

Big Data With Spark

Course 1

Course Contents

- ❑ Spark Concepts
- ❑ Programming with RDDs and DataFrames
- ❑ Spark Streaming
- ❑ Machine Learning with Spark

Outline

☐ Introduction

☐ Programming with RDDs

☐ Working with pair RDDs

☐ Spark With Hadoop

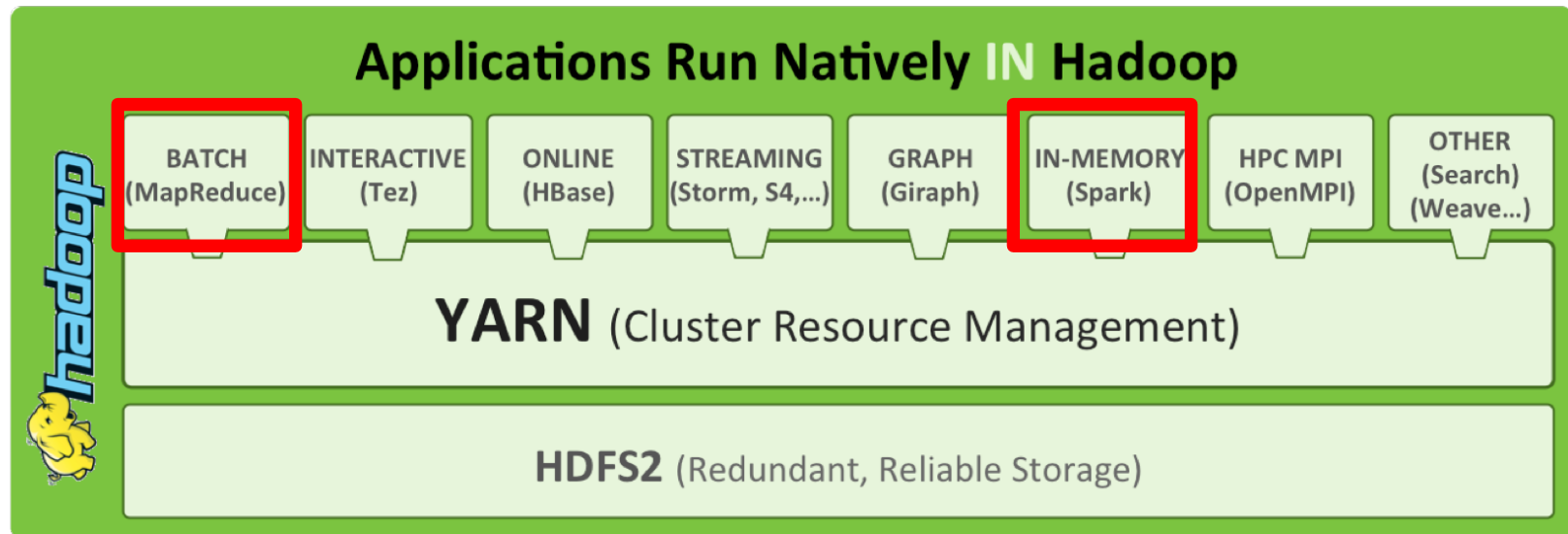
Big Data

Volume Variety Velocity
Value Veracity

Big Data Sources

- Online Online Online Online:
 - Every click, server requests, emails, advertisements, surfing ..
- User Generated Content:
 - Facebook, Twitter, Youtube, Trip Advisor...
- Graph Data
 - Social Networks; Telecommunication and Computer Networks, Road Networks, Relationships...
- Log Files
 - Web Server Logs, Machine Logs
- Internet of Things

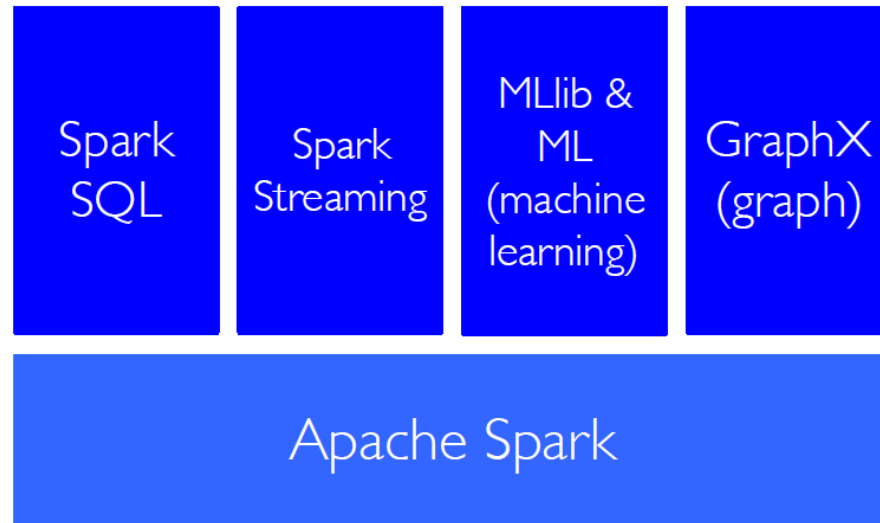
Big Data Frameworks: Hadoop Ecosystem



Spark

- Spark is an **open source in-memory** application framework for **distributed** data processing and **iterative** analysis on **massive** data volumes
- Spark allows scalable and efficient analysis of Big Data
- It was a research project in 2009, UC Berkeley, RAD lab (AMPlab later)

Big Data Frameworks: Spark



Shortcomings of Map-Reduce

- Workflow
 - ☹ Force your pipeline into Map and Reduce steps
- Iterative algorithms (i.e. machine learning)?
 - ☹ Read from disk for each MapReduce job
- Languages and Interactivity
 - ☹ Only native JAVA programming interface

