



**Barcelona  
Supercomputing  
Center**

*Centro Nacional de Supercomputación*



# Data Analytics using Apache Spark

## PATC 2019

Josep Lluís Berral García  
Data Centric Computing - BSC

February, 2019 (v.2)





## « 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 orchestrated 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
- Data Base queries

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

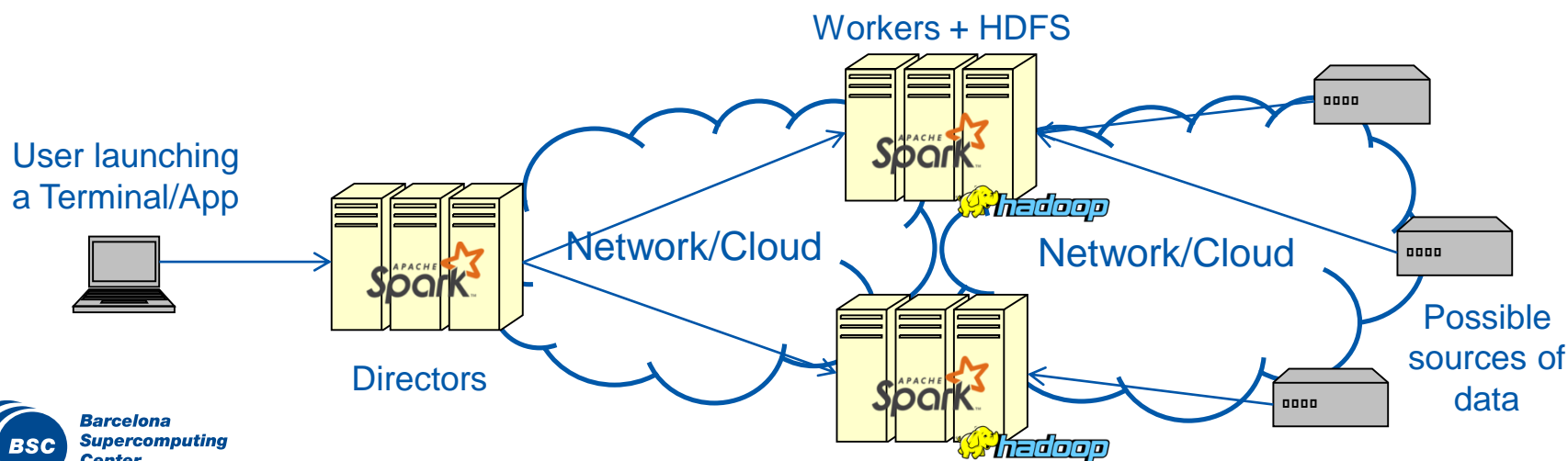
# Introduction to Apache Spark

## Cluster Computing Framework

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

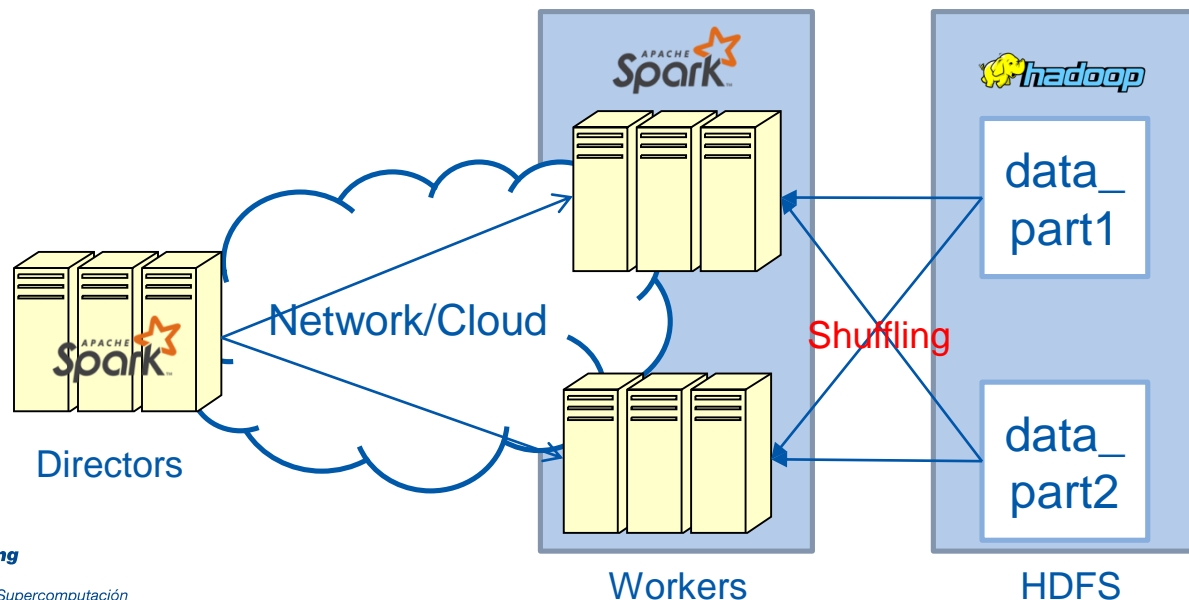
## Deployment:

