



**Barcelona  
Supercomputing  
Center**  
*Centro Nacional de Supercomputación*



EXCELENCIA  
SEVERO  
OCHOA

# Data Analytics using Apache Spark

RDA Europe spring school on weather, climate and air quality 2017

Josep Lluís Berral García  
Data Centric Computing - BSC

May, 2017



# Introduction

## « What is Apache Spark

- Cluster Computing Framework
- Programming clusters with data parallelism and fault tolerance
- Programmable in Java, Scala, Python and R

## « Motivation for using Spark

- Spark schedules data parallelism implicitly
  - User defines the set of operations to be performed, and Spark performs an orchestrated execution
- It works with Hadoop and HDFS
  - Spark can bring execution to where data is distributed
  - It can work over Hadoop and use Distributed File Systems
- It provides libraries for distributed algorithms on Machine Learning, Graphs, Streaming, DataBase queries, etc.

1. Introduction to Spark
2. Treating data as relational with SparkSQL
3. Machine Learning using SparkML

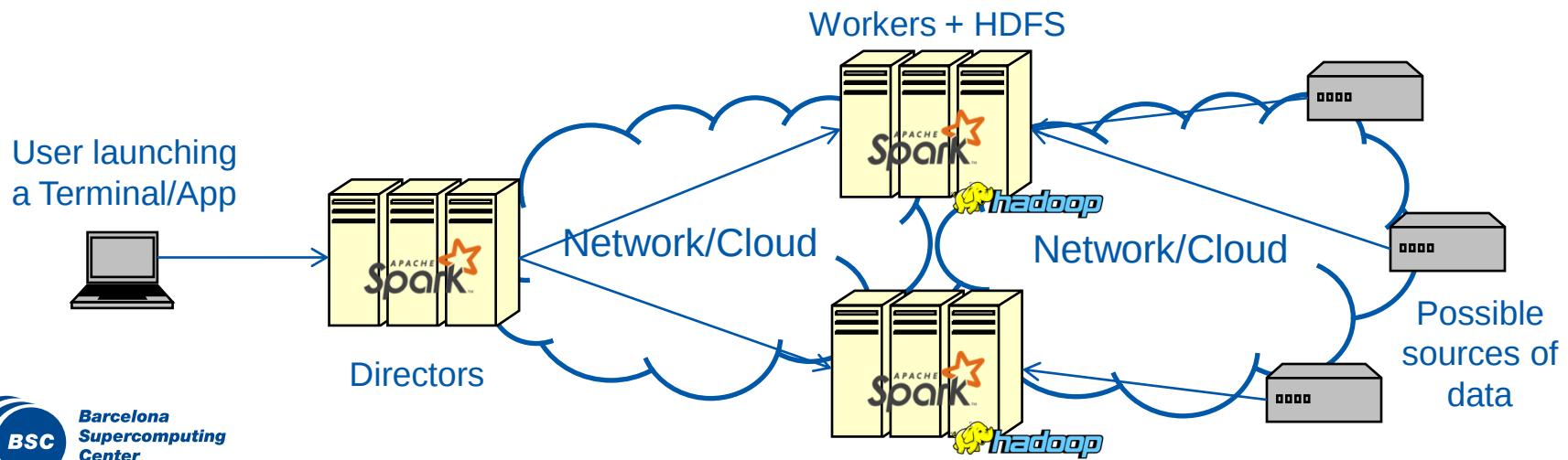
# Introduction to Apache Spark

## Cluster Computing Framework

- Implemented in Java
- Programmable in Java, Scala, Python and R
- Paradigm of Map-Reduce

## Deployment:

- Define your cluster (directors and workers)
- Link to your distributed File System
- Start a session / Create an app
- Let Spark to plan and execute the workflow and dataflow



# Step 0: Installing the Environment

## « Installation Notes:

- Download Java
- Download Spark (also Hadoop and Yarn if building a cluster with HDFS)

## « In the provided Vagrant VM, Spark is already downloaded

- “`/home/ubuntu/spark`” (or corresponding version)

## « What we will do now:

- We will use spark in “**standalone mode**”, using our computing as single node
- We will use the **local filesystem**, although we could set-up a distributed filesystem like HDFS