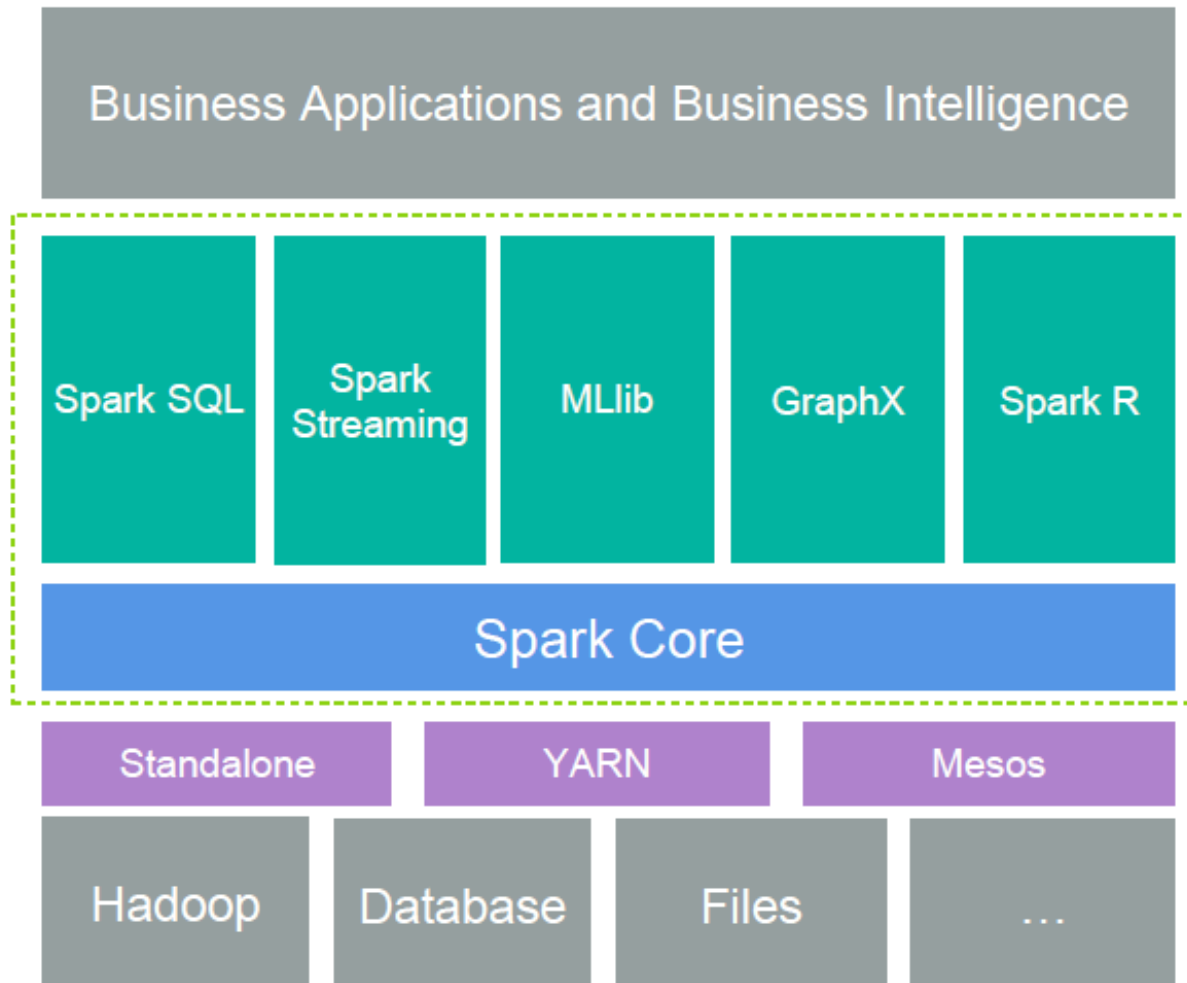
Spark The Solution

- Workflow
 - ☺ **MANY** efficient distributed operations, any combination of them
- Iterative algorithms (i.e. machine learning)?
 - ☺ in-memory caching of data, specified by the user
- Languages and Interactivity
 - ☺ Native Python, Scala (, R) interface. Interactive shells.

Spark vs MapReduce

- Apache Spark is rapidly becoming the computation engine of choice for big data.
- Spark programs are more concise and often run 10-100 times faster than Hadoop MapReduce jobs.
- Spark developers are becoming increasingly valued and sought

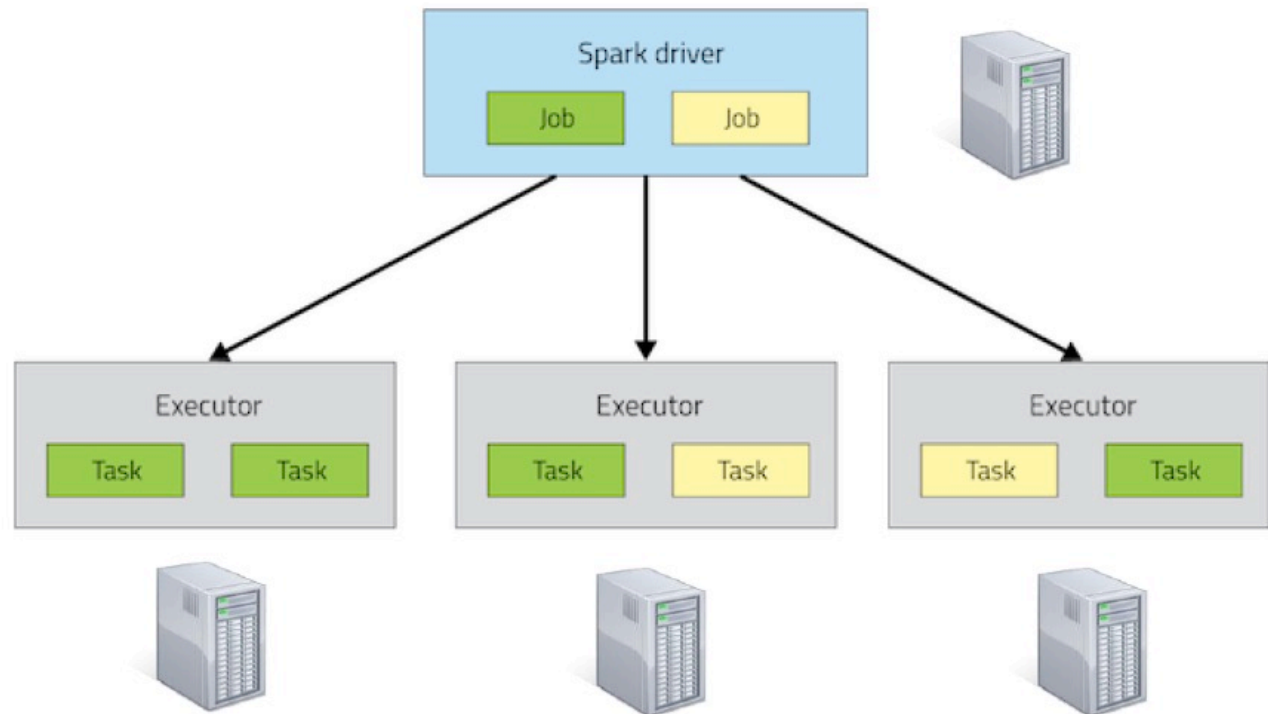
Spark on Any Data Source



Spark: A distributed Computation Framework



Spark RDD
In-memory distribution



Spark Concepts

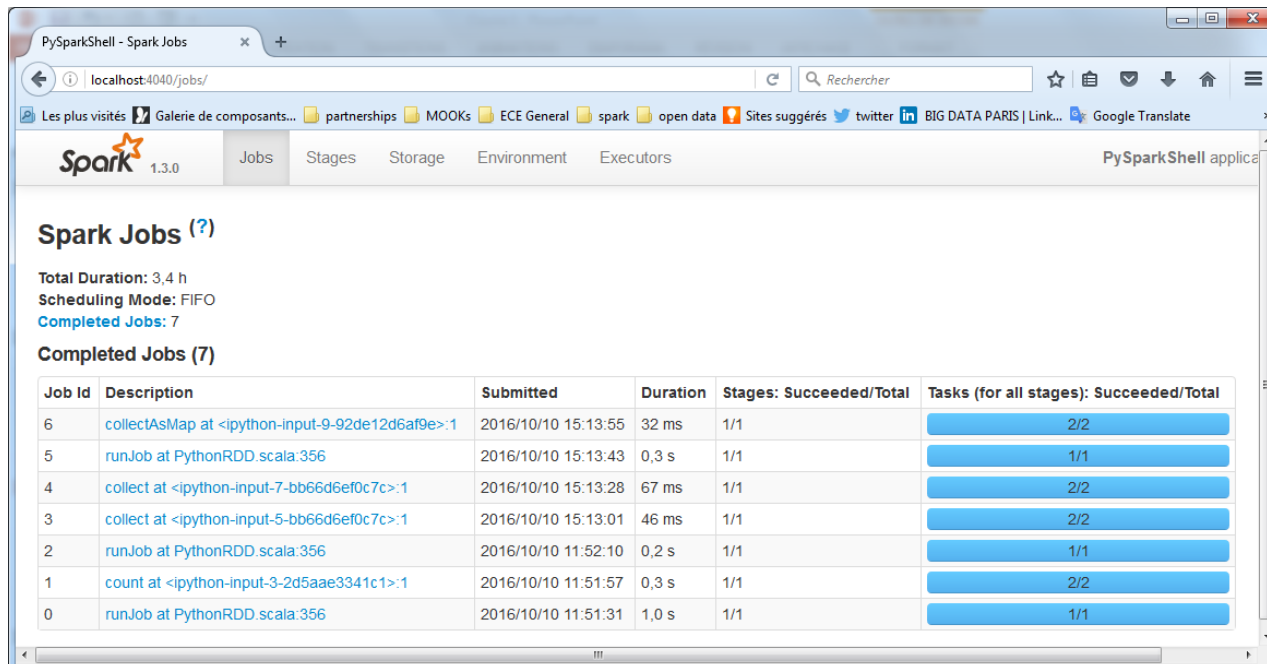
- Driver Program
 - Launches various parallel operations on a cluster
 - Contains application's main function
 - Defines distributed datasets on the cluster and applies operations to them
 - A driver Program may be a spark shell

Spark Concepts

- Spark Context
 - Driver programs access spark through a spark context
 - Spark context represents a connection to a computing cluster
 - In a shell, a SparkContext is automatically created.
 - In Batch processing programs, the SparkContext shall be explicitly created.
- Spark Session
 - Starting from Spark 2.0
 - SparkContext can be accessed from SparkSession

Spark Concepts

- Spark Jobs
 - Access to the SparkUI on **http://[ipadress]:4040**
 - All sorts of information about your tasks and cluster.