- 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



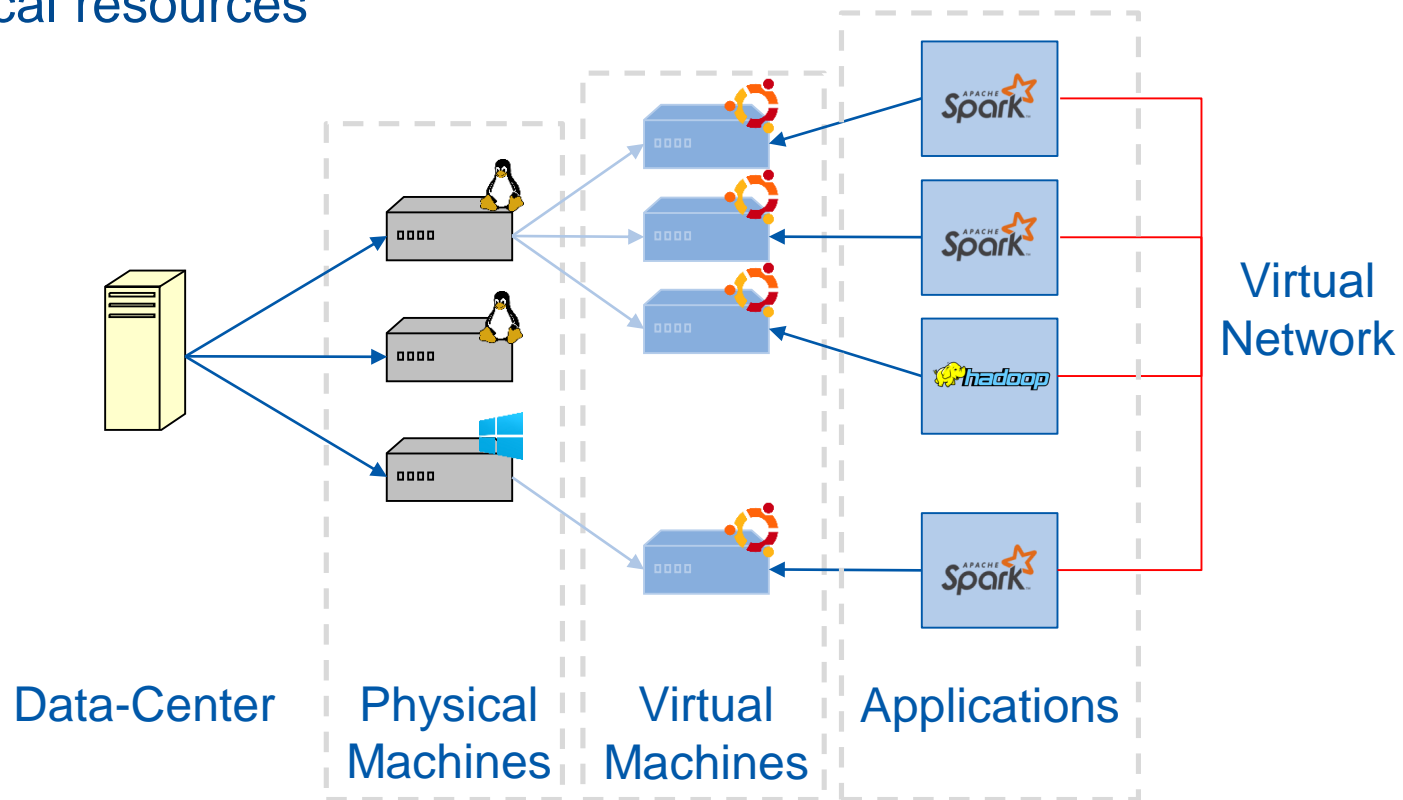
## Computing and Shuffling

- 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



## « Data-Center environments

- Virtualization: Allows running systems isolated, move systems, resize systems... independently from the base system
- The Cloud and DC-farms use Virtual Machines to allocate clients into physical resources



