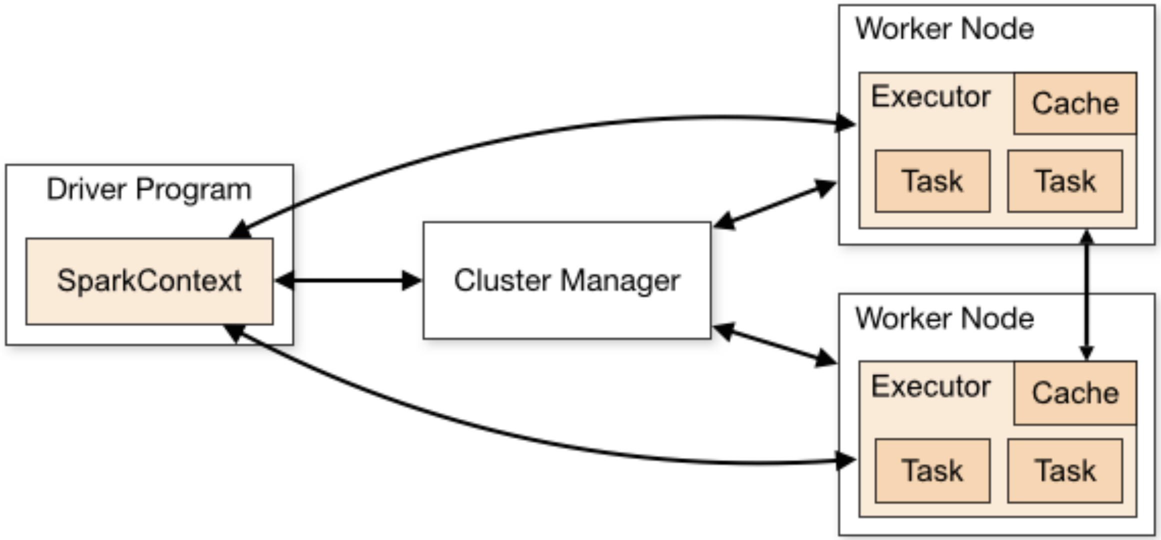
Spark vs Hadoop

- Pros
  - In-memory processing allows speed-up up to 100x for some problems
  - Supports multiple storage backends
    - Cassandra, HDFS, SQL databases…
  - Multiple language binding
    - Scala (main), Python, Java, R, Clojure
  - Well documented and easy to learn (personal opinion)
- Cons
  - Not as mature as Hadoop ecosystem

Spark core architecture



Cluster mode overview



- Cluster Managers
  - Localhost (to do the exercises in this course)
  - Standalone (included in Spark)
  - Apache Mesos
  - Hadoop YARN
  - Amazon EC2, through scripts

Resilient Distributed Datasets (RDDs)

- Are the core concept of Spark
- Keep data partitioned across the cluster nodes
- Are fault-tolerant
- Support two groups of actions
  - Transformations
  - Operations
- Are lazily evaluated: transformations are started only when operations are requested

First Example (Python)

- Console interactive mode:

```
$ IPYTHON=1 pyspark
In [1]: (sc.parallelize([1,2,3,4,5,6,7])
...: .filter(lambda v : v%2 == 1)
...: .map(lambda v : v*2)
...: .sum())
Out[1]: 32
```

1,3,5,7

2,6,10,14

First Example (Scala)

- Console interactive mode:

```
$ spark-shell
scala> (sc.parallelize(Array(1,2,3,4,5,6,7))
     | .filter( v => v%2 == 1)
     | .map(v => v*2)
     | .sum())
res0: Double = 32.0
```

Creating RDDs

- RDDs can be obtained from SQL/NoSQL databases, Scala/Python/Java/R/Clojure data types, disk files…
- In the exercises of this course, we will only use:
  - Data types

    ```
    rdd = sc.parallelize([1,2,3,4,5,6,7])
    ```
  - Disk files

    ```
    rdd = sc.textFile("derby.log")
    ```

Common operations on RDDs

- reduce(function)
  - Aggregates all the elements from an RDD according to the function
- collect()
  - Returns all the elements from an RDD as a list/array
- count()
  - Returns the number of elements in the RDD
- first()
  - Return the first element from an RDD
- histogram(classes)
  - Returns an histogram of for the ‘classes’ list
- saveAsTextFile(fileName)
- take(n)
  - Returns a list with the ‘n’ first elements
- Statistical functions: mean(), variance(), stdev(), sum(), max(), min()...

