www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Data Analytics using Apache Spark

PATC 2018

Josep Lluís Berral García
Data Centric Computing - BSC

February, 2018

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

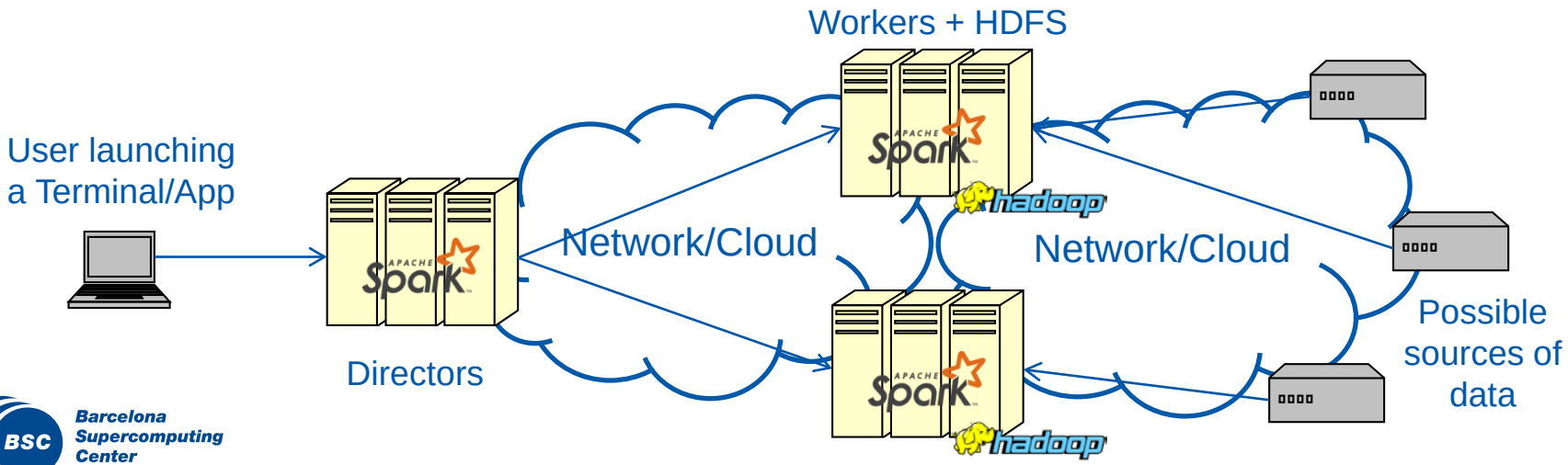
- Spark schedules data parallelism implicitly
  - User defines the set of operations to be performed
  - Spark performs an orchestred execution
- It works with Hadoop and HDFS
  - Bring execution to where data is distributed
  - Taking advantage Distributed File Systems
- It provides libraries for distributed algorithms
  - Machine Learning
  - Graphs
  - Streaming
  - DataBase queries