# Step 0: Installing the Environment

## ⌘ Note:

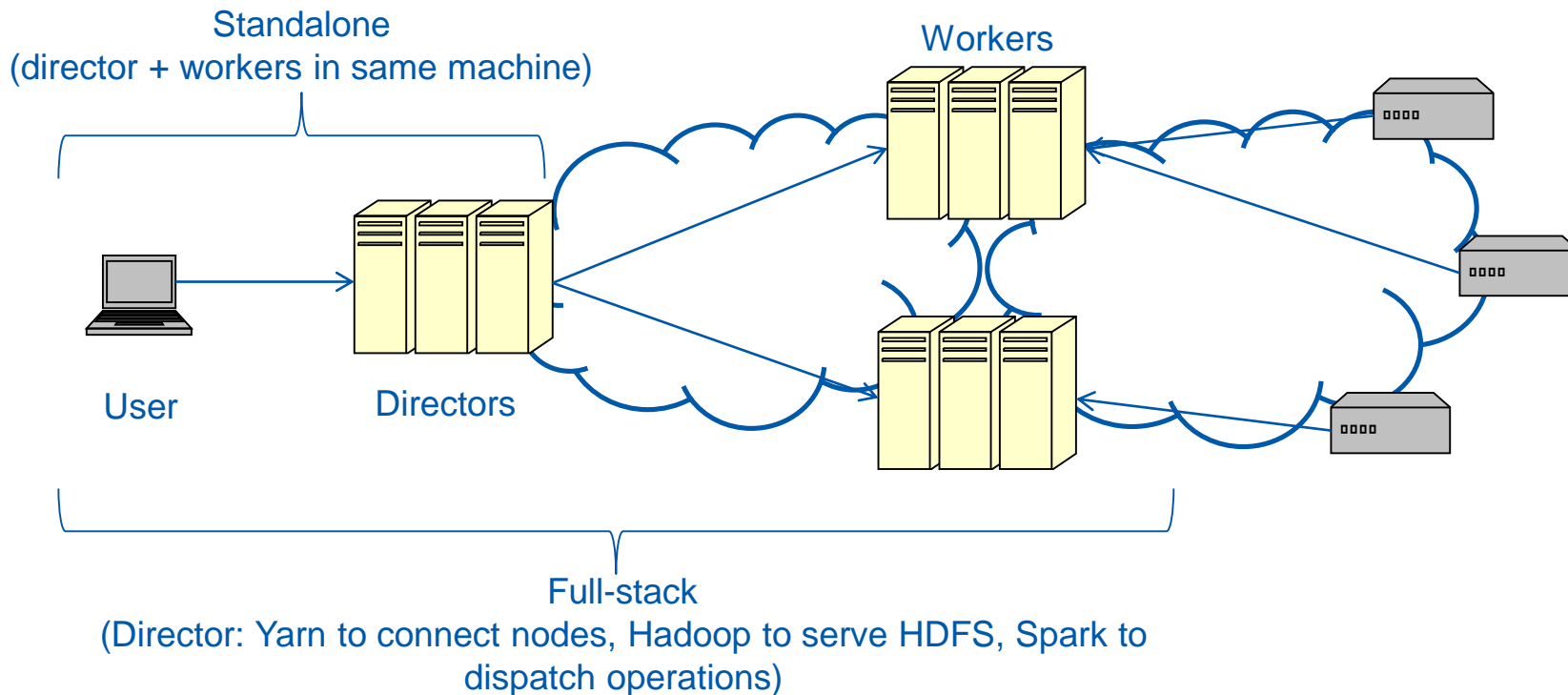
- We're working on a Unix Terminal
- You should have installed HDFS, OpenSSH and JAVA, in your VM or bare system, as indicated previously.

## ⌘ Download the software:

- Let's get the latest compiled version (e.g. 2.4.0 version)
  - `wget https://bit.ly/2HCwQ9b -O spark-2.4.0-bin-hadoop2.7.tgz`
- Then unpack it:
  - `tar xvzf spark-2.4.0-bin-hadoop2.7.tgz`
  - `ln -s spark-2.4.0-bin-hadoop2.7 spark`
- Set the environment, adding to `~/.bashrc`:
  - `export SPARK_HOME=/home/vagrant/spark`
  - `export PATH=$PATH:$SPARK_HOME/bin`
- Then, reload the environment:
  - `source ~/.bashrc`



# 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 and synchronize



## 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 can check the content of variables, values and references

- `b`  
`res0: String = 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)

## ⌘ Prepare a dataset (from CLI):

### – Download the linkage dataset:

- `wget http://bit.ly/1Aoywaq --output-document=donation.zip`

### – Create a working directory:

- `mkdir linkage`
- `mv donation.zip linkage/`
- `cd linkage/`
- `unzip donation.zip`
- `unzip 'block_*.zip'`

## ⌘ Uploading data to HDFS:

### – Start the HDFS

- `$HADOOP_HOME/sbin/start-dfs.sh`

### – 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`

## ⌘ Dimensioning the VM and the Environment:

- If you followed the “set-up document”, you already told the VM to use 4GB.
- If not, outside of the VM (stop the HDFS, exit and halt the VM), edit **Vagrantfile**, and add inside the configuration:

```
config.vm.provider "virtualbox" do |vb|  
  vb.memory = "4096"  
end
```

- ... then restart the VM, now with 4GB of RAM instead the default (1GB), and connect and start HDFS again

## ⌘ Opening Session:

- `spark-shell --master local[*] --driver-memory 2G --executor-memory 768M --executor-cores 2`

## ⌘ Load blocks:

- If using the local FS:
  - `val rawblocks = sc.textFile("file:///home/vagrant/linkage/block_all.csv")`
- If using the HDFS:
  - `val rawblocks = sc.textFile("hdfs://localhost:54310/linkage")`
- Check that data is loaded, by printing the 1<sup>st</sup> element:
  - `rawblocks.first()`

## 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`

## « 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

## ⌘ 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`

## 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)`

## 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("/home/vagrant/spark/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)
```

⌘ 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)
```