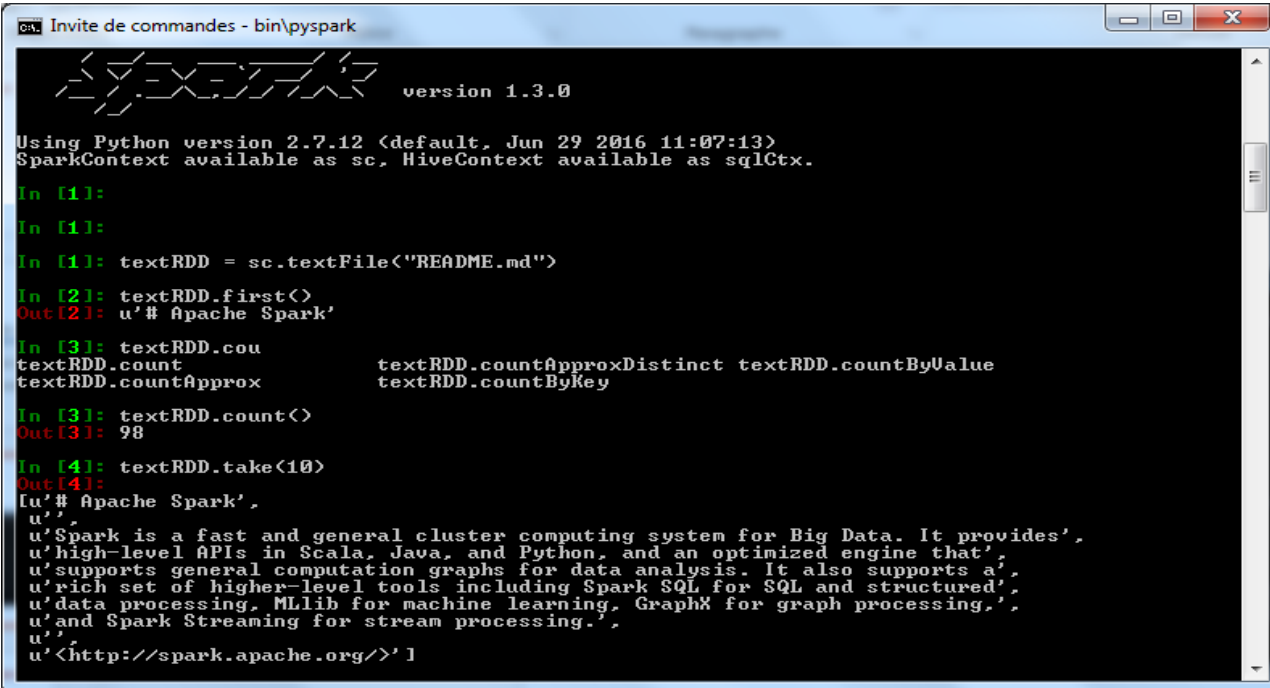


The screenshot shows the Spark UI interface in a web browser. The browser address bar displays 'localhost:4040/jobs/'. The Spark logo and version '1.3.0' are visible in the top left. The 'Jobs' tab is selected in the top navigation bar. The main content area is titled 'Spark Jobs (?)' and shows summary statistics: 'Total Duration: 3,4 h', 'Scheduling Mode: FIFO', and 'Completed Jobs: 7'. Below this, a table lists the completed jobs with columns for Job Id, Description, Submitted, Duration, Stages, and Tasks.

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
6	collectAsMap at <ipython-input-9-92de12d6af9e>:1	2016/10/10 15:13:55	32 ms	1/1	2/2
5	runJob at PythonRDD.scala:356	2016/10/10 15:13:43	0,3 s	1/1	1/1
4	collect at <ipython-input-7-bb66d6ef0c7c>:1	2016/10/10 15:13:28	67 ms	1/1	2/2
3	collect at <ipython-input-5-bb66d6ef0c7c>:1	2016/10/10 15:13:01	46 ms	1/1	2/2
2	runJob at PythonRDD.scala:356	2016/10/10 11:52:10	0,2 s	1/1	1/1
1	count at <ipython-input-3-2d5aae3341c1>:1	2016/10/10 11:51:57	0,3 s	1/1	2/2
0	runJob at PythonRDD.scala:356	2016/10/10 11:51:31	1,0 s	1/1	1/1

Spark Shell

- Spark shell allows interaction (ad hoc analysis) with data that is distributed on disk or memory across many machines. Spark takes care of automatically distributing this processing.



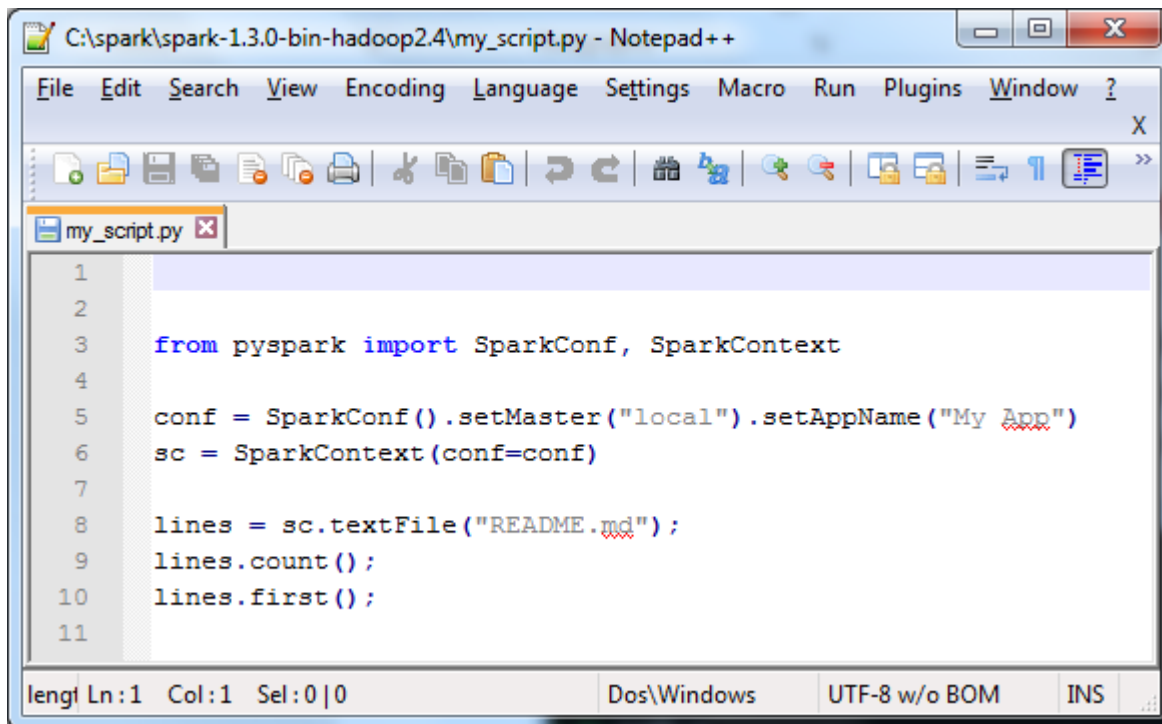
```
Invite de commandes - bin\pyspark
version 1.3.0

Using Python version 2.7.12 (default, Jun 29 2016 11:07:13)
SparkContext available as sc, HiveContext available as sqlCtx.

In [1]:
In [1]:
In [1]: textRDD = sc.textFile("README.md")
In [2]: textRDD.first()
Out[2]: u'# Apache Spark'
In [3]: textRDD.count
textRDD.countApprox      textRDD.countApproxDistinct  textRDD.countByValue
textRDD.countApprox
In [3]: textRDD.count()
Out[3]: 98
In [4]: textRDD.take(10)
Out[4]:
[u'# Apache Spark',
 u'',
 u'Spark is a fast and general cluster computing system for Big Data. It provides',
 u'high-level APIs in Scala, Java, and Python, and an optimized engine that',
 u'supports general computation graphs for data analysis. It also supports a',
 u'rich set of higher-level tools including Spark SQL for SQL and structured',
 u'data processing, MLlib for machine learning, GraphX for graph processing.',
 u'and Spark Streaming for stream processing.',
 u'',
 u'<http://spark.apache.org/>' ]
```

Spark Batch Processing

- Spark also allows batch processing. The user can define a series of jobs in a program to be executed without manual intervention



The screenshot shows a Notepad++ window titled "C:\spark\spark-1.3.0-bin-hadoop2.4\my_script.py - Notepad++". The menu bar includes File, Edit, Search, View, Encoding, Language, Settings, Macro, Run, Plugins, Window, and Help. The toolbar contains various icons for file operations and editing. The active tab is "my_script.py". The code in the editor is as follows:

```
1
2
3  from pyspark import SparkConf, SparkContext
4
5  conf = SparkConf().setMaster("local").setAppName("My App")
6  sc = SparkContext(conf=conf)
7
8  lines = sc.textFile("README.md");
9  lines.count();
10 lines.first();
11
```

The status bar at the bottom shows "lengt Ln:1 Col:1 Sel:0 | 0", "Dos\Windows", "UTF-8 w/o BOM", and "INS".