Index

Introduction to Apache Spark

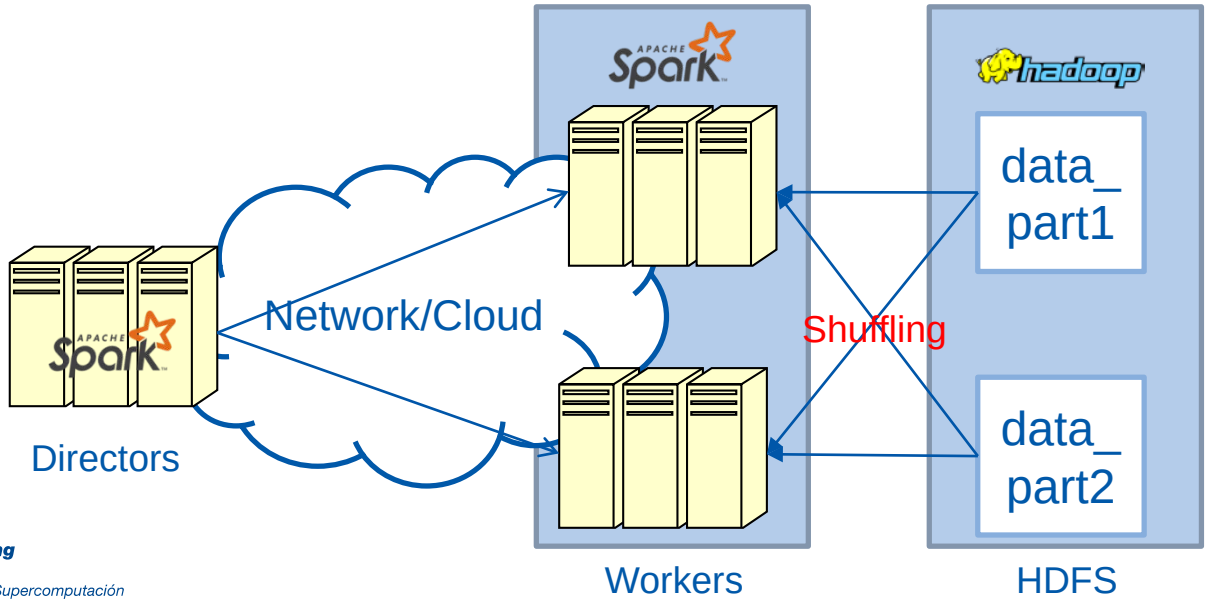
- Cluster Computing Framework
  - Implemented in Java
  - Programmable in Java, Scala, Python and R
  - Paradigm of Map-Reduce

- 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



Computing and Shuffling

- Spark attempts to compute data “where it sits”
- When using a DFS, Spark takes advantage of distribution
- If operations require to cross data from different places
  - Shuffling: Data needs to be crossed among workers
  - We must think of it when preparing operations
  - … also when distributing data on the DFS
  - … also when manually partitioning data for distributed processing



Step 0: Installing the Environment

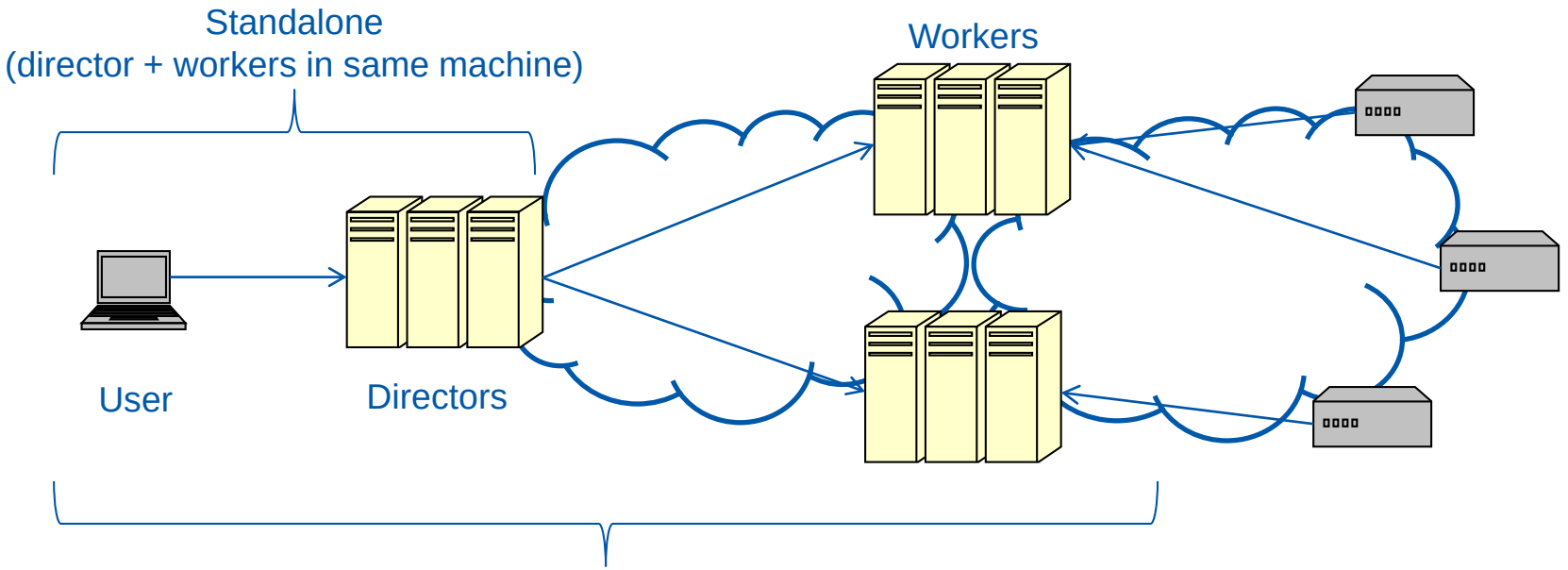
- Note:
  - We're working on a Unix Terminal (Linux, BSD, Mac, Bash on Windows...)