# Step 0: Installing the Environment

## Let's run SPARK!

### – Options:

- `./bin/spark-shell` → We open a Scala session to Spark
- `./bin/sparkR` → We open a R-cran session to Spark
- `./bin/spark-submit` → We send our Spark application to Spark

### – For now, we're using Scala

```
Spark context Web UI available at http://10.0.0.100:4040
```

```
Spark context available as 'sc' (master = local[*], app id = local-1485515595386).
```

```
Spark session available as 'spark'.
```

```
Welcome to
```

```
    _/\_/_\_\_/\_\_/\_\_/\_\_/\_\_/\_\_
   / \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
  / \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
 / \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
 / \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
 / \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
 / \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
 / \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
 / \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
```

version 2.0.0

```
Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_91)
```

```
Type in expressions to have them evaluated.
```

```
Type :help for more information.
```

```
scala>
```

# Spark and Scala

## Scala is oriented towards functional programming

- We use **val** for static values and **var** for variable values

- val a = "hola"
- var b = "adeu"
- b = "adeu siau"

- ~~Maps~~

- Operations on RDDs will be distributed by SPARK over the available nodes
- Also RDD distributed operations will happen over the HDFS partitioning of data

- Note: Exit Scala

- :q (two dots + q)

## Learning by example

### donation

- Find the dataset in "/home/ubuntu/datasets/donation" for
- Originally obtained from <http://bit.ly/1AoYwaq>

# Spark and Scala

## « Accessing the data:

- This dataset is already partitioned, and it can be uploaded to HDFS, or we can access all in local
- For working with it in local, we should reunite the blocks
  - Manually unzip the files
    - unzip 'block\_\*.zip'
  - Fuse all blocks in one
    - cat block\_\*.csv > block\_all.csv

## « Opening Session:

- Back in the spark folder
- ./bin/spark-shell --master local [\*]

## « Load blocks from the file:

- ```
val rawblocks =  
sc.textFile("file:///home/ubuntu/datasets/donation/bloc  
k_all.csv")
```

# Spark and Scala

## Let's examine the data:

- val head = rawblocks.take(10)  
head: Array[String] = Array("id\_1", "id\_2", ..., 1, TRUE)
- head.length  
res7: Int = 10
- head.foreach(println)  
"id\_1", "id\_2", "cmp\_fname\_c1", "cmp\_fname\_c2", ..., "is\_match"  
31641, 62703, 1, ?, 1, ?, 1, 1, 1, 1, 1, TRUE  
27816, 46246, 1, ?, 1, ?, 1, 1, 1, 1, 1, TRUE  
980, 2651, 1, ?, 1, ?, 1, 1, 1, 1, TRUE  
6514, 8780, 1, ?, 1, ?, 1, 1, 1, 1, 1, TRUE  
5532, 14374, 1, ?, 1, ?, 1, 1, 1, 1, 1, TRUE  
25763, 61342, 1, ?, 1, ?, 1, 1, 1, 1, 1, TRUE  
59655, 59657, 1, ?, 1, ?, 1, 1, 1, 1, 1, TRUE  
23800, 32179, 1, ?, 1, ?, 1, 1, 1, 1, 1, TRUE  
33568, 38196, 1, ?, 1, ?, 1, 1, 1, 1, 1, TRUE

# Spark and Scala

## « Define a function:

- ```
def isHeader(line: String) = line.contains("id_1")  
isHeader: (line: String) Boolean
```

## « Use our function:

- ```
head.filter(isHeader).foreach(println)  
"id_1", "id_2", "cmp_fname_c1", ..., "is_match"
```

## « Notice how we treat data:

1. ~~Read~~

2. We apply a filter (select rows based on a boolean vector/condition)
3. We introduce our function, that given a String returns a boolean
4. The result is the selected rows satisfying the condition
5. ~~Type~~ `foreach println` for each element of such rows

# Spark and Scala

## « Keep using our function over data:

- head.filter(isHeader).length  
res10: Int = 1
- head.filterNot(isHeader).length  
res11: Int = 9

## « Here we are simplifying a lambda function

- head.filterNot(isHeader).length  
**is equivalent to**
- head.filterNot(x => isHeader(x)).length  
**also is equivalent to**
- head.filterNot(isHeader(\_)).length  
**also is equivalent to**
- head.filter(!isHeader(\_)).length  
**etcetera...**