## “ 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)
```

« 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

## « 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

# Prepare the Data-Sets

## ⌘ Now, let's work with HDFS (if you weren't yet)

- **You should have come with HDFS installed**
- Launch HDFS if you didn't yet
  - `$HADOOP_HOME/sbin/start-dfs.sh`

## ⌘ Let's get some Big Data, (compared to previous examples)

- Un-compress the data:
  - `wget http://bit.ly/2jZgeZY -O csv_hus.zip`
  - `mkdir hus`
  - `mv csv_hus.zip hus/`
  - `cd hus`
  - `unzip csv_hus.zip`
- Put data into the HDFS
  - `hdfs dfs -mkdir /hus`
  - `hdfs dfs -put ss13husa.csv /hus/`
- We should see it in
  - `hdfs dfs -ls /hus`

## ⌘ Remember the port for HDFS (shown in core-site.xml, e.g. localhost:54310)

# Loading HDFS data

- ⌘ The file we downloaded contains some CSV files
  - We are going to open one, directly as a DataFrame

- ⌘ 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,2  
  ,null,null,null,null,null,null,null,null,null,null,nu  
  ll,null,null,null,null,null,null,null, null...)
```

## Let's print the DataFrame 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)
| . . .
```

## ⌘ 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

## ⌘ 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



## 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

## 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|
+-----+-----+
```

⌘ 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

## 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

## ⌘ 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|  
+-----+-----+
```

## ⌘ 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()`

## ⌘ 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()`

## Operations Using SparkSQL

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

## Storage formats like Parquet



## SparkML

- This is the Machine Learning library for Spark (spark.ml)
- Before versions 2.3, library MLlib was used (spark.mllib)

## Distributed ML Algorithms

- Basic Statistics
  - Summaries, Correlations, Hypothesis Tests...
- Classification and Prediction
  - We'll see the Linear Regression, 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

- ⌋ 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.ml.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.ml.feature.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)))
```

## ⌘ Matrices

```
- import org.apache.spark.ml.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))
```

## ⌘ Visit values on matrices

### We can iterate rows and columns

```
- dm.colIter.foreach(println)  
- dm.rowIter.foreach(println)
```

## Summaries

```
- import org.apache.spark.ml.linalg.{Vector, Vectors}
- import org.apache.spark.ml.stat.Summarizer
```

### We create a DataFrame

```
- val data = Seq(
  (Vectors.dense(2.0, 3.0, 5.0), 1.0),
  (Vectors.dense(4.0, 6.0, 7.0), 2.0)
)
- val df = data.toDF("features", "weight")
- df.show()
```

```
+-----+-----+
|   features|weight|
+-----+-----+
|[2.0,3.0,5.0]|  1.0|
|[4.0,6.0,7.0]|  2.0|
+-----+-----+
```

### Then we create a Summary

```
- val (meanVal, varianceVal) = df.select(Summarizer.metrics("mean", "variance")
  .summary($"features", $"weight").as("summary")).select("summary.mean", "summary.variance")
  .as[(Vector, Vector)].first()
meanVal: ...Vector = [3.333333333333333,5.0,6.333333333333333]
varianceVal: ...Vector = [2.0,4.5,2.0]

- val (meanVal2, varianceVal2) = df.select(Summarizer.mean($"features"),
  Summarizer.variance($"features")).as[(Vector, Vector)].first()
meanVal2: org.apache.spark.ml.linalg.Vector = [3.0,4.5,6.0]
varianceVal2: org.apache.spark.ml.linalg.Vector = [2.0,4.5,2.0]
```

## Correlations

```
- import org.apache.spark.ml.linalg.{Matrix, Vectors}
- import org.apache.spark.ml.stat.Correlation
- import org.apache.spark.sql.Row
```

### We create four series to check for correlation

```
- val data = Seq(
  Vectors.sparse(4, Seq((0, 1.0), (3, -2.0))),
  Vectors.dense(4.0, 5.0, 0.0, 3.0),
  Vectors.dense(6.0, 7.0, 0.0, 8.0),
  Vectors.sparse(4, Seq((0, 9.0), (3, 1.0)))
)
- val df = data.map(Tuple1.apply).toDF("features")
```

### We compute the Pearson Correlation

```
- val Row(coeff1: Matrix) = Correlation.corr(df, "features").head
- println(s"Pearson Correlation matrix:\n $coeff1")
  1.0          0.055641488407465814  NaN  0.4004714203168137
  0.055641488407465814  1.0          NaN  0.9135958615342522
  NaN          NaN          1.0  NaN
  0.4004714203168137  0.9135958615342522  NaN  1.0
```

### Also we can compute the Spearman Correlation

```
- val Row(coeff2: Matrix) = Correlation.corr(df, "features", "spearman").head
- println(s"Spearman correlation matrix:\n $coeff2")
  1.0          0.10540925533894532  NaN  0.400000000000000174
  0.10540925533894532  1.0          NaN  0.9486832980505141
  NaN          NaN          1.0  NaN
  0.400000000000000174  0.9486832980505141  NaN  1.0
```