- Download the software:
  - Let's get the latest compiled version (e.g. 2.2.1 version)
  - Via browser:
    - `http://spark.apache.org/downloads.html`
  - Via console:
    - `wget http://bit.ly/2AxxoEB`

- Then unpack it:
  - `tar xvzf spark-2.2.1-bin-hadoop2.7.tgz`

- Spark runs on Java (JDK v.8). If you don't have it, get it here:
  - `http://bit.ly/1onrImb`

Step 0: Setting up the Environment



- What we will do now:
  - We will use spark in “standalone mode”, using our computing as single node
- What we could do now:
  - Set up HDFS in a cluster
  - Set up YARN to connect all SPARK nodes
  - Set up all SPARK nodes to find YARN ans synchronize

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 2.2.1
      /_/

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 = “hello”`
    - `var b = “bye”`
    - `b = “good bye”`
  - We use RDDs as “Resilient Distributed Datasets”
    - 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
  - Download the linkage dataset, from CLI:
    - `mkdir linkage`
    - `cd linkage/`
    - `wget http://bit.ly/1Aoywaq --output-document=donation.zip`
    - `unzip donation.zip`
    - `unzip 'block_*.zip'`

* Some examples here are addapted from: Sandy Ryza et al. “Advanced Analytics with Spark”

Spark and Scala

- Accessing the data:
  - This dataset is already partitioned, and it can be uploaded to HDFS
    - `hdfs dfs -mkdir /linkage`
    - `hdfs dfs -put block_*.csv /linkage`
  - ...or we can access all in local
    - Fuse all blocks in one
      - `cat block_*.csv > block_all.csv`
    - We're doing this now, so all operations will be done locally
- Opening Session:
  - `./bin/spark-shell --master local[*]`
- Load blocks:
  - HDFS:
    - `val rawblocks = sc.textFile("hdfs://linkage")`
  - Local:
    - `val rawblocks = sc.textFile("file:///[YOUR PATH]/linkage/block_all.csv")`

- 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,?,1,1,1,1,1,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. We have our value “head”
  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. We apply the “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`

  et cetera…

- 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,TRUE
    ```

  - `val line = head(5)`
  - `val pieces = line.split(',')`
  - `val id1 = pieces(0).toInt`
  - `val id2 = pieces(1).toInt`
  - `val matched = pieces(11).toBoolean`

  - `def toDouble(s: String) = { if ("?".equals(s)) Double.NaN else s.toDouble }`

  Beware ↓: counting starts at 0 so 2 is "3rd pos.", and here "11" is not fetched

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

- 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).map(toDouble)
      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, 1.0),
      true)
    ```