Resilient Distributed Dataset

Dataset

- Data storage created from: HDFS, S3, HBase, JSON, text, local hierarchy of folders
- Or created transforming another RDD

Resilient Distributed Dataset

Distributed

- Distributed across the cluster of machines
- Divided in **partitions**, chunks of data
- The partition is the unit of execution

Resilient Distributed Dataset

Resilient

- Recover from errors, e.g. node failure, slow
- Track history of each partition, re-run

Outline

- ❑ Introduction
- ❑ Programming with RDDs
- ❑ Working with pair RDDs
- ❑ Spark With Hadoop

Programming with RDDs

- Spark allows In-Memory processing
- An RDD is an immutable distributed collection of objects
- RDDs can contain any type of Python, Scala and JAVA objects including user defined classes
- Create an RDD:

```
text_RDD=sc.textFile("fileName")
```

```
Nums_RDD=sc.parallelize([1, 2, 3])
```

```
Nums_RDD= sc.parallelize(range(10))
```

Programming with RDDs

Can we do every thing in
Memory?

Programming with RDDs

- Can we do everything in Memory?
 - Absolutely NO
 - We will never have enough memory to fit all data
- Spark computes RDDs in a **LAZY FASHION**
- Spark RDDs offer two types of operations
 - **Transformations**
 - **Actions**

Transformations

- Remember that RDDs are immutable (Never modify RDD in place)
- A transformation creates an RDD from a previous one:
- Example:
 - map : apply function to each element of RDD

```
def lower(line):  
    return line.lower()  
lower_text_RDD = text_RDD.map(lower)
```

Actions

- An action is the final stage of workflow
- It triggers the execution of different transformations
- Returns results to the driver or writes to an external storage (e.g. HDFS)
- Example:

```
lower_text_RDD.collect()  
text_RDD.first()
```

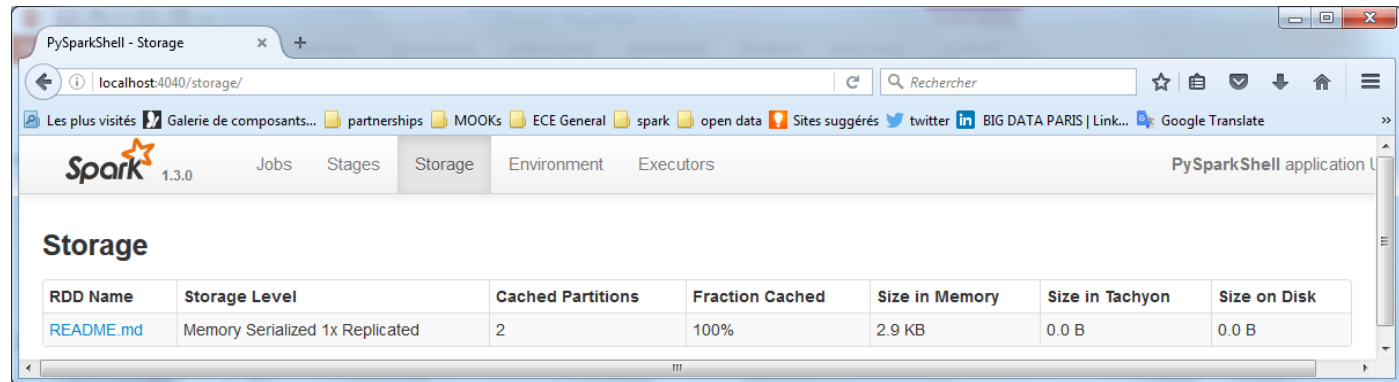
Persist

- Spark RDDs are by default recomputed each time you run an action on them
- If you like to use an RDD in multiple actions, you can ask Spark to persist it
- Example

```
rdd.persist()  
rdd.cache()
```

Persist

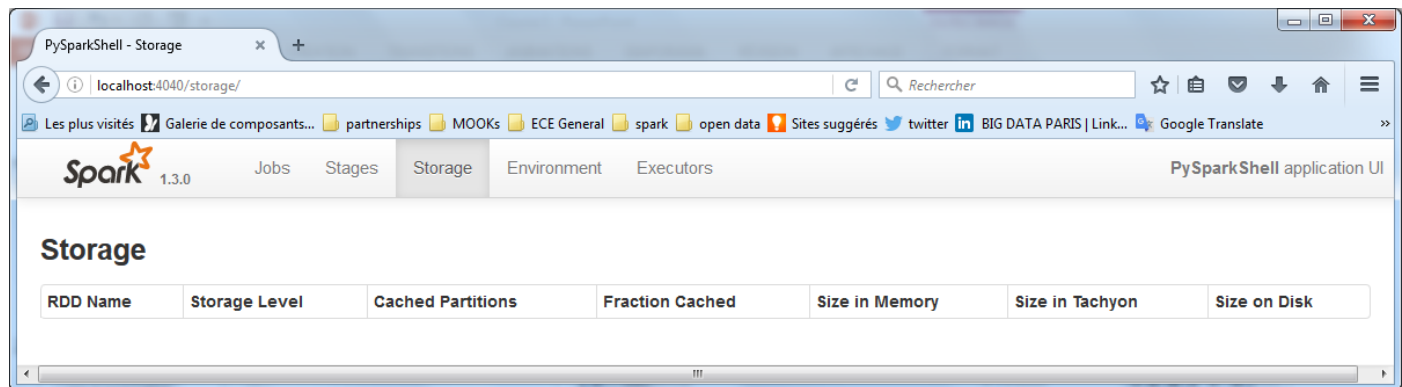
```
textRDD = sc.textFile('README.md')  
textRDD.persist()  
textRDD.first()
```



The screenshot shows the PySparkShell application UI with the 'Storage' tab selected. A table lists the RDD 'README.md' with the following details:

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
README.md	Memory Serialized 1x Replicated	2	100%	2.9 KB	0.0 B	0.0 B

```
textRDD.unpersist()
```



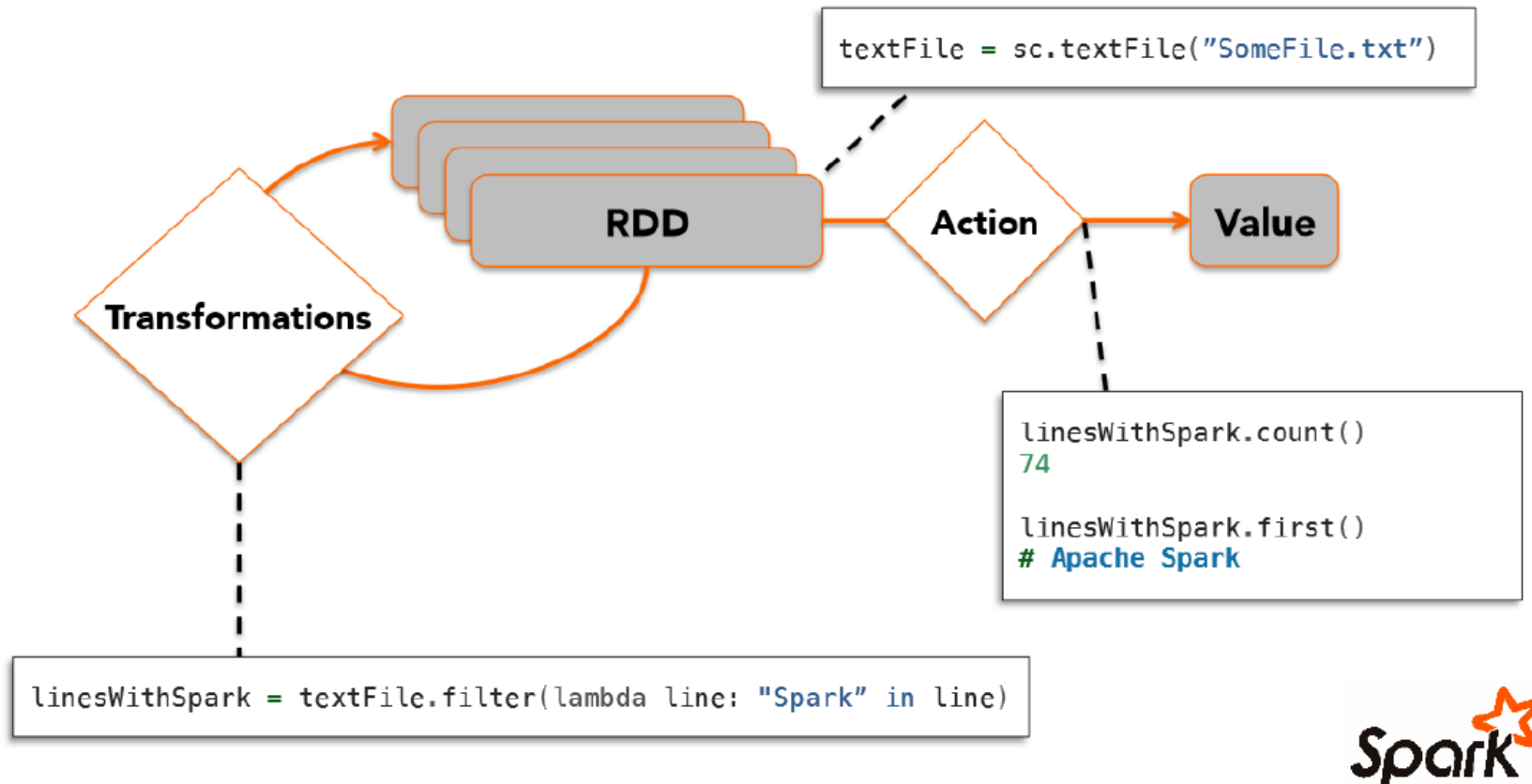
The screenshot shows the PySparkShell application UI with the 'Storage' tab selected. The table is empty, indicating that the RDD 'README.md' has been unpersisted and is no longer in the storage list.