## Generate Random Data

```
- import org.apache.spark.util.random
```

### Bernoulli and Poisson Samplers

```
- val bs = new random.BernoulliSampler(0.3)
```

```
- bs.sample()
```

```
res157: Int = 0
```

```
- val ps = new random.PoissonSampler[Double](0.6)
```

```
- ps.sample()
```

```
res161: Int = 1
```

At this time, for Gaussian samples, we must go for RDDs (later)

## Linear Regression

```
- import org.apache.spark.ml.regression.LinearRegression
```

### Load the Data

```
- var datafile =  
  "/home/vagrant/spark/data/mllib/sample_linear_regression_data.txt"  
- val dataset = spark.read.format("libsvm").load(datafile)
```

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

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

### Then we train our model

```
- val lr = new  
  LinearRegression().setMaxIter(10).setRegParam(0.3).setElasticNetParam(0.8)  
- val model = lr.fit(training)  
  
- println(s"Coefficients: ${model.coefficients} Intercept:  
  ${model.intercept}")  
  Coefficients: [0.0,0.322925166774,-0.343854803456,1.915601702345,  
  0.052880586803,0.765962720459,0.0,-0.151053926691,-0.215879303609,  
  0.220253691888] Intercept: 0.159893684423
```



```
« ...  
- import org.apache.spark.ml.regression.LinearRegressionModel  
- import org.apache.spark.ml.evaluation.RegressionEvaluator
```

## We can pass the Test set

```
- val predictions = model.transform(test)  
- predictions.show()  
+-----+-----+-----+  
|          label|          features|          prediction|  
+-----+-----+-----+  
|-26.805483428483072| (10, [0, 1, 2, 3, 4, 5, ...]| 0.7396609342028824|  
|-26.736207182601724| (10, [0, 1, 2, 3, 4, 5, ...]| -1.9523217339135148|  
| ... | ... | ... |  
|-12.467656381032860| (10, [0, 1, 2, 3, 4, 5, ...]| -3.2321660582830720|  
+-----+-----+-----+
```

## Then we evaluate the predictor

```
- val predictionAndLabels = predictions.select("prediction", "label")  
- val evaluator = new RegressionEvaluator().setMetricName("mse")  
- val lm_eval = evaluator.evaluate(predictionAndLabels)  
- println(s"Test set accuracy = ${lm_eval}")  
  Test set accuracy = 121.31149228612746
```

## Finally, we can save the model

```
- model.write.overwrite().save("LR_Model")  
- val sameModel = LinearRegressionModel.load("LR_Model")
```

## Support Vector Machines

- `import org.apache.spark.ml.classification.LinearSVC`
- `import org.apache.spark.ml.classification.LinearSVCModel`

### Load the data

- `var datafile =  
 "/home/vagrant/spark/data/mllib/sample_libsvm_data.txt"`
- `val dataset = spark.read.format("libsvm").load(datafile)`

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

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

### Train the model with the Training Set

- `val lsvc = new LinearSVC().setMaxIter(10).setRegParam(0.1)`
- `val model = lsvc.fit(training)`

### Apply the model to predict the Test Set

- `val predictions = model.transform(test)`
- `predictions.show()`

```
« ...  
- import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
```

## We create an “accuracy” evaluator

```
- val evaluator = new MulticlassClassificationEvaluator().setMetricName("accuracy")
```

## Then evaluate the predictions

```
- val predictionAndLabels = predictions.select("prediction", "label")  
- predictionAndLabels.show()
```

```
+-----+-----+  
|prediction|label|  
+-----+-----+  
|          0.0|  0.0|  
|          0.0|  0.0|  
|      ...    |  ... |  
|          1.0|  1.0|  
|          1.0|  1.0|  
+-----+-----+
```

```
- val svc_eval = evaluator.evaluate(predictionAndLabels)  
- println(s"Test set accuracy = ${svc_eval}")  
  Test set accuracy = 1.0
```

## And we save the model

```
- model.write.overwrite().save("SVM_SGD_Model")  
- val sameModel = LinearSVCModel.load("SVM_SGD_Model")
```

## Our beloved classical k-means

- `import org.apache.spark.ml.clustering.KMeans`
- `import org.apache.spark.ml.clustering.KMeansModel`
- `import org.apache.spark.ml.evaluation.ClusteringEvaluator`

### First of all, load and parse the data

- `var datafile =`  
`"/home/vagrant/spark/data/mllib/sample_kmeans_data.txt"`
- `val dataset = spark.read.format("libsvm").load(datafile)`

### Then train a model

- `val kmeans = new KMeans().setK(2).setSeed(1L)`
- `val model = kmeans.fit(dataset)`
- `model.clusterCenters.foreach(println)`

### Use for prediction

- `val predictions = model.transform(dataset)`
- `predictions.show()`

« ...

We can evaluate our model using Silhouette score

- val evaluator = new ClusteringEvaluator()
- val silhouette = evaluator.evaluate(predictions)  
silhouette: Double = 0.9997530305375207
- println(s"Silhouette with squared euclidean distance =  
\$silhouette")  
Silhouette with squared euclidean distance = 0.999753030...