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
    ...
```

- 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(_._2,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(("",71), (the,24), ...
    ```
  - `ranking.take(5).foreach(println)`

    ```
    (,71)
    (the,24)
    (to,17)
    (Spark,16)
    (for,12)
    ```

Map/Reducing Operations

- We can process the results
  - Also we can include data process, e.g. 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)`

Part 1 – Recap

- What is Spark
- Where/when are things computed in Spark
- Installing the Spark environment
- Spark and Scala:
  - Creation of values
  - Reading files from local/HDFS as RDDs
  - Show and filter data
  - Creating functions and objects/classes
- Map/Reduce operations

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

Step 0: Prepare the Environment

- Let's prepare HDFS
  - To work with distributed files (even if in local)
  - Use SparkSQL by working on distributed data.frames

- Prepare the system:
  - Install the HDFS
  - **[You should have come with this done...]**

- Launch HDFS:
  - `./hadoop/sbin/start-dsf.sh`

- Here you should see your HDFS system:
  - `hdfs dfs –ls /`

Step 0: Prepare the Environment

- Let's get some Big Data
  - ...compared to previous examples
  - `wget http://bit.ly/2jZgeZY && unzip csv_hus.zip`
  - Putting data into HDFS:
    - `hdfs dfs –mkdir /hus`
    - `hdfs dfs -put ss13husa.csv /hus/`
  - We should see it in
    - `hdfs dfs –ls /hus`

- Before starting: Locate our HDFS location:port
  - sudo netstat -plten | grep java

```
tcp 0 0 127.0.0.1:54310 0.0.0.0:* LISTEN 1000 6090571
  22603/java
```

  - We can also check Hadoop's configuration for fs.default.name
  - Remember it! We are connecting to this FileSystem.

Loading HDFS data

- 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("hdfs://localhost:54310/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,nul
  l,null,null,null,null,null,null,null,null,null,null,null,2
  ,null,null,null,null,null,null,null,null,null,null,null,nu
  ll,null,null,null,null,null,null,null...
```

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 Data.Frame
  - `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 DataFrames

- Selecting Data (column selection)
  - `df.select($"SERIALNO",$"RT",$"DIVISION",$"REGION").show()`

```
+--------+---+--------+------+
|SERIALNO| RT|DIVISION|REGION|
+--------+---+--------+------+
|      84|  H|       6|     3|
|     154|  H|       6|     3|
|     156|  H|       6|     3|
|     160|  H|       6|     3|
|     231|  H|       6|     3|
|     ... |...|   ...  |  ... |
|     776|  H|       6|     3|
|     891|  H|       6|     3|
|     944|  H|       6|     3|
|    1088|  H|       6|     3|
|    1117|  H|       6|     3|
|    1242|  H|       6|     3|
+--------+---+--------+------+
only showing top 20 rows
```

Relational DataFrames

- Filtering Data (row selection)
  - `df.select($"SERIALNO",$"RT",$"DIVISION",$"REGION").filter($"PUMA" > 2600).show()`

```
+--------+---+--------+------+
|SERIALNO| RT|DIVISION|REGION|
+--------+---+--------+------+
|     154|  H|       6|     3|
|     156|  H|       6|     3|
|     160|  H|       6|     3|
|     ... |...|   ...  |  ... |
|     944|  H|       6|     3|
|    1117|  H|       6|     3|
|    1242|  H|       6|     3|
|    1369|  H|       6|     3|
|    1779|  H|       6|     3|
|    1782|  H|       6|     3|
|    1791|  H|       6|     3|
+--------+---+--------+------+
only showing top 20 rows
```

Relational DataFrames

- Grouping Data
  - `df.groupBy("DIVISION").count().show()`

```
+--------+------+
|DIVISION| count|
+--------+------+
|       1| 58103|
|       6| 60389|
|       3|139008|
|       5|179043|
|       9|163137|
|       4| 55641|
|       8| 63823|
|       7| 36921|
+--------+------+
```

SQL DataFrames

- To use SQL, creating a temporal View (preserved across sessions):
  - `df.createGlobalTempView("husa")`