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
----------	---------------	-------------------	-----------------	----------------	-----------------	--------------

Spark Program Work Flow

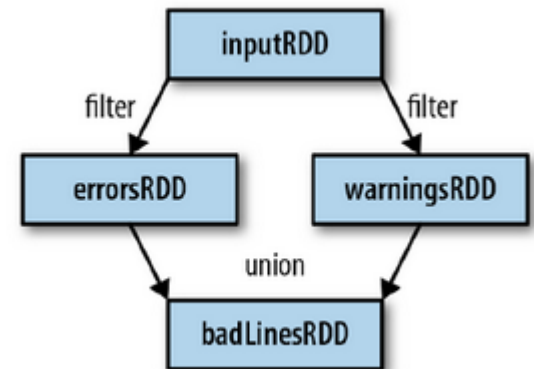
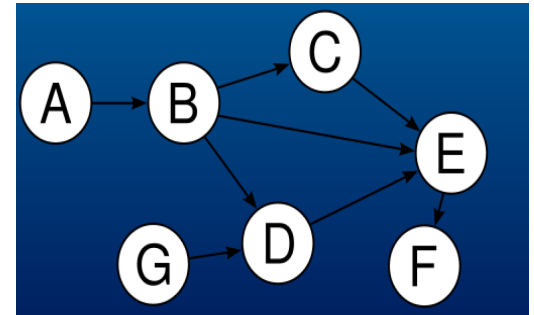
- Create input RDD from external data
Ex: `listRDD = sc.parallelize(1,2,3,4, 2)`
- Transform them to define new RDDs
(`listRDD.map(func)`)
- Ask Spark to `persist()` any intermediate RDD
- Launch actions like `first()` and `collect()` to kickoff a parallel computation which is then optimized and executed by Spark

Spark Program Work Flow



Directed Acyclic Graphs (Lineage Graphs)

- Spark keeps track of the set of dependencies between the different RDDs
- It uses this information to compute each RDD and to recover lost data
- Action forces the evaluation of transformations required for the RDD they were called on



RDD Transformations

Function Name	Purpose
map (<i>func</i>)	Return a new distributed dataset formed by passing each element of the source through a function <i>func</i> .
filter (<i>func</i>)	Return a new dataset formed by selecting those elements of the source on which <i>func</i> returns true.
flatMap (<i>func</i>)	Similar to map, but each input item can be mapped to 0 or more output items (so <i>func</i> should return a Seq rather than a single item).
sample (<i>withReplacement</i> , <i>fraction</i> , <i>seed</i>)	Sample a fraction <i>fraction</i> of the data, with or without replacement, using a given random number generator seed.
distinct ()	Removes duplicates

RDD Transformations

Function Name	Purpose	Example	Result
union (<i>otherDataset</i>)	Return a new dataset that contains the union of the elements in the source dataset and the argument.	rdd.union (other)	{1, 2, 3, 3, 4, 5}
intersection (<i>other</i>)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.	rdd.intersection (other)	{3}
subtract (<i>other</i>)	Remove the contents of one RDD (e.g., remove training data)	rdd.subtract (other)	{1, 2}
cartesian (<i>other</i>)	Cartesian product with other RDD	rdd.cartesian (other)	{(1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 3), (3, 4), (3, 5)}

rdd={1, 2, 3}, other={3, 4, 5}



ÉCOLE
POLYTECHNIQUE
UNIVERSITÉ PARIS-SACLAY

ECE PARIS
ÉCOLE D'INGÉNIEURS

RDD Actions

Function Name	Purpose
reduce (<i>func</i>)	Aggregate the elements of the dataset using a function <i>func</i> (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect ()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count ()	Return the number of elements in the dataset.
first ()	Return the first element of the dataset (similar to take(1)).
take (<i>n</i>)	Return an array with the first <i>n</i> elements of the dataset.
takeSample (<i>withReplacement</i> , <i>num</i> , [<i>seed</i>])	Return an array with a random sample of <i>num</i> elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.

RDD Actions

Function Name	Purpose
top (num)	Return the num top elements
fold (zero)(func)	Same as reduce, but with provided zero value
aggregate (zeroValue, seqOp, combOp)	Similar to reduce, but used to return a different type
takeOrdered (<i>n</i> , [ordering])	Return the first <i>n</i> elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile (<i>path</i>)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.

Example 1: Aggregate()

Calculate the sum and the number of elements in an RDD

In Python:

```
listRDD = sc.parallelize([1,2,3,4], 2) // 2 partitions
seqOp = (lambda local_result, list_element: (local_result[0] + list_element,
local_result[1] + 1) )
combOp = (lambda some_local_result, another_local_result:
(some_local_result[0] + another_local_result[0], some_local_result[1] +
another_local_result[1]))
listRDD.aggregate( (0, 0), seqOp, combOp)
```

Exercise: Aggregate()

Write a Python program that calculates:

1. the sum of elements of an RDD
2. the squares' sum
3. the number of elements in an RDD

Exercise Aggregate()

Write a Python program that calculates:

1. the sum of elements of an RDD
2. the squares' sum
3. the number of elements in an RDD

```
listRDD = sc.parallelize([1,2,3,4, 5, 6])
seqOp = lambda local_result, list_element: (
    local_result[0] + list_element,
    local_result[1] + 1,
    local_result[2] + list_element**2)
combOp = lambda some_local_result, another_local_result: (
    some_local_result[0] + another_local_result[0],
    some_local_result[1] + another_local_result[1],
    some_local_result[2] + another_local_result[2])
listRDD.aggregate( (0, 0, 0), seqOp, combOp)
```

Exercise Aggregate()

Write a Scala program that calculates:

1. the sum of elements of an RDD
2. the squares' sum
3. the number of elements in an RDD

```
val listRDD = sc.parallelize(list(1,2,3,4, 5, 6))  
val r = listRDD.aggregate((0, 0, 0))  
    ((x, y) => (x._1 + y, x._2 + 1, x._3 + y**2),  
    (x, y) => (x._1 + y._1, x._2 + y._2, x._3 + y._3))
```


Lazy Fashion: Does it make sense?

- Spark computes RDDs in a lazy fashion
- This has a lot of sense as only the needed memory is used
- For instance, with `textFile("filename")`, if Spark was to load and store in memory all the lines in the file, this would waste a lot of memory
- Then based on the action, Spark loads only the necessary data into memory
- With `first()`, Spark only scans the first line
- Lazy evaluation allows Spark to reduce the number of passes it has to take over data!
- Developers can develop simple programs (They don't worry about writing complex programs to reduce passes)

Action and Transformation

How to distinguish?

- Transformations are operations on RDDs that return new RDDs
- Actions are operations that return something (a result) to the driver or write it to a storage !