Also we can use evaluate our model using the Sum of Squared Errors

- val WSSSE = model.computeCost(dataset)  
WSSSE: Double = 0.1199999999999994547
- println("Within Set Sum of Squared Errors = " + WSSSE)  
Within Set Sum of Squared Errors = 0.1199999999999994547

Finally, we save the model

- model.write().overwrite().save("KMeansModel")
- val sameModel = KMeansModel.load("KMeansModel")

## Principal Component Analysis

- `import org.apache.spark.ml.feature.PCA`
- `import org.apache.spark.ml.linalg.Vectors`
- `import org.apache.spark.sql.Session`

### 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 dataframe =  
 spark.createDataFrame(data.map(Tuple1.apply)).toDF("features")`
- `dataframe.show()`

```
+-----+  
|           features |  
+-----+  
| (5, [1, 3], [1.0, 7.0]) |  
| [2.0, 0.0, 3.0, 4.0, ...] |  
| [4.0, 0.0, 0.0, 6.0, ...] |  
+-----+
```

(( ...

## Compute the Principal Components

- `val pca = new PCA()  
 .setInputCol("features")  
 .setOutputCol("psi_features")  
 .setK(4)  
 .fit(dataframe)`
- `pca.explainedVariance`  
`res28: org.apache.spark.ml.linalg.DenseVector =`  
`[0.79439325322,0.205606746776,1.25729185411E-16,5.38535301254E-17]`

## Project Points into the new space

- `val result = pca.transform(dataframe).select("psi_features")`
- `result.show(truncate = false)`

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|psi_features|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|[1.6485728230883807,-4.0132827005162960, . . . ]|
|[-4.645104331781534,-1.1167972663619026, . . . ]|
|[-6.428880535676489,-5.3379514277753550, . . . ]|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

## « Tweaking Data Frames

```
- result.show(truncate = false)
+-----+
|psi_features|
+-----+
|[1.6485728230883807,-4.0132827005162960,-5.524543751369388,0.17258344691630860]|
|[-4.645104331781534,-1.1167972663619026,-5.524543751369387,0.17258344691630922]|
|[-6.428880535676489,-5.3379514277753550,-5.524543751369389,0.17258344691630967]|
+-----+
```

We need to read the Columns as Vectors, the convert to Arrays (we create a SQL User Defined Functions)

```
- val vecToArray = udf((xs: org.apache.spark.ml.linalg.Vector) => xs.toArray)
- val aux = result.withColumn("sep_features", vecToArray($"psi_features"))
```

We create a SQL expression

```
- val header = Array("psi_1", "psi_2", "psi_3", "psi_4")
- val sqlExpr = header.zipWithIndex.map{
  case (alias, idx) => col("sep_features").getItem(idx).as(alias)
}
- aux.select(sqlExpr : _*).show()
+-----+-----+-----+-----+
|          psi_1|          psi_2|          psi_3|          psi_4|
+-----+-----+-----+-----+
|1.6485728230883807|-4.013282700516296|-5.524543751369388|0.1725834469163086|
|-4.645104331781534|-1.1167972663619026|-5.524543751369387|0.17258344691630922|
|-6.428880535676489|-5.337951427775355|-5.524543751369389|0.17258344691630967|
+-----+-----+-----+-----+
```

We save the new DataFrame

```
- val result_fix = aux.select(sqlExpr : _*)
```



- ⌘ MLlib is still active, and can be used for managing models as RDDs
- ⌘ The same example of ML seen before can be recreated using MLlib
- ⌘ Going low-level using RDDs will allow to develop pipelines when the SparkML high-level library can't provide us abstractions or hooks

# MLlib 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)))
```

# MLlib 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)))
```

# MLlib 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)
```

## « Summaries

```
- import org.apache.spark.mllib.stat.Statistics
- import
  org.apache.spark.mllib.stat.MultivariateStatisticalSummary
```

### 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
```

## “( 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, 55))`

### 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")`

## Generate Random Data

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

1 Million random doubles, from  $\sim N(0,1)$ , distributed in 10 partitions

```
- val u = normalRDD(sc, 1000000L, 10)
```

```
- u.take(1)
```

```
res57: Array[Double] = Array(0.19022854265237688)
```

Let's consider  $\sim N(1,4)$  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)
```

## 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("/home/vagrant/spark/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(5).foreach(println)`  
`(-0.4307829, [-1.63735562648104, -2.00621178480549, ...])`  
`(-0.1625189, [-1.98898046126935, -0.722008756122123, ...])`  
`(-0.1625189, [-1.57881887548545, -2.1887840293994, ...])`  
`(-0.1625189, [-2.16691708463163, -0.807993896938655, ...])`  
`(0.3715636, [-0.507874475300631, -0.458834049396776, ...])`



« ...

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")
```

## Support Vector Machines

```
- import org.apache.spark.mllib.classification.SVMWithSGD
- import org.apache.spark.mllib.classification.SVMModel
- import org.apache.spark.mllib.util.MLUtils
- import
  org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
```

### Load and parse some data

```
- val data = MLUtils.loadLibSVMFile(sc,
  "/home/vagrant/spark/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)
```

« ...