- Select using SQL:
  - `spark.sql("SELECT SERIALNO, RT, DIVISION, REGION FROM global_temp.husa").show()`

```
+--------+---+--------+------+
|SERIALNO| RT|DIVISION|REGION|
+--------+---+--------+------+
|      84|  H|       6|     3|
|     154|  H|       6|     3|
|     156|  H|       6|     3|
|     ...|...|   ...  |  ... |
|     944|  H|       6|     3|
|    1088|  H|       6|     3|
|    1117|  H|       6|     3|
|    1242|  H|       6|     3|
+--------+---+--------+------+
only showing top 20 rows
```

SQL DataFrames

- Filtering using SQL:
  - `spark.sql("SELECT SERIALNO, RT, DIVISION, REGION FROM global_temp.husa WHERE PUMA < 2100").show()`

```
+--------+---+--------+------+
|SERIALNO| RT|DIVISION|REGION|
+--------+---+--------+------+
|     154|  H|       6|     3|
|     156|  H|       6|     3|
|     160|  H|       6|     3|
|   ...  |...|   ...  |  ... |
|     944|  H|       6|     3|
|    1117|  H|       6|     3|
|    1242|  H|       6|     3|
|    1369|  H|       6|     3|
|    1779|  H|       6|     3|
|    1782|  H|       6|     3|
|    1791|  H|       6|     3|
+--------+---+--------+------+
only showing top 20 rows
```

SQL DataFrames

- Grouping using SQL:
  - `spark.sql("SELECT DIVISION, COUNT(*) FROM global_temp.husa GROUP BY DIVISION").show()`

```
+--------+--------+
|DIVISION|count(1)|
+--------+--------+
|       1|   58103|
|       6|   60389|
|       3|  139008|
|       5|  179043|
|       9|  163137|
|       4|   55641|
|       8|   63823|
|       7|   36921|
+--------+--------+
```

Relational side-by-side SQL DataFrames

- 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()`

DDF Transformations and Storage

- SparkSQL results can be stored
  - Parquet: a columnar format widely supported

Save our DDF into Parquet, then load again

```
df.write.parquet("hdfs://localhost:54310/husa.parquet")
val pqFileDF = spark.read.parquet(
"hdfs://localhost:54310/husa.parquet")
```

Also, Parquet DFs can be used directly like regular DFs

```
pqFileDF.createOrReplaceTempView("parquetFile")
val namesDF = spark.sql("SELECT SERIALNO FROM
parquetFile WHERE PUMA < 2100")
namesDF.map(attributes => "SerialNo: " +
attributes(0)).show()
```

- Finally, if using SparkR, selection, filtering and grouping by can be done in R style.

Part 2 - Recap

- Setting up the Hadoop Distributed FileSystem (HDFS)

- Operations Using SparkSQL
  - Relational Algebra functions
    - Selecting
    - Filtering
    - Grouping
  - SQL usual functions
    - Same as above
  - Comparison between two styles

- Storage formats like Parquet

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
    - We'll see the Least Squares, also the Support Vector Machines
  - Clustering
    - We'll see the k-means
  - Dimension Reduction
    - We'll see Principal Component Analysis
  - Collaborative Filtering
  - Frequent Pattern Mining

* Some examples here are addapted from the Apache Spark introduction tutorials

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, one labeled “1”, the other “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)`

Basic Statistics with SparkML

Summaries

- `import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary, Statistics}`

We create a RDD dataset

- `val vd1 = Vectors.dense(1.0, 10.0, 100.0)`
- `val vd2 = Vectors.dense(2.0, 20.0, 200.0)`
- `val vd3 = Vectors.dense(5.0, 33.0, 366.0)`
- `val data = sc.parallelize(Seq(vd1, vd2, vd3))`

Then we create a Summary

- `val summary: MultivariateStatisticalSummary = Statistics.colStats (data)`

The RDD summary contains the column-wise stats for our examined data:

- count, max, mean, min, variance, numNonZeros, normL1, normL2

- `summary.mean`

Basic Statistics with SparkML

Correlations

- `import org.apache.spark.mllib.linalg._`
- `import org.apache.spark.mllib.stat.Statistics`
- `import org.apache.spark.rdd.RDD`