Exercise:

Lower and Upper Case Count

Write a Python program for spark that displays how many words in a script start by a (lower and upper case) and by b

RDD Persistence

Each node stores any partitions of the cache that it computes in memory
Reuses them in other actions on that dataset or (datasets derived from it)

Storage Level	Meaning
MEMORY_ONLY	Store as deserialized Java objects in the JVM. If the RDD does not fit in memory, part of it will be cached. The other will be recomputed as needed. This is the default. The <code>cache()</code> method uses this.
MEMORY_AND_DISK	Same except also store on disk if it doesn't fit in memory. Read from memory and disk when needed.
MEMORY_ONLY_SER	Store as serialized Java objects (one byte array per partition). Space efficient, but more CPU intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_AND_DISK but stored as serialized objects.
DISK_ONLY	Store only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Store RDD in serialized format in Tachyon.

e.g.: `rdd.persist(StorageLevel.MEMORY_AND_DISK)`

Outline

- ❑ Introduction
- ❑ Programming with RDDs
- ❑ Working with pair RDDs
- ❑ Spark With Hadoop

Creating Pair RDDs

- Spark allows working using [key, values] pairs, which is very useful in many applications.
- Pair RDDs can be created with `parallelize()`:
`data = [('a', 3), ('b', 4), ('a', 1)]`
`rdd = sc.parallelize(data)`
- They can also be created depending on the application:

Python example:

```
pairs = lines.map(lambda x: (x.split(" ")[0], x))
```

Scala example:

```
val pairs = lines.map(x => (x.split(" ")(0), x))
```

Programming Pair RDDs

- Like 'normal' RDDs, Spark provides 2 main operations to process pair RDDs:
 - Transformations return RDDs
 - Actions return result
- Spark provides a set of powerful transformations and actions to process pair RDDs based on keys

Transformations on one pair RDD

RDD example: $\{(1, 2), (3, 4), (3, 6)\}$

Function name	Purpose	Example	Result
<code>reduceByKey(func)</code>	Combine values with the same key.	<code>rdd.reduceByKey(lambda (x, y): x + y)</code>	$\{(1, 2), (3, 10)\}$
<code>groupByKey()</code>	Group values with the same key.	<code>rdd.groupByKey()</code>	$\{(1, [2]), (3, [4, 6])\}$
<code>combineByKey(createCombiner, mergeValue, mergeCombiners, partitioner)</code>	Combine values with the same key using a different result type.	See example later	

Transformations on one pair RDD

RDD example: {(1, 2), (3, 4), (3, 6)}

Function name	Purpose	Example	Result
mapValues(func)	Apply a function to each value of a pair RDD without changing the key.	rdd.mapValues(lambda x: x+1)	{(1, 3), (3, 5), (3, 7)}
flatMapValues(func)	Apply a function that returns an iterator to each value of a pair RDD, and for each element returned, produce a key/value entry with the old key. Often used for tokenization.	rdd.flatMapValues(lambda x: range (x, 6)	{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)}

Transformations on one pair RDD

RDD example: {(1, 2), (3, 4), (3, 6)}

Function name	Purpose	Example	Result
keys()	Return an RDD of just the keys.	rdd.keys()	{1, 3, 3}
values()	Return an RDD of just the values.	rdd.values()	{2, 4, 6}
sortByKey()	Return an RDD sorted by the key.	rdd.sortByKey()	{(1, 2), (3, 4), (3, 6)}

Example: Word count in Python

```
x = sc.textFile("README.md")
y = x.flatMap(lambda x: x.split(" ") )
z = y.map(lambda x: (x, 1))
t = z.reduceByKey(lambda x, y: x+y)
```

Transformations on two pair RDDs

rdd= {(1, 2), (3, 4), (3, 6)} other = {(3, 9)}

Function name	Purpose	Example	Result
subtractByKey	Remove elements with a key present in the other RDD.	rdd.subtractByKey(other)	{(1, 2)}
join	Perform an inner join between two RDDs.	rdd.join(other)	{(3, (4, 9)), (3, (6, 9))}
rightOuterJoin	Perform a join between two RDDs where the key must be present in the first RDD.	rdd.rightOuterJoin(other)	{(3,(Some(4),9)), (3,(Some(6),9))}
leftOuterJoin	Perform a join between two RDDs where the key must be present in the other RDD.	rdd.leftOuterJoin(other)	{(1,(2,None)), (3,(4,Some(9))), (3,(6,Some(9)))}
cogroup	Group data from both RDDs sharing the same key.	rdd.cogroup(other)	{(1,([2],[])),

Example: Join

```
storeAddress = { (Store("Ritual"), "1026 Valencia St"),  
                  (Store("Philz"), "748 Van Ness Ave"),  
                  (Store("Philz"), "3101 24th St"),  
                  (Store("Starbucks"), "Seattle") }
```

```
storeRating = { (Store("Ritual"), 4.9),  
                (Store("Philz"), 4.8) }
```

```
storeAddress.join(storeRating)
```

```
{ (Store("Ritual"), ("1026 Valencia St", 4.9)),  
  (Store("Philz"), ("748 Van Ness Ave", 4.8)),  
  (Store("Philz"), ("3101 24th St", 4.8)) }
```

Example: Left Outer Joins

```
storeAddress = { (Store("Ritual"), "1026 Valencia St"),  
                  (Store("Philz"), "748 Van Ness Ave"),  
                  (Store("Philz"), "3101 24th St"),  
                  (Store("Starbucks"), "Seattle") }
```

```
storeRating = { (Store("Ritual"), 4.9),  
                 (Store("Philz"), 4.8) }
```

```
storeAddress.leftOuterJoin(storeRating)
```

```
{(Store("Ritual"),("1026 Valencia St",Some(4.9))),  
 (Store("Starbucks"),("Seattle",None)),  
 (Store("Philz"),("748 Van Ness Ave",Some(4.8))),  
 (Store("Philz"),("3101 24th St",Some(4.8)))}
```

Example: Right Outer Joins

```
storeAddress = { (Store("Ritual"), "1026 Valencia St"),  
                  (Store("Philz"), "748 Van Ness Ave"),  
                  (Store("Philz"), "3101 24th St"),  
                  (Store("Starbucks"), "Seattle") }
```

```
storeRating = { (Store("Ritual"), 4.9),  
                 (Store("Philz"), 4.8) }
```

```
storeAddress.rightOuterJoin(storeRating)
```

```
{(Store("Ritual"),(Some("1026 Valencia St"),4.9)),  
 (Store("Philz"),(Some("748 Van Ness Ave"),4.8)),  
 (Store("Philz"), (Some("3101 24th St"),4.8)) }
```

Example: Sorting Data

In Python:

```
rdd.sortByKey(ascending=True, numPartitions=None,  
              keyfunc = lambda x: str(x))
```

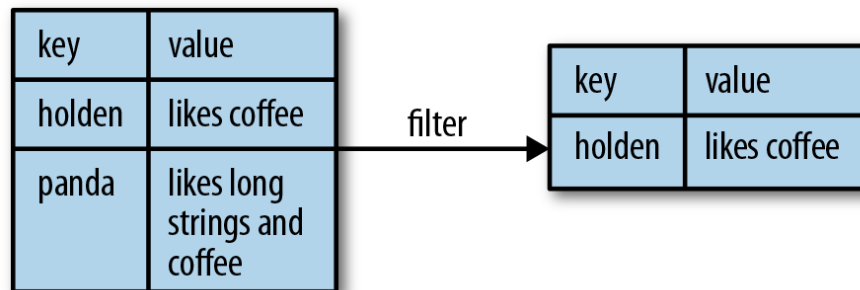

Simple Filter on Second Element

Python:

```
result = pairs.filter(lambda keyValue: len(keyValue[1]) < 20)
```

Scala:

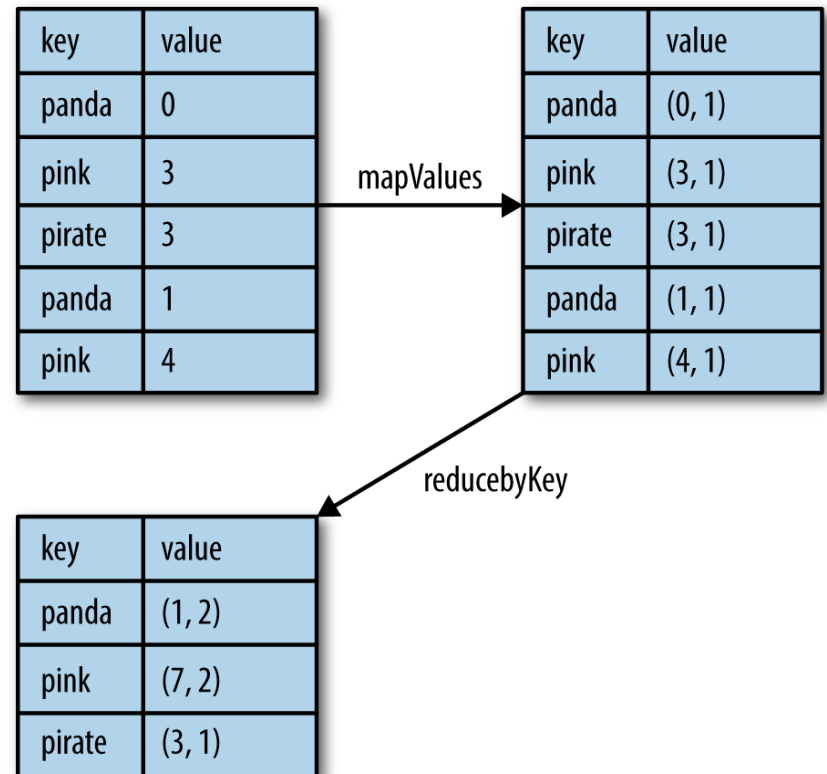
```
pairs.filter{case (key, value) => value.length < 20}
```



Exercise

Write a Python program that returns per-key average with `reduceByKey()`!