RDD operations examples (Scala interactive shell)

```
scala> val orig =
sc.parallelize(Array(34,1,345,12,1,45,7))

scala> Math.sqrt(orig.reduce((a,b) => a*a+b*b))
res6: Double = 45988.31822321838

scala> orig.count()
res7: Long = 7

scala> orig.first()
res8: Int = 34

scala> orig.histogram(Array(0.0, 10.0, 100.0, 1000.0))
res11: Array[Long] = Array(3, 3, 1)

scala> orig.mean()
res12: Double = 63.57142857142857
```

Common transformations on RDDs

- filter(function)
  - Returns a new RDD with the elements from the original that make the parameter function return ‘true’
- map(function)
  - returns a new RDD as the result of individually applying the function over the elements on the original RDD
- distinct()
  - removes duplicates from original RDDs
- sortBy(function)
  - orders an RDD according to the chriteria specified in the function
- union(otherRDD)
  - Returns an RDD as a result of the union on the target RDD and the parameter
- intersection(otherRDD)
  - Analogue to union, for intertsections

RDD transformation examples (Scala interactive shell)

```
scala> val orig =
sc.parallelize(Array(34,1,345,12,1,45,7))

scala> orig.map(v => "Val #"+v).collect()
res3: Array[String] = Array(Val #34, Val #1, Val
#345, Val #12, Val #1, Val #45, Val #7)

scala> orig.distinct().collect()
res4: Array[Int] = Array(12, 345, 45, 1, 34, 7)

scala> orig.sortBy(x => -x).collect()
res5: Array[Int] = Array(345, 45, 34, 12, 7, 1, 1)
```

Key-value pair RDD

- A key-value pair RDD is a special type of RDD formed by tuples, where the first element of the tuple is a key and the second element is an iterable element
- Common transformations
  - groupBy(func)
    - From an ordinary RDD, returns a new KVP RDD grouping the result of applying the function to each of its members
  - keys
    - Returns a list of keys
  - values
    - Returns a list of values
  - groupByKey()
  - mapValues()
  - sortByKey()
- Common operations
  - reduceByKey()
  - countByKey() / countByValue()
  - collectAsMap()

KVP examples

```
scala> orig.groupBy(v => v%10).sortByKey().collect()
res15: Array[(Int, Iterable[Int])] = Array((1,CompactBuffer(1, 1)),
(2,CompactBuffer(12)), (4,CompactBuffer(34)), (5,CompactBuffer(345,
45)),
(7,CompactBuffer(7)))

scala> val kvp = sc.parallelize(Array(("John", "Smith"), ("Jamie",
"Lee curtis"), ("John", "McEnroe")))

scala> kvp.keys.collect()
res30: Array[String] = Array(John, Jamie, John)

scala> kvp.values.collect()
res31: Array[String] = Array(Smith, Lee curtis, McEnroe)

scala> kvp.groupByKey().collect()
res32: Array[(String, Iterable[String])] =
Array((Jamie,CompactBuffer(Lee curtis)), (John,CompactBuffer(Smith,
McEnroe)))

scala> kvp.countByKey()
res34: scala.collection.Map[String,Long] = Map(Jamie -> 1, John -> 2)

scala> kvp.reduceByKey((value1, value2) => value1 + " and " +
value2).collect()
res35: Array[(String, String)] = Array((Jamie,Lee curtis),
(John,Smith and McEnroe))
```

Persisting RDDs

- The next (python) script may be inefficient

```
rdd1 = sc.parallelize([12,3,45,76,89,79])
rdd2 = sc.parallelize([345,3,23,12,54])
all = rdd1.union(rdd2).distinct()
print 'The collected elements are:'
print all.reduce(lambda a,b: str(a) +", " + str(b))
print "Max: %d " % all.max()
print "Min: %d " % all.min()
print "Average: %d " % all.mean()
print "Std Dev: %d " % all.stdev()
```

Persisting RDDs

- Persistence allows caching intermediate transformations to avoid recalculating them

```
rdd1 = sc.parallelize([12,3,45,76,89,79])
rdd2 = sc.parallelize([345,3,23,12,54])
all = rdd1.union(rdd2).distinct().persist()
print 'The collected elements are:'
print all.reduce(lambda a,b: str(a) +", " + str(b))
print "Max: %d " % all.max()
print "Min: %d " % all.min()
print "Average: %d " % all.mean()
print "Std Dev: %d " % all.stdev()
```

Hands-on: prominence calculator

https://github.com/mariomac/patc-spark/tree/master/exercises/1-intro

Overview of the course

1. Introduction
2. Apache Spark’s basic concepts
3. **Spark SQL**
4. Mllib

Data Frame

- A Data Frame is a distributed collection of data, organised as columns with an associated name.
  - The concept is similar to SQL tables
- The entry point to Spark SQL is the SQLContext class:

Python:

```
from pyspark.sql import SQLContext
sqlCtx = SQLContext(sc)
```

Scala:

```
import org.apache.spark.sql.SQLContext
val sqlContext = new SQLContext(sc)
```