We create two series to check for correlation

- `val seriesX: RDD[Double] = sc.parallelize(Array(1,2,3,3,5))`
- `val seriesY: RDD[Double] = sc.parallelize(Array(11,22,33,33, 555))`

We compute the pearsons correlation (spearman also available)

- `val correlation: Double = Statistics.corr(seriesX, seriesY, "pearson")`
- `println(s"Correlation is: $correlation“)`

```
Correlation is: 0.8500286768773001
```

Also we can compute correlation matrices (with previous data)

- `val correlMatrix: Matrix = Statistics.corr(data, "pearson")`

Basic Statistics with SparkML

Generate Random Data

```
– import org.apache.spark.mllib.random.RandomRDDs._
```

1 Million random doubles, from ~N(0,1), distributed in 10 partitions

```
– val u = normalRDD(sc, 1000000L, 10)
– u.take(1)
    res57: Array[Double] = Array(0.1902285426523768 8)
```

Let's consider ~N(1,4) instead by transforming our randomized RDD

```
– val v = u.map(x => 1.0 + 2.0 * x)
– v.take(1)
    res58: Array[Double] = Array(1.3804570853047538)
```

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 = 7.451032810102601 5
```

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 using 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")

Dimensionality Reduction

Principal Component Analysis

- `import org.apache.spark.mllib.linalg.{Matrix, Vectors}`
- `import org.apache.spark.mllib.linalg.distributed.RowMatrix`

Create some sample data, and parallelize

```
- val data = Array(
  Vectors.sparse(5, Seq((1, 1.0), (3, 7.0))),
  Vectors.dense(2.0, 0.0, 3.0, 4.0, 5.0),
  Vectors.dense(4.0, 0.0, 0.0, 6.0, 7.0))

- val dataRDD = sc.parallelize(data, 2)

- val mat: RowMatrix = new RowMatrix(dataRDD)
```

Dimensionality Reduction

Principal Component Analysis

Compute the Principal Components

```
– val pc: Matrix = mat.computePrincipalComponents(4)
    pc: org.apache.spark.mllib.linalg.Matrix =
    -0.44859172075  -0.28423808214   0.083445452575   0.83641020094
     0.13301985745  -0.05621155904   0.044239792581   0.17224337841
    -0.12523156359   0.763626477466  -0.578071228563   0.25541548866
     0.21650756651  -0.56529587735   -0.795540506278   4.85812142E-5
    -0.84765129311  -0.11560340501   -0.155011789143  -0.45333554916
```

Project Points into the new space

```
– val projected: RowMatrix = mat.multiply(pc)
– projected.rows.collect.foreach(println)
    [1.6485728230883,-4.0132827005162,-5.5245437513693,0.17258344691630]
    [-4.645104331781,-1.1167972663619,-5.5245437513693,0.17258344691630]
    [-6.428880535676,-5.3379514277753,-5.5245437513693,0.17258344691630]
```

Part 3 – Recap

- What is SparkML a.k.a. MLlib

- Some examples of :
  - Types of Data in Mllib
  - Basic Statistics that can be performed
    - Summary
    - Correlation
    - Random Number Generation
  - Modeling and prediction
    - Least Squares
    - Support Vector Machines
  - Clustering
    - K-Means
  - Dimensionality Reduction
    - PCA

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

SparkStream

- In Spark versions prior to v2.2, streams required us to create a thread per stream, then create a function to update the "steady" thread
  - We created a StreamContext to periodically read
  - We defined the functions to be applied in the stream context
  - We created a function to update the steady context for each stream update

- From Spark v2.2 on we have "structured streaming"
  - We define which query will be executed, and the periodicity
  - We define where to dump the results
    - This can be a variable in the "steady" thread
    - Or can be another stream opened for writing

Structured Streaming

- Load the required libraries
  - `import org.apache.spark.sql.functions._`
  - `import org.apache.spark.sql.SparkSessionval`
  - `import spark.implicits._`

- Streams for reading (wordcount example)

  Create the stream of input lines from connection to localhost:9999
  - `val lines = spark.readStream.format("socket").option("host", "localhost").option("port", 9999).load()`

  Write the wordcount example
  - `val words = lines.as[String].flatMap(_.split(" "))`
  - `val wordCounts = words.groupBy("value").count()`

- Streams for writting

  Start running the query that prints the running counts to the console
  - `val query = wordCounts.writeStream.outputMode("complete").format("console")`

Structured Streaming

- Start another terminal

  Let's use "nc" to send words to our Spark streaming context. In a bash terminal, run:
  - `nc -lk 9999`

  We opened the streaming context in host: localhost and port: 9999

- Back to the Spark terminal

  We start the thread for streaming
  - `query.start()`

  We tell the thread to wait until input stream associated to query stops
  - `query.awaitTermination()`

- In the Bash Terminal

  Just start introducing words, separated by space (as we indicated in the "map" operation). E.g.:
  - `hola adeu`
  - `hola que tal`
  - `adeu siau`
  - `hola hola hola`

Structured Streaming

- Back to the Spark terminal

  Our streaming context is performing wordcount continuously