# Spark and Scala

## Now, lets process the data a little bit:

- val noheader = rawblocks.filter(!\_isHeader(\_))
- noheader.first
  - res15: String = 31641,62703,1,?,1,?,1,1,1,1,1,1,TRUE
- val line = head(5)
- val pieces = line.split(',')'
- val id1 = pieces(0).toInt
- val id2 = pieces(1).toInt
- val matched = pieces(11).toBoolean
- deftoDouble(s: String) = { if ("?".equals(s)) Double.NaN else s.toDouble }

Beware counting starts at 0! Don't push here!!  
fetched

- val rawscores = pieces.slice(2,11)
- val scores = rawscores.map(x => toDouble(x))

# Spark and Scala

## « We can put it all together in a function

- def parse(line: String) = {  
    val pieces = line.split(',', ')')  
    val id1 = pieces(0).toInt  
    val id2 = pieces(1).toInt  
    val scores = pieces.slice(2, 11).maptoDouble)  
    val matched = pieces(11).toBoolean  
    (id1, id2, scores, matched)  
}  
parse: (line: String) (Int, Int, Array[Double], Boolean)
- val tup = parse(line)  
tup: (Int, Int, Array[Double], Boolean) =  
(5532, 14374, Array(1.0, NaN, 1.0, NaN, 1.0, 1.0, 1.0, 1.0), true)

# Spark and Scala

## Accessing our processed values

like this, it starts at 1

- `tup._1`  
`res17: Int = 5532`

like this, it starts at 0, and see the return type

- `tup.productElement(0)`  
`res18: Any = 5532`

we can see the number of elements direct into the value

- `tup.productArity`  
`res19: Int = 4`

also we can access to the elements of the array inside the value

- `tup._3.foreach(println)`  
`1.0`  
`NaN`  
`...`

# Spark and Scala

## « We can create objects

- case class MatchData(id1: Int, id2: Int, scores: Array[Double], matched: Boolean)

and use them as types

- def parse(line: String) = {  
    val pieces = line.split(',', ',')  
    val id1 = pieces(0).toInt  
    val id2 = pieces(1).toInt  
    val scores = pieces.slice(2, 11).map(toDouble)  
    val matched = pieces(11).toBoolean  
    MatchData(id1, id2, scores, matched)  
}

Then use as values

- val md = parse(line)
- md.matched
- md.scores

# Spark and Scala

## « Map operations to elements

- ```
val mds = head.filter(s => !isHeader(s)).map(l =>
parse(l))
```

```
mds: Array[MatchData] = Array(MatchData(31641, 62703, ...))
```
- ```
mds.foreach(println)
```

```
MatchData(31641, 62703, [D@3a5b429, true])
```

```
...
```

## « Grouping data, then operate by groups

- ```
val grouped = mds.groupBy(md => md.matched)
```

```
grouped: scala....Map[Boolean,Array[MatchData]] = Map(true ->
Array(MatchData(31641, 62703, [D@63cdd551, true], ...)), false->Array(...))
```
- ```
grouped.mapValues(x => x.size).foreach(println)
```

```
(true, 9)
```

# Spark and Scala

## Let's keep data parsed, and enabled in cache

- `val parsed = noheader.map(line => parse(line))`
- `parsed.cache()`

## Let's do an operation that requires to be distributed

- `val mapMatch = parsed.map(entry => entry.matched)`
- `val matchCounts = mapMatch.countByValue()`  
`res17: scala.collection.Map[Boolean, Long] = Map(false -> 5728201,  
true -> 20931)`
- Here, due to "count" Spark requires to check all the data and give results, so it starts distributing computation
- You can see in console how the different "executors" process data, and how the stages on the scheduled plan are passed
- This is different from before because when playing with "head" it was in the Director machine/process, while now we are querying to all distributed (if it is) data.