And then with `combineByKey()`



Per-key average with combineByKey()

Partition 1

coffee	1
coffee	2
panda	3

Partition 1 trace:

(coffee, 1) -> new key

accumulators[coffee] = createCombiner(1)

(coffee, 2) -> existing key

accumulators[coffee] = merge Value(accumulators[coffee], 2)

(panda, 3) -> new key

accumulators[panda] = createCombiner(3)

Partition 2

coffee	9
--------	---

Partition 2 trace:

(coffee, 9) -> new key

accumulators[coffee] = createCombiner(9)

Merge Partitions:

mergeCombiners(partition1.accumulators[coffee],
partition2.accumulators[coffee])

```
def createCombiner(value):  
  (value, 1)
```

```
def mergeValue(acc, value):  
  (acc[0] + value, acc[1] + 1)
```

```
def mergeCombiners(acc1, acc2):  
  (acc1[0] + acc2[0], acc1[1] + acc2[1])
```

Actions on Pair RDDs

Function	Description	Example	Result
countByKey()	Count the number of elements for each key.	rdd.countByKey()	{{(1, 1), (3, 2)}}
collectAsMap()	Collect the result as a map to provide easy lookup.	rdd.collectAsMap()	Map{(1, 2), (3, 4), (3, 6)}
lookup(key)	Return all values associated with the provided key.	rdd.lookup(3)	[4, 6]

Outline

- ❑ Introduction
- ❑ Programming with RDDs
- ❑ Working with pair RDDs
- ❑ Spark With Hadoop

Hadoop Cluster

- HDFS is Hadoop's storage subsystem
- YARN: Hadoop's processing subsystem (or resource scheduling system)
- Hadoop Cluster contains 2 clusters:
 - HDFS Cluster
 - YARN Cluster

HDFS Cluster

- HDFS:
 - Virtual file system (files are composed of blocks distributed over the cluster)
 - Master Slave architecture:
 - NameNode: provides clients with block locations for read/write operations. It maintains the metadata (treasure map) in memory
 - DataNode: manages blocks of data (storage and access) for reading and writing data as well as block replication.

YARN Cluster

- YARN: Yet Another Resource Negotiator
 - It governs and orchestrates the processing of data in Hadoop
 - Master Slave architecture:
 - One master daemon: ResourceManager
 - Several slave daemons: NodeManagers
- **ResourceManager**: responsible for granting cluster compute resources to applications.
 - Resources are granted in units called containers.
 - Containers are predefined combinations of CPU cores and memory.
 - It provides a web UI on port **8088** on the node running this daemon (basically for displaying status of running applications such as Spark's).
- **DataNode** manages containers on the slave node host.
 - Containers execute the application's tasks
 - The first container allocated by the resource manager is the ApplicationMaster, it plans and manage the application, determine necessary ressources for each stage, etc..

SPARK On Hadoop With YARN

- A Spark client submits an application to the ResourceManager (RM)
- The RM allocates an ApplicationMaster (AM) process (on a NodeManager NM)
- The AM negotiates task containers with RM and dispatches processing to the NodeManagers (NM) hosting this containers
- NMs report task status and progress the AM
- AM reports progress and the status of the application to the RM
- The RM reports progress and results to the client

YARN's Resource Manager Web UI

The screenshot displays the YARN Resource Manager Web UI in a Mozilla Firefox browser. The page title is "All Applications - Mozilla Firefox". The browser address bar shows "vm4:8088/cluster". The page features the Hadoop logo and a sidebar with navigation links: Cluster, About, Nodes, Applications, NEW, NEW SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED, KILLED, Scheduler, and Tools.

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total
5	0	0	5	0	0 B	5.98 GB	0 B	0	8

Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes
4	0	0	0	0

User Metrics for dr.who

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Containers Pending	Containers Reserved	Memory Used	Memory Pending	Memory Reserved	VCores Used	VCores Pending	VCores Reserved
0	0	0	0	0	0	0	0 B	0 B	0 B	0	0	0

Show 20 entries

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCoers	Allocated Memory MB	Reserved CPU VCoers	Reserved Memory MB
application_1537913080474_0005	training	Spark shell	SPARK	root.users.training	Fri Sep 28 16:17:49 +0200 2018	Fri Sep 28 18:31:31 +0200 2018	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A
application_1537913080474_0004	training	Spark shell	SPARK	root.users.training	Fri Sep 28 16:16:28 +0200 2018	Fri Sep 28 18:30:11 +0200 2018	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A
application_1537913080474_0003	training	Spark shell	SPARK	root.users.training	Fri Sep 28 16:13:08 +0200 2018	Fri Sep 28 16:14:28 +0200 2018	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A

Spark applications running on YARN can be accessed from YARN's ResourceManger Web UI

Spark Applications in Spark's History Server Web UI

centos7-gnome.pue.es:1 (training) x +


← → ↻ Effectuez une recherche sur Google ou saisissez une URL

Applications Places Firefox Mon 21:51

History Server - Mozilla Firefox

YARN (MR2 Included) - Cl X All Applications x History Server x All Applications x +

vm5:18088

 **History Server**
1.6.0

Event log directory: hdfs://vm2:8020/user/spark/applicationHistory

Showing 1-7 of 7 1

App ID	App Name	Started	Completed	Duration	Spark User	Last Updated
application_1537913080474_0005	Spark shell	2018/09/28 16:17:44	2018/09/28 18:31:30	2.2 h	training	2018/09/28 18:31:31
application_1537913080474_0004	Spark shell	2018/09/28 16:16:24	2018/09/28 18:30:11	2.2 h	training	2018/09/28 18:30:11
application_1537913080474_0003	Spark shell	2018/09/28 16:13:04	2018/09/28 16:14:28	1.4 min	training	2018/09/28 16:14:28
application_1537913080474_0002	Spark shell	2018/09/28 15:19:31	2018/09/28 16:11:09	52 min	training	2018/09/28 16:11:09
application_1537913080474_0001	Spark shell	2018/09/28 09:55:17	2018/09/28 14:35:55	4.7 h	training	2018/09/28 14:35:55
application_1537791984851_0002	Spark shell	2018/09/24 22:25:21	2018/09/24 23:41:41	1.3 h	training	2018/09/24 23:41:41
application_1537791984851_0001	Spark shell	2018/09/24 14:29:37	2018/09/24 14:43:17	14 min	training	2018/09/24 14:43:17