```
-------------------------------------------
Batch: 0
-------------------------------------------
+-----+-----+
|value|count|
+-----+-----+
| hola|    1|
| adeu|    1|
+-----+-----+

-------------------------------------------
Batch: 1
-------------------------------------------
+-----+-----+
|value|count|
+-----+-----+
|  que|    1|
| hola|    5|
| siau|    1|
| adeu|    2|
|  tal|    1|
+-----+-----+
```

Structured Streaming

- Stream sources (readers)
  - sockets: TCP sockets, options are host, port, etc...
  - applications: kafka, flume, kinesis... (have their own options, like “server”, “topic”, “user”, etc...)
  - files: files in a directory, e.g. as CSV
  - memory: a table in SparkSQL
- Stream sinks (writters)
  - console: direct to terminal
  - sockets: dump into TCP sockets
  - applications: dump into application listeners
  - files: dump results into a file (in the FS, HDFS, etc...)
  - memory: a table in SparkSQL
- The Spark application can build a pipeline for processing data
  - Receive from multiple sources
  - Process inputs
  - Dump results into multiple sinks

Structured Streaming

Stream Periodicity and updates

```
– import org.apache.spark.sql.functions._
– import org.apache.spark.sql.SparkSession
– import spark.implicits._
– import org.apache.spark.sql.streaming.ProcessingTime

– val lines = spark.readStream.format("socket").option("host",
  "localhost").option("port", 9999).load()
– val words = lines.as[String].flatMap(_.split(" "))
– val wordCounts = words.groupBy("value").count()
```

We can set the interval for processing with a “trigger”

```
– var query =
  wordCounts.writeStream.outputMode("complete").trigger(ProcessingTime("5
  seconds")).format("console")
– query.start().awaitTermination()
```

We can decide that we don’t want to keep full memory, just the updates

```
– var query =
  wordCounts.writeStream.outputMode(“update").trigger(ProcessingTime("5
  seconds")).format("console")
– query.start().awaitTermination()
```

Structured Streaming

- Stream Queries
  - Objects from Structured Streaming are SparkSQL DataFrames
  - Operations are from SparkSQL, and inputs and results be treated as tables
  - Also input streams can be joined and aggregated in time windows

- Stream Management

  Streams can be treated as Threads
  - They have Ids
  - Can be started ans stopped
  - Can wait other streams
  - Can be monitored
    - `println(query.lastProgress)`

Part 4 – Recap

- What is SparkStream
- Structured Streams
  - Spark streaming
  - Data sources and sinks
  - Stream properties
- Some examples:
  - Streaming WordCount

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 that 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 a simple “translation” of Spark to R: An API for working with Data.Frames in R over Spark

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");
```

www.bsc.es

Thanks for your Attention

Josep Ll. Berral García

Data Centric Computing - BSC