# Map/Reducing Operations

## « The classical example: WordCount

We open a file to count

- val textFile = sc.textFile("/README.md")

we check that we could read the file

- textFile.first  
res21: String = # Apache Spark

now we map/reduce

- val textFlatMap = textFile.flatMap(line =>  
line.split(""))
- val words = textFlatMap.map(word => (word, 1))
- val counts = words.reduceByKey(\_ + \_)

then retrieve some results (triggering the scheduler)

- counts.take(5).foreach(println)  
(package,1)  
(this,1)  
(Version"] (<http://spark.apache.org/d...-version>) ,1)  
(Because,1)  
(Python,2)

# Map/Reducing Operations

## « We can order the results:

Counts is still a distributed operation, and we can keep operating

- val ranking = counts.sortBy(\_.value, false)

But now, if we want to see the ranking, we need to collect all data, triggering the scheduler

- val local\_result = ranking.collect()

Then get the top 5

- ranking.take(5)  
res31: Array[(String, Int)] = Array(("the", 71), ("Spark", 24), ...)
- ranking.take(5).foreach(println)  
(the, 71)  
(Spark, 24)  
(to, 17)  
(Spark, 16)  
(for, 12)

# Map/Reducing Operations

## « We can process the results

Also we can include data processing to remove the word

- val cleancount = counts.filter(x => { !"" .equals(x.\_1) })
- val cleanrank = cleancount.sortBy(\_. \_2, false)
- val local\_cleanrank = cleanrank.collect()
- local\_cleanrank.take(5).foreach(println)  
(the, 24)  
(to, 17)  
(Spark, 16)  
(for, 12)  
(and, 9)

Or filter it before counting, by modifying the previous instruction

- val words2 = textFlatMap.filter(word => { !"" .equals(word) } )
- val filteredwords = words2.map(word => (word, 1))
- val counts2 = filteredwords.reduceByKey(\_ + \_)
- val ranking2 = counts2.sortBy(\_. \_2, false)
- val local\_result2 = ranking2.collect()
- local\_result2.take(5).foreach(println)

# SparkSQL

## « With Spark we can

- Access our data as it was a Relational DB
- Using a syntax really close to SQL
- ...great news for people used to DB systems

## « Now we are using DataFrames!

- ...Distributed DataFrames
- This means that we can also open a SparkR session, and do most of the R usual stuff into those data.frames

# Loading some big data

## « Let's get some Big Data

- Locate the file `ss13husa.csv` from `csv_hus` folder
  - Originally obtained from <http://bit.ly/2jZqeZY>

## « The file we downloaded contains some CSV files

- We are going to open one, directly as a Data.Frame

## « Let's open it:

- ```
val df = spark.read.format("csv")
    .option("inferSchema", true)
    .option("header", "true")
    .load("/home/ubuntu/datasets/csv_hus/ss13husa.csv")
```

## « First look (as RDD):

- ```
df.take(5)
```

```
res0: Array[org.apache.spark.sql.Row] =
  Array([H,84,6,2600,3,1,1000000,1007549,0,1,3,null,null,null,null,null,n
ull,null,null,null,null,null,null,null,null,2,null,null,null,null,null,n
ull,null,null,null,null,null,null,null,null,null,null,null,null,null,n
ull..]
```

# Relational DataFrames

## « Let's print the Data.Frame schema:

- df.printSchema()

```
root
| -- RT: string (nullable = true)
| -- SERIALNO: integer (nullable = true)
| -- DIVISION: integer (nullable = true)
| -- PUMA: integer (nullable = true)
| -- REGION: integer (nullable = true)
| -- ST: integer (nullable = true)
| -- ADJHSG: integer (nullable = true)
| -- ADJINC: integer (nullable = true)
| -- WGTP: integer (nullable = true)
| . . .
```

# Relational DataFrames

## « Examining the DataFrame

- `df.count()`  
`res4: Long = 756065`

## « Selecting a subset of rows

- `val df1 = df.limit(10)`
- `df1.show()`  
[HERE GOES A SQL-like TABLE!]
- Now we can operate with “df” as tables in a SQL server

# Relational side-by-side SQL DataFrames

## « You can work with SQL syntax

- ~~But you need to register the DataFrame as a temporary view~~
- `df.createGlobalTempView("husa")`

## « Selecting

1. `df.select($"SERIALNO", $"RT", $"DIVISION", $"REGION").show()`
2. `spark.sql("SELECT SERIALNO, RT, DIVISION, REGION FROM global_temp.husa").show()`

## « Filtering

1. `df.select($"SERIALNO", $"RT", $"DIVISION", $"REGION").filter($"PUMA" > 2600).show()`
2. `spark.sql("SELECT SERIALNO, RT, DIVISION, REGION FROM global_temp.husa WHERE PUMA < 2100").show()`

