Data analytics with Apache Spark

Mario Macías Lloret
http://macias.info
@MaciasUPC
http://github.com/mariomac

May 2016

Overview of the course

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

Objectives of this course

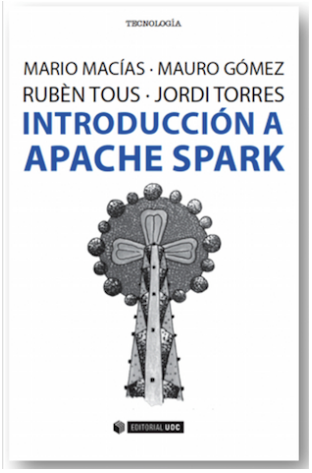
1. To get introduced in the Apache Spark architecture and software ecosystem
2. To learn the basics of some of its core components and libraries
3. Generally speaking, to expand the view about what kind of problems can be solved by means of Big Data frameworks and programming models

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/

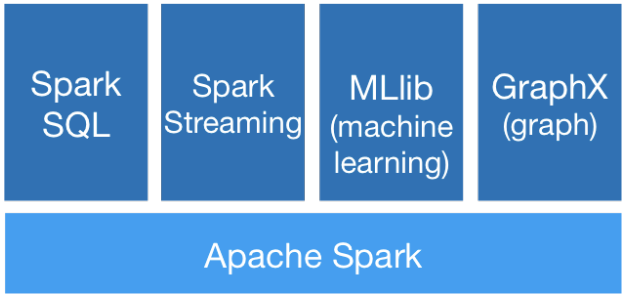


Overview of the course

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

What is Apache Spark

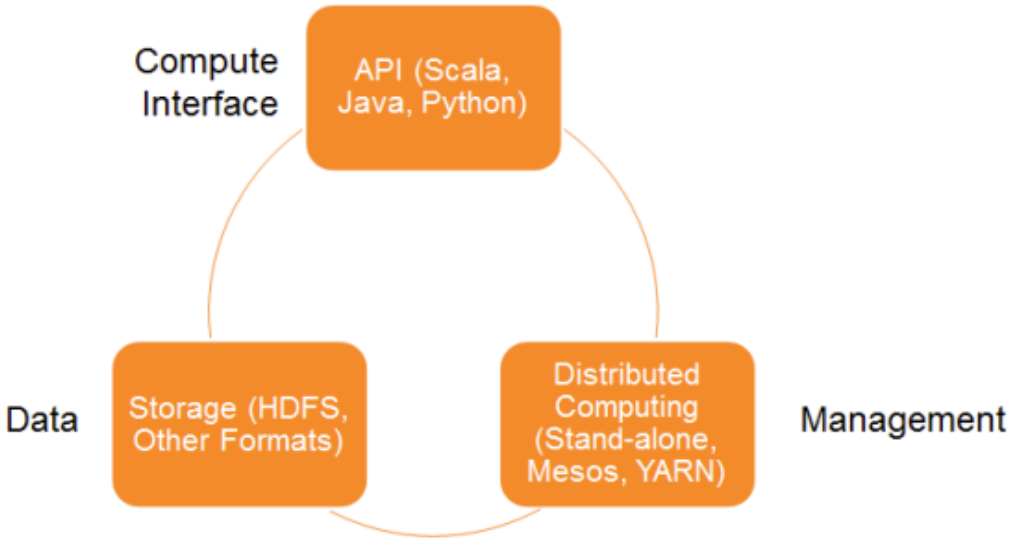
- Cluster computing framework
- Set of programming models and libraries
- Suited for data-intensive applications and Machine Learning problems
- 5 main components
  - Spark Core and RDDs
  - Spark SQL
  - Spark Streaming
  - Mllib
  - GraphX



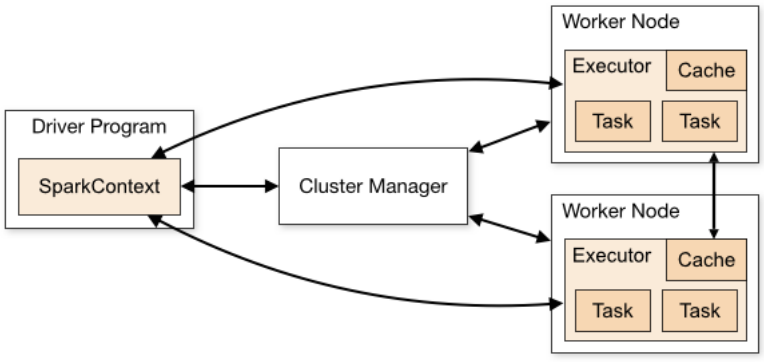
Spark vs Hadoop

- 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

- 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

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

```
In [3]: orig = sc.parallelize([34,1,345,12,1,45,7])

In [5]: import math
In [8]: math.sqrt(orig.reduce(lambda a,b: a*a+b*b))
Out[8]: 2.007092677359242e+20

In [9]: orig.count()
Out[9]: 7

In [10]: orig.first()
Out[10]: 34

In [11]: orig.histogram([0,10,100,1000])
Out[11]: ([0, 10, 100, 1000], [3, 3, 1])

In [12]: orig.mean()
Out[12]: 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

```
In [12]: orig = sc.parallelize(
                                [34,1,345,12,1,45,7])

In [13]: orig.map(lambda v: v*2).collect()
Out[13]: [68, 2, 690, 24, 2, 90, 14]

In [14]: (orig.map(lambda v: "Val #"+str(v))
              .collect())
Out[14]: ['Val #34', 'Val #1', 'Val #345', 'Val
#12', 'Val #1', 'Val #45', 'Val #7']

In [15]: orig.distinct().collect()
Out[15]: [12, 1, 45, 345, 34, 7]

In [17]: orig.sortBy(lambda v: -v).collect()
Out[17]: [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

```
In [1]: orig = sc.parallelize([34,1,345,12,1,45,7])
In [2]: orig.groupBy(lambda  v: v%10).sortByKey().collect()
Out[3]:
[(1, <pyspark.resultiterable.ResultIterable  at 0x101d62f10>),
 (2, <pyspark.resultiterable.ResultIterable  at 0x101d52990>),
 (4, <pyspark.resultiterable.ResultIterable  at 0x101d52cd0>),
 (5, <pyspark.resultiterable.ResultIterable  at 0x101d52a10>),
 (7, <pyspark.resultiterable.ResultIterable  at 0x101d52c90>)]

In [5]: kvp = sc.parallelize([("John","Smith"),("Jamie",
"Lee Curtis"), ("John", "McEnroe")])

In [7]: kvp.keys.collect()
Out[7]: ['John', 'Jamie', 'John']

In [8]: kvp.values().collect()
Out[8]: ['Smith', 'Lee Curtis', 'McEnroe']

In [9]: kvp.groupByKey().collect()
Out[9]:
[('Jamie', <pyspark.resultiterable.ResultIterable  at 0x101d62c10>),
 ('John', <pyspark.resultiterable.ResultIterable  at 0x101b4cc10>)]

In [10]: kvp.reduceByKey(lambda  val1,val2: val1 + " and " +
val2).collect()
Out[10]: [('Jamie', 'Lee Curtis'), ('John', 'Smith and McEnroe')]
```

Persisting RDDs

- The next 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