## 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")`

## Our beloved classical k-means

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

### First of all, load and parse the data

```
- var datafile = "/home/vagrant/spark/data/mllib/kmeans_data.txt"
- val data = sc.textFile(datafile)
- 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 model = KMeans.train(parsedData, numClusters, numIterations)
```

### Then we evaluate our model using the Sum of Squared Errors

```
- val WSSSE = model.computeCost(parsedData)
  WSSSE: Double = 0.1199999999999994547
```

### Finally, we save the model

```
- model.save(sc, "KMeansModel_RDD")
- val sameModel = KMeansModel.load(sc, "KMeansModel_RDD")
```

## 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)`

### 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]`

## « What is SparkML

## « 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

- ❧ 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

- “ 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



⌘ For the following exercise, we'll need to open another terminal

- If you are using the VM with vagrant, just open another terminal/cmd and create a new connection “vagrant ssh”
- If you are using the VM with the VirtualBox GUI, open a terminal/cmd and connect using a SSH client (check the “set-up document” for more details)
- If you are running Spark on a bare machine, just open a new terminal/cmd!

⌘ Now we have two terminals:

- A first terminal will have Spark running
- A second terminal will have the “nc” command sending data to Spark, via TPC sockets.

## ⌘ On the Spark session...

## ⌘ 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 writing

Start running the query that prints the running counts to the console

- `val query = wordCounts.writeStream.outputMode("complete").format("console")`

## « On the second 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 and tell the thread to wait until input stream associated to query stops

```
- query.start().awaitTermination()
```

IMPORTANT! If “nc” (or the data-source program) is not started, `query.start()` will return an exception

## « 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
```

## « 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
+-----+-----+
```

## “( 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 (writers)

- 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

## « Stream Periodicity and updates

```
- import org.apache.spark.sql.functions._
- import org.apache.spark.sql.Session
- 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()
```

## 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 and stopped
- Can wait other streams
- Can be monitored
  - `println(query.lastProgress)`

# Streaming Contexts (RDD)

- Building functions to manually treat streams
  - Before Spark 2.3 there was no structured streaming
  - After Spark 2.3, we also can manually manage streams

## RDD-based Streams

- We must indicate how stream data is going to join steady data

## Create the SparkContext

- `import org.apache.spark.rdd.RDD`
- `import org.apache.spark.SparkContext`
- `import org.apache.spark.storage.StorageLevel`
- `import org.apache.spark.streaming.{StreamingContext, Seconds, Time}`

We create a Streaming Context from the Spark Context, updating each two seconds:

- `val ssc = new StreamingContext(sc, Seconds(2))`

From now on, we will use the streaming context for receiving and treating updates, and the static context for storing aggregated information



## ⌘ The Network WordCount:

We create an input source for Lines

```
- val lines =  
  ssc.socketTextStream("localhost",  
    9999,  
    StorageLevel.MEMORY_AND_DISK_SER)
```

Then what we want to do with the inputs

```
- val words =  
  lines.flatMap(_.split(" "))  
  
- val wordCounts = words.map(x =>  
  (x,1)).reduceByKey(_ + _)  
  
- wordCounts.print()
```

At this time we told Spark *how things are* at each context update

– But the streaming is not running yet....

## ⌘ In the Spark terminal

We start the thread for streaming

```
- ssc.start()  
- ssc.awaitTermination()
```

```
-----  
Time: 1485882478000 ms  
-----
```

```
-----  
Time: 1485882480000 ms  
-----
```

```
-----  
Time: 1485882482000 ms  
-----
```

...

Now, time is running!

# Streaming Counting (RDD)



## ❧ Start another terminal

- Let's use "nc" to send words to our Spark streaming context
- We opened the streaming context in host: localhost and port: 9999

## ❧ In the Bash Terminal

We start "nc" to send characters to the stream:

- nc -lk 9999

Just start introducing words, separated by space (as we indicated in the "map" operation). E.g.:

- hola que tal
- adeu siau hola hola adeu

## ❧ Back to the Spark terminal

Our streaming context is performing word-count each 2 seconds

```
-----  
Time: 1485882376000 ms  
-----
```

```
(hola,1)  
(que,1)  
(tal,1)
```

```
-----  
Time: 1485882378000 ms  
-----
```

```
-----  
Time: 1485882380000 ms  
-----
```

```
(hola,2)  
(siau,1)  
(adeu,2)
```

## Breaking the connection:

- If the connection is broken (you shut down “nc”), Spark will complain each 2 seconds...
- But you can reconnect “nc”, and Spark will keep counting

## Stopping the context

(Try to) write back into the console [Ctrl+C twice lets you write back]

- `ssc.stop()`

## Notice that wordcount was executed in each update, but each one was independent from each other

- We are “stateless”
- We told *how things are* in each iteration, but not *how things are in general*