Creating a Data Frame

- Data Frames can be loaded from different sources: JSON, JDBC/ODBC and Apache Hive.
  - Check official documentation
- In addition, Spark allows creating data frames from RDDs

```
scala> val rdd = sc.parallelize(Array(("Maria",
35),("Jose", 42),("Antonia", 25)))

scala> val people = sqlContext.createDataFrame(rdd)

scala> people.show()
+-------+---+
|     _1| _2|
+-------+---+
|  Maria| 35|
|   Jose| 42|
|Antonia| 25|
+-------+---+
```

Providing extra information to Data Frames

```
import org.apache.spark.sql.Row
import org.apache.spark.sql.types._

val rdd = sc.parallelize(Array(("Maria", 35),("Jose",
42),("Antonia", 25)))
val rowRdd = rdd.map(v => Row(v._1,v._2))
val schema = StructType(List(
     StructField("name",StringType),
     StructField("age",IntegerType)))
val people = sqlContext.createDataFrame(rowRdd,schema)
people.show()
+-------+---+
|   name|age|
+-------+---+
|  Maria| 35|
|   Jose| 42|
|Antonia| 25|
+-------+---+
```

Common operations and transformations for DF's

- **agg(*expressions)** returns a new DF with a single row, containing the results of the expressions passed by parameter

```
people.agg(avg(people.col("age")),max(people.col("age")),
min(people.col("age"))).show()

|avg(age)|max(age)|min(age)|
+--------+--------+--------+
|    34.0|      42|      25|
```

- **corr(col1, col2), cov(col1, col2)** returns the correlation/covariance between two columns
- **drop(col)** returns a new DF with the specified column dropped from the original
- **withColumn(name, expr)** returns a new column, given a name and the expression that provides its value:

```
people.withColumn("female",
people.col("name").endsWith("a")).show()

|   name|age|female|
+-------+---+------+
|  Maria| 35|  true|
|   Jose| 42| false|
|Antonia| 25|  true|
```

- Other methods common to RDDs: count, distinct, filter, ...

SQL-like operations on DFs

- select(*cols) returns a new DF with only the specified columns:

```
people.select("name").show()
+-------+
|   name|
+-------+
|  Maria|
|   Jose|
|Antonia|
+-------+
```

- filter(condition), where(condition) filters the rows given a condition

```
people.where(people.col("age") < 30).show()
+-------+---+
|   name|age|
+-------+---+
|Antonia| 25|
+-------+---+
```

- groupBy(*columns) similar to SQL GROUP BY, providing aggregation functions:

```
people.groupBy(people.col("age") % 10).avg().show()
+----------+--------+
|(age % 10)|avg(age)|
+----------+--------+
|         2|    42.0|
|         5|    30.0|
+----------+--------+
```

SQL text queries

```
val schema = StructType(List(
     StructField("name",StringType),
     StructField("age",IntegerType),
     StructField("passport",StringType)))

val students = sqlContext.createDataFrame(
    sc.parallelize(
       Array(Row("Jaime",32,"12345-f"),
          Row("Maria",19,"22222-g"),
          Row("Alex",23,"65432-z"))), schema)

students.registerTempTable("students")
```

SQL text queries (II)

```
val schema2 = StructType(List(
    StructField("passport",StringType),
    StructField("year",IntegerType)))

val enrollments = sqlContext.createDataFrame(
     sc.parallelize(
       Array(Row("12345-f", 1990),
             Row("22222-g", 2014),
             Row("65432-z", 2009))), schema2)

enrollments.registerTempTable("enrollments")
```

SQL text queries (III)

```
sqlContext.sql("""
     SELECT students.name, enrollments.year AS
enrollment_year
     FROM students, enrollments
     WHERE students.passport = enrollments.passport
     ORDER BY students.name ASC
     """).show()
```

```
+-----+---------------+
| name|enrollment_year|
+-----+---------------+
| Alex|           2009|
|Jaime|           1990|
|Maria|           2014|
+-----+---------------+
```

Hands-on: Google Cluster data analiser

https://github.com/mariomac/patc-spark/tree/master/exercises/2-sql

Overview of the course

1. Introduction
2. Apache Spark's basic concepts
3. Spark SQL
4. **<u>Mllib</u>**

Machine Learning Library (MLlib)

- spark.mllib: toolset of learning algorithms and utilities
  - Data types
  - Basic statistic tools
  - Classification and regressions
  - Collaborative filtering
  - Clustering
  - Dimensionality reduction
  - Feature extraction and transformation
  - Frequent pattern mining
  - Evaluation metrics
- spark.ml: high-level APIs for ML pipelines

Spark ML pipelines

- DataFrame
  - from SQL as ML dataset
- Transformer
  - algorithm that transforms a DataFrame into another DataFrame
- Estimator
  - algorithm which can be fit into a DataFrame to produce a Transformer
  - e.g. a learning algorithms that learns from a DataFrame to produce a model
- Pipeline
  - chains multiple Transformers and estimators to specify a ML workflow
- Parameter
  - Common API shared by transformers and estimators to specify parameters