## « Grouping

1. `df.groupBy("DIVISION").count().show()`
2. `spark.sql("SELECT DIVISION, COUNT(*) FROM global_temp.husa GROUP BY DIVISION").show()`

# SparkML

## « SparkML a.k.a Mllib

- This is the Machine Learning library for Spark

## « Distributed ML Algorithms

- Basic Statistics
  - Summaries, Correlations, Hypothesis Tests...
- Classification and Prediction
  - ~~SVM~~
- Clustering
  - ~~k-Means~~ k-means
- Others
  - Dimension Reduction
  - Collaborative Filtering
  - Frequent Pattern Mining

# Distributed Algorithms with SparkML

## « Spark takes advantage of splitting data in subsets

- Subsets are distributedly processed for models
- Partial Models are aggregated into a general model
- Such methodology is not as fitted as centralized approaches...
- ... But at least can be processed
- ... Also, we could discuss how huge datasets could bring to statistically significant sampled subsets

## « ML process relies on a Map/Reduce strategy

# SparkML Types of Data

## « Vectors (Local)

- import org.apache.spark.mllib.linalg.{Vector, Vectors}

### DenseVectors (all values)

- val dv: Vector = Vectors.dense(1.0, 0.0, 3.0)

### SparseVectors (length, indexes, values)

- val sv: Vector = Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0))

## « Labeled Points

- import org.apache.spark.mllib.regression.LabeledPoint

### Example of Two points labeled (1,0)

- val pos = LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0))
- val neg = LabeledPoint(0.0, Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0)))

# SparkML Types of Data

## Matrices

- `import org.apache.spark.mllib.linalg.{Matrix, Matrices}`

### Dense Matrices

- `val dm: Matrix = Matrices.dense(3, 2, Array(1.0, 3.0, 5.0, 2.0, 4.0, 6.0))`

### Sparse Matrices

- `val sm: Matrix = Matrices.sparse(3, 2, Array(0, 1, 3), Array(0, 2, 1), Array(9, 6, 8))`

## Distributed Matrices

We prepare some Vectors as RDD for using as rows

- `val v0 = Vectors.dense(1.0, 0.0, 3.0)`
- `val v1 = Vectors.sparse(3, Array(1), Array(2.5))`
- `val v2 = Vectors.sparse(3, Seq((0, 1.5), (1, 1.8)))`

# SparkML Types of Data

## Row Matrices

- import  
org.apache.spark.mllib.linalg.distributed.RowMatrix

We convert the Vectors into an RDD

- val rows1 = sc.parallelize(Seq(v0, v1, v2))

Then we construct the Matrix

- val mat: RowMatrix = new RowMatrix(rows1)

Now we can operate with the Matrix

- val m = mat.numRows()
- val n = mat.numCols()

This matrix works by rows

- mat.rows.collect.foreach(println)

# Modeling and Prediction - Regression

## Least Squares

- import org.apache.spark.mllib.regression.LinearRegressionModel
- import org.apache.spark.mllib.regression.LinearRegressionWithSGD

### Load and parse some data

- val data = sc.textFile("data/mllib/ridge-data/lpsa.data")
- val lines = data.map(x => x.split(', '))
- val parsedData = lines.map(x => LabeledPoint(x(0).toDouble, Vectors.dense(x(1).split(' ').map(x => x.toDouble)))) .cache()

### Take a look at our parsed data

- parsedData.take(3).foreach(println)  
(-0.4307829, [-1.63735562648104, -2.00621178480549, ...])  
(-0.1625189, [-1.98898046126935, -0.722008756122123, ...])  
(-0.1625189, [-1.57881887548545, -2.1887840293994, ...])

# Modeling and Prediction - Regression

« ...