## Let's prepare some permanent status

We prepare again the context and wordcount function

- `val ssc = new StreamingContext(sc, Seconds(2))`
- `val lines = ssc.socketTextStream("localhost", 9999, StorageLevel.MEMORY_AND_DISK_SER)`
- `val words = lines.flatMap(_.split(" "))`
- `val wordDstream = words.map(x => (x, 1)).reduceByKey(_ + _)`

We prepare a function to add the stateless counter with the stateful one

- `val updateFunc = (values: Seq[Int], state: Option[Int]) => {  
 val currentCount = values.foldLeft(0)(_ + _)  
 val previousCount = state.getOrElse(0)  
 Some(currentCount + previousCount)  
}`

Then we indicate that our wordcount must map to the status, using our function

- `val stateDstream =  
 wordDstream.updateStateByKey[Int](updateFunc)`
- `stateDstream.print()`

## « And we start it again (and stop when you want)

- `ssc.start()`
- `ssc.awaitTermination()`

-----  
Time: 1485884876000 ms  
-----

-----  
Time: 1485884878000 ms  
-----

## Then Stop it again

- `ssc.stop()`

-----  
(hola,1)  
(que,1)  
(tal,1)  
-----

-----  
Time: 1485884880000 ms  
-----

(hola,1)  
(que,1)  
(i,1)  
(siau,1)  
(tal,2)  
(adeu,1)  
-----

-----  
Time: 1485884882000 ms  
-----

(hola,4)  
(que,1)  
(i,1)  
(siau,1)  
(tal,2)  
(adeu,1)  
-----

## « What is SparkStream

## « Structured Streams

- Spark streaming
- Data sources and sinks
- Stream properties

## « Stream Contexts

- Stateful Streaming

## « Some examples:

- Streaming WordCount

## ⌋ SparkR

Not a simple “translation” of Spark to R: An API for working with Data.Frames in R over Spark

Must install R before:

- `sudo add-apt-repository "deb http://cran.rstudio.com/bin/linux/ubuntu xenial/"`
- `sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys E084DAB9`
- `sudo apt-get update`
- `sudo apt-get install r-base`

## ⌋ Using R + Spark

Allows to be started from an R session

- ```
Sys.getenv("SPARK_HOME");  
[1] "/home/vagrant/spark"
```
- ```
library(SparkR, lib.loc = c(file.path(Sys.getenv("SPARK_HOME"), "R",  
"lib")));
```
- ```
sparkR.session(master = "local[*]", sparkConfig = list(spark.driver.memory  
= "2g", spark.executor.memory = "768M", spark.executor.cores = 2));
```

## Using SparkR

DataFrames and Apply functions are implemented over Spark

```
- h_sub <- read.parquet(x = "hdfs://localhost:54310/husa.parquet");  
- aux <- select(x = h_sub, col = c("SERIALNO", "DIVISION"))  
- aux  
  SparkDataFrame[SERIALNO:int, DIVISION:int]  
- head(aux)
```

	SERIALNO	DIVISION
1	84	6
2	154	6
3	156	6
4	160	6
5	231	6
6	286	6

```
- aux <- summarize(groupBy(h_sub, h_sub$DIVISION), count = n(h_sub$PUMA))  
- head(arrange(aux, desc(aux$count)))
```

	DIVISION	count
1	5	179043
2	9	163137
3	3	139008
4	8	63823
5	6	60389
6	1	58103



## Other examples of distributing ML tasks:

### – Prepare the Dataset IRIS:

- `dataset <- read.csv("/home/vagrant/spark/data/mllib/iris_libsvm.txt", header = FALSE, sep = " ", stringsAsFactors = FALSE);`
- `dataset[,2:5] <- t(apply(dataset[,2:5], 1, function(x) sapply(x, function(y) sapply(y, function(z) as.numeric(strsplit(z,":")[[1]][2])))));`
- `species <- data.frame(species = c("setosa","versicolor","virginica"), label = c(0,1,2));`
- `dataset[,1] <- species[dataset[,1] + 1, "species"];`
- `colnames(dataset) <- c("Species", "Sepal_Length", "Sepal_Width", "Petal_Length", "Petal_Width");`

### – Train a GLM

- `iris_df <- createDataFrame(dataset)`
- `training <- filter(iris_df, iris_df$Species != "setosa");`
- `model <- glm(Species ~ Sepal_Length + Sepal_Width, data = training, family = "binomial");`
- `summary(model)`

### – Distribute different experiments on a dataset

- `families <- c("gaussian", "poisson")`
- `train <- function(family) {`
  - `model <- glm(Sepal_Length ~ Sepal_Width + Species, data = dataset, family = family);`
  - `summary(model)``}`
- `model.summaries <- spark.lapply(families, train)`
- `print(model.summaries)`

## 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

« Using Spark + R: SparkR

« Other Libraries: Spark GraphX



**Barcelona  
Supercomputing  
Center**

*Centro Nacional de Supercomputación*



# Thanks for your Attention

Josep Ll. Berral García  
Data Centric Computing - BSC