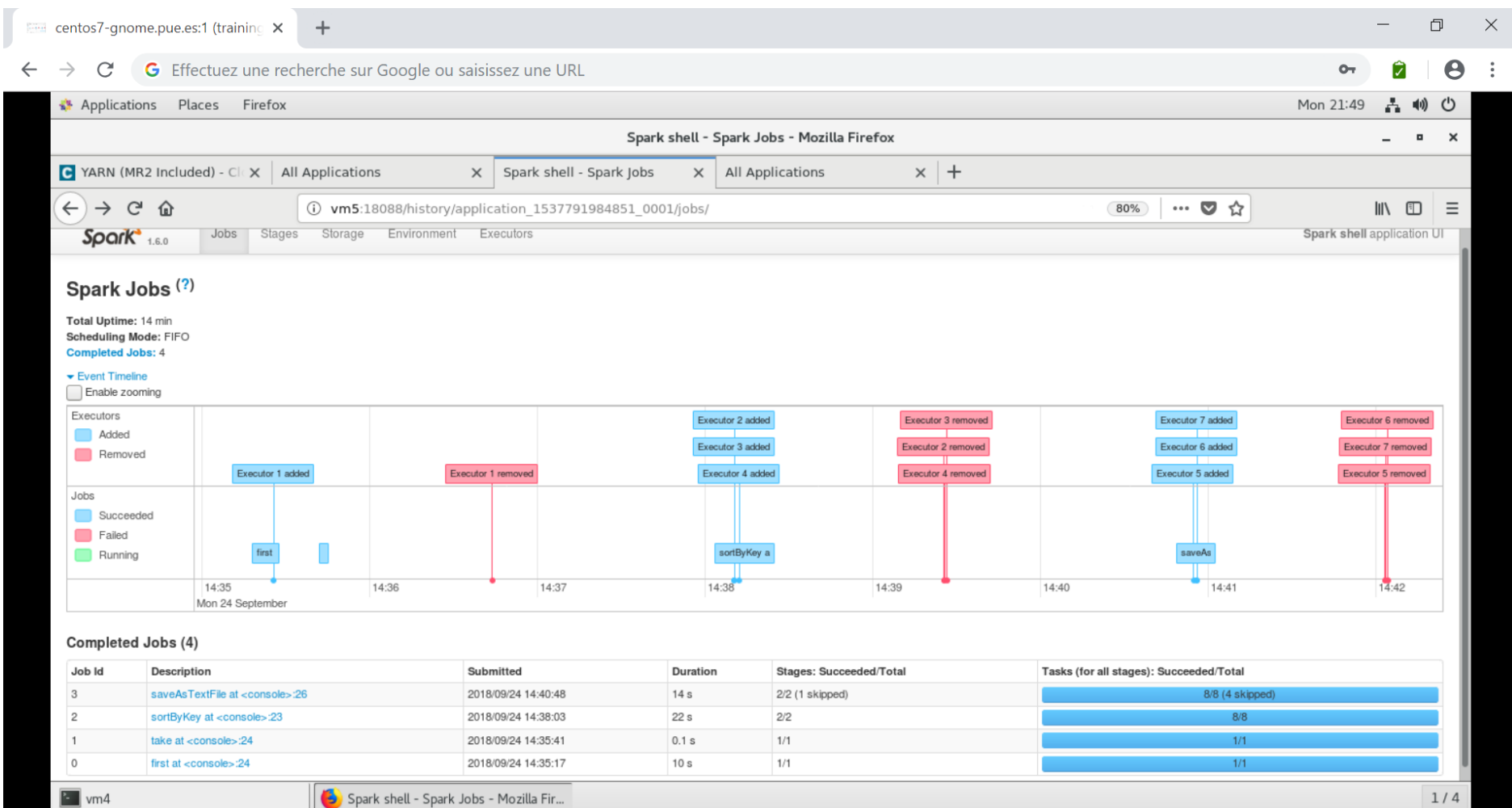
[Show incomplete applications](#)

vm4 History Server - Mozilla Firefox 1 / 4

Spark Jobs

- A Spark **job** is split into a set of **stages** by the **DAGScheduler**
- **Jobs** are Spark actions (collect(), first(), etc..)
- Each **stage** contains a set of **transformations** that can be executed without shuffling the entire data set (for instance map() and filter())
- A **stage** is set of parallel **tasks**, one task for each **partition**.

Spark Jobs in Spark's Web UI



Stages of Spark Jobs

centos7-gnome.pue.es:1 (training) x +

← → ↻ Effectuez une recherche sur Google ou saisissez une URL

Applications Places Firefox Mon 21:47

Spark shell - Details for Job 2 - Mozilla Firefox

YARN (MR2 Included) - x All Applications x Spark shell - Details for Job 2 x All Applications x +

vm5:18088/history/application_1537791984851_0001/jobs/job?id=2 90% ... ☆

Event Timeline
▼ DAG Visualization

Stage 2

Stage 3

Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3	sortByKey at <console>:23 +details	2018/09/24 14:38:23	2 s	4/4			16.3 MB	
2	map at <console>:23 +details	2018/09/24 14:38:03	13 s	4/4	518.2 MB			9.9 MB

vm4 Spark shell - Details for Job 2 - Moz... 1 / 4

Tasks of a Spark's Job Stage

centos7-gnome.pue.es:1 (training) x +

← → ↻ 🔍 | 🔑 🟢 👤 ⋮

Applications Places Firefox Tue 01:11 🖨 🔊 🔌

Spark shell - Details for Stage 3 (Attempt 0) - Mozilla Firefox

File Edit View History Bookmarks Tools Help

Spark - Cloudera Manage X Spark shell - Details for Stag X +

← → ↻ 🏠 🔍 vm5:18088/history/application_1537791984851_0001/stages/stage/?id=3&attempt=0 110% ⋮ 📄 ⋮

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.4 s	1 s	1 s	1 s	1 s
GC Time	0 ms	0 ms	9 ms	33 ms	33 ms
Shuffle Read Size / Records	3.9 MB / 212001	3.9 MB / 212293	4.2 MB / 212491	4.4 MB / 213515	4.4 MB / 213515

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Read Size / Records	Blacklisted
2	CANNOT FIND ADDRESS	2 s	1	0	1	4.2 MB / 212293	0
3	CANNOT FIND ADDRESS	2 s	2	0	2	7.7 MB / 424492	0
4	CANNOT FIND ADDRESS	1 s	1	0	1	4.4 MB / 213515	0

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Shuffle Read Size / Records	Errors
0	6	0	SUCCESS	NODE_LOCAL	2 / vm2	2018/09/24 14:38:23	1 s		4.2 MB / 212293	
1	7	0	SUCCESS	NODE_LOCAL	3 / vm3	2018/09/24 14:38:23	1 s		3.9 MB / 212001	
2	8	0	SUCCESS	NODE_LOCAL	4 / vm5	2018/09/24 14:38:23	1 s	9 ms	4.4 MB / 213515	
3	9	0	SUCCESS	NODE_LOCAL	3 / vm3	2018/09/24 14:38:24	0.4 s	33 ms	3.9 MB / 212491	

vm4 Spark shell - Details for Stage 3 (At... 1 / 4

Installation & Environments

Installation and Environments

Local (Windows and UBUNTU)

- Document sent by mail
- <https://medium.com/@GalarnykMichael/install-spark-on-windows-pyspark-4498a5d8d66c>

Databricks

<https://community.cloud.databricks.com>

Microsoft Azure:

<https://portal.azure.com/>

References

- [1] **Learning Spark: Lightning-Fast Data Analysis** by
Holden Karau, Andy Konwinski, Patrick Wendell & Matei Zaharia
Publisher: O'REILLY

- [2] **High Performance Spark: Best Practices For Scaling & Optimizing Apache Spark**
Holden Karau & Rachel Warren
Publisher: O'REILLY

- [3] **Advanced Analytics With Spark: Patterns For Learning From Data At Scale**
Sandy Ryza, Uri Laserson, Sean Owen & Josh Wills
Publisher: O'REILLY

- [4] **Data Analysis with SPARK Using PYTHON**
Jeffrey Aven

References

Spark Official Documentation:

- [1] <http://spark.apache.org>
- [2] <http://spark.apache.org/docs/latest/quick-start.html>
- [3] <http://spark.apache.org/docs/latest/rdd-programming-guide.html>
- [4] <https://spark.apache.org/docs/latest/api/python/index.html>

References

Recommended MOOCs

1. **Hadoop Platform and Applications Framework (Coursera, UC San Diego):**
 - <https://www.coursera.org/learn/hadoop/home/welcome>
2. **Introduction to Apache Spark (EDX, Berkeley):**
 - <https://courses.edx.org/courses/course-v1:BerkeleyX+CS105x+1T2016/info>
3. **Distributed Machine Learning with Apache Spark (EDX, Berkeley):**
 - <https://courses.edx.org/courses/course-v1:BerkeleyX+CS120x+2T2016/info>
4. **Big Data Analysis with Apache Spark (EDX, Berkeley):**
 - <https://courses.edx.org/courses/course-v1:BerkeleyX+CS110x+2T2016/info>
5. **Machine Learning (Coursera, Stanford)**
 - <https://www.coursera.org/learn/machine-learning/home/welcome>

References

Start with Python

1. Python Basics

- <http://ai.berkeley.edu/tutorial.html#PythonBasics>

2. Python Strings

- https://www.tutorialspoint.com/python/python_strings.htm

3. Python Standard Library:

- <https://docs.python.org/2/library/index.html>

4. Python Tutorial:

- <https://docs.python.org/3/tutorial/index.html>