Now we build a model

- val numIterations = 100
- val stepSize = 0.00000001
- val model = LinearRegressionWithSGD.train(parsedData, numIterations, stepSize)

Then we evaluate our model: Training MSE

- val predictions = parsedData.map(x => (x.label, model.predict(x.features)))
- val MSE = predictions.map{case(v,p) => math.pow((v-p),2)}.mean()
- ```
println("training Mean Squared Error = " + MSE)
      training Mean Squared Error = 7.4510328101026015
```

Finally, we proceed to save our model, to be reloaded after

- model.save(sc, "LR\_SGD\_Model")
- val sameModel = LinearRegressionModel.load(sc, "LR\_SGD\_Model")

# Modeling and Prediction - Classification

## Support Vector Machines

- import org.apache.spark.mllib.classification.{SVMModel, SVMWithSGD}
- import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
- import org.apache.spark.mllib.util.MLUtils

### Load and parse some data

- val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample\_libsvm\_data.txt")

### We do some splitting for Training vs. Test data (60% vs. 40%)

- val splits = data.randomSplit(Array(0.6, 0.4), seed=11L)
- val training = splits(0).cache()
- val test = splits(1)

# Modeling and Prediction - Classification

« ...

## Now we build a model

- val numIterations = 100
- val model = SVMWithSGD.train(training, numIterations)
- model.clearThreshold()

## Then we evaluate our model: Test AUC

- val predictions = test.map(x => (model.predict(x.features), x.label))
- val metrics = new BinaryClassificationMetrics(predictions)
- val auROC = metrics.areaUnderROC()
- println("Area under ROC = " + auROC)  
Area under ROC = 1.0

## Then we save and load it again

- model.save(sc, "SVM\_SGD\_Model")
- val sameModel = SVMMODEL.load(sc, "SVM\_SGD\_Model")

# Clustering

## « Our beloved classical k-means

- import org.apache.spark.mllib.clustering.{KMeans, KMeansModel}

First of all, load and parse the data

- val data = sc.textFile("data/mllib/kmeans\_data.txt")
- val parsedData = data.map(s => Vectors.dense(s.split(' ').map(x => x.toDouble))).cache()

Then train a model

- val numClusters = 2
- val numIterations = 20
- val clusters = KMeans.train(parsedData, numClusters, numIterations)

# Clustering

« ...

Then we evaluate our model usin the Sum of Squared Errors

- val WSSSE = clusters.computeCost(parsedData)
- println("Within Set Sum of Squared Errors = " + WSSSE)

Within Set Sum of Squared Errors = 0.1199999999994547

Finally, we save the model

- clusters.save(sc, "KMeansModel")
- val sameModel = KMeansModel.load(sc, "KMeansModel")

# Other Spark Libraries – SparkStream

- « Spark Stream is the library for dealing with Streams of data
  - Spark has a context for running operations
  - SparkStream has a context that updates each time
- « RDDs in the streaming context change each time the stream is updated
  - In Scala, as we indicate *how things* are but not *what to do*, each streaming context will state that “*things are*” how we defined
  - ...this is, we will have no loops for each update, but an updated dynamic context always running

# Other Spark Libraries – GraphX

## GraphX - SparkGraphs

- The distributed Graph processing library of Spark
- GraphX extends the Spark RDD by introducing new Graph abstractions
- Details:
  - Used to parse graphs (more than graphs that need to be updated)
  - Two separate implementations: Pregel abstraction and MapReduce
  - Treat data as a graph database

# Other Spark Libraries – SparkR

## SparkR

- ~~Not implemented SparkR API taking in mind DataFrames/RoofSpark~~

Allows to be started from an R session

- ```
Sys.setenv(SPARK_HOME='/opt/spark-2.1.0');
```
- ```
.libPaths(c(file.path(Sys.getenv('SPARK_HOME'), 'R', 'lib'), .libPaths()));
```
- ```
library(SparkR);
```
- ```
sc <- sparkR.init();
```
- ```
sqlContext <- sparkRSQl.init(sc);
```

Data Frames and Apply functions are implemented over Spark

- ```
# Example 1
```
- ```
housing_sub <- read.parquet(sqlContext, "hvalp1000.parquet");
```
- ```
aggreg <- SparkR:::lapply(housing_sub, function(x) paste(as.character(x$ST), "AAA", sep="_"))
```
- ```
take(aggreg,10);
```
- ```
# Example 2
```
- ```
df <- read.df(sqlContext, "./iris.data");
```
- ```
training <- filter(df, df$Species != "setosa");
```
- ```
model <- glm(Species ~ Sepal_Length + Sepal_Width, data = training, family = "binomial");
```



**Barcelona  
Supercomputing  
Center**  
*Centro Nacional de Supercomputación*



# Thanks for your Attention

Josep Ll. Berral García  
Data Centric Computing - BSC