CentraleSupélec

Big Data : Informatique pour les données et calculs massifs

7 – SPARK technology

Stéphane Vialle

Stephane.Vialle@centralesupelec.fr
http://www.metz.supelec.fr/~vialle

CentraleSupélec

Spark Technology

CentraleSupélec

1 - Spark main objectives

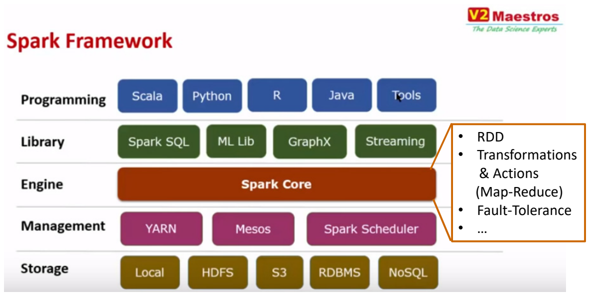
Spark has been designed:

- To efficiently run iterative and interactive applications
  → keeping data in-memory between operations
- To provide a low-cost fault tolerance mechanism
  → low overhead during safe executions
  → fast recovery after failure
- To be easy and fast to use in interactive environment
  → Using compact *Scala* programming language
- To be « scalable »
  → able to efficiently process bigger data on larger computing clusters

Spark is based on a distributed data storage abstraction:

- the « **RDD** » (*Resilient Distributed Datasets*)
- compatible with many distributed storage solutions

1 - Spark main objectives



Spark design started in 2009, with the PhD thesis of Matei Zaharia at Berkeley Univ.
Matei Zaharia co-founded DataBricks in 2013.

Spark Technology

1. Spark main objectives
2. **RDD concepts and operations**
3. SPARK application scheme and execution
4. Application execution on clusters and clouds
5. Basic programming examples
6. Basic examples on pair RDDs
7. PageRank with Spark

2 - RDD concepts and operations

A RDD (*Resilient Distributed Dataset*) is:

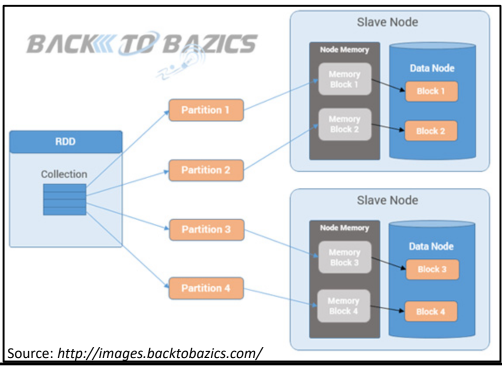
- an immutable (read only) dataset
- a partitioned dataset
- usually stored in a distributed file system (like HDFS)

When stored in HDFS:

- One RDD → One HDFS file
- One RDD partition block → One HDFS file block
- Each RDD partition block is replicated by HDFS

2 - RDD concepts and operations

Example of a 4 partition blocks stored on 2 data nodes (no replication)



Source: *http://images.backtobazics.com/*

2 - RDD concepts and operations

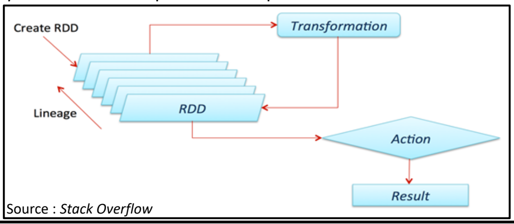
Initial input RDDs:

- are usually created from distributed files (like HDFS files),
- Spark processes read the file blocks that become in-memory RDD

Operations on RDDs:

- Transformations : read RDDs, compute, and generate a new RDD
- Actions : read RDDs and generate results out of the RDD world

Map and *Reduce* are parts of the operations



Source : *Stack Overflow*

2 - RDD concepts and operations

Exemple of Transformations and Actions

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

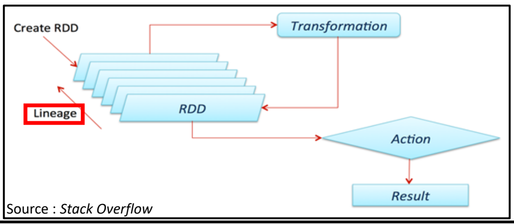
Source : *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*. **Matei Zaharia** *et al.* Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. San Jose, CA, USA, **2012**

2 - RDD concepts and operations

Fault tolerance:

- Transformation are coarse grained op: they apply on all data of the source RDD
- RDD are read-only, input RDD are not modified
- A sequence of transformations (a *lineage*) can be easily stored

→ In case of failure: Spark has *just* to re-apply the lineage of the missing RDD partition blocks.



Source : *Stack Overflow*

2 - RDD concepts and operations

5 main internal properties of a RDD:

- A list of partition blocks
  `getPartitions()`
- A function for computing each partition block
  `compute(…)`
- A list of dependencies on other RDDs: parent RDDs and transformations to apply
  `getDependencies()`

To compute and re-compute the RDD when failure happens

Optionally:

- A Partitioner for key-value RDDs: metadata specifying the RDD partitioning
  `partitioner()`

To control the RDD partitioning, to achieve co-partitioning…

- A list of nodes where each partition block can be accessed faster due to data locality
  `getPreferredLocations(…)`

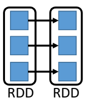
To improve data locality with HDFS & YARN…

2 - RDD concepts and operations

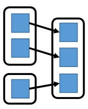
Narrow transformations

- Local computations applied to each partition block
  → no communication between processes/nodes
  → only local dependencies (between parent & son RDDs)

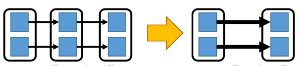
•Map()
•Filter()



•Union()



- In case of sequence of Narrow transformations:
  →possible pipelining inside one step

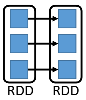


2 - RDD concepts and operations

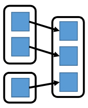
Narrow transformations

- Local computations applied to each partition block
  → no communication between processes/nodes
  → only local dependencies (between parent & son RDDs)

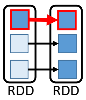
•Map()
•Filter()



•Union()



- In case of failure:
  → recompute only the damaged partition blocks
  → recompute/reload only its parent blocks

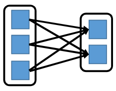


2 - RDD concepts and operations

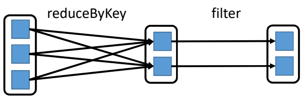
Wide transformations

- Computations requiring data from all parent RDD blocks
  → many communication between processes/nodes (*shuffle & sort*)
  → non-local dependencies (between parent & son RDDs)

•groupByKey()
•reduceByKey()



- In case of sequence of transformations:
  → no pipelining of transformations
  → wide transformation must be totally achieved before to enter next transformation

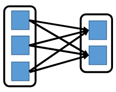


2 - RDD concepts and operations

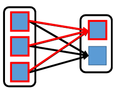
Wide transformations

- Computations requiring data from all parent RDD blocks
  → many communication between processes/nodes (*shuffle & sort*)
  → non-local dependencies (between parent & son RDDs)

•groupByKey()
•reduceByKey()



- In case of sequence of failure:
  → recompute the damaged partition blocks
  → recompute/reload all blocks of the parent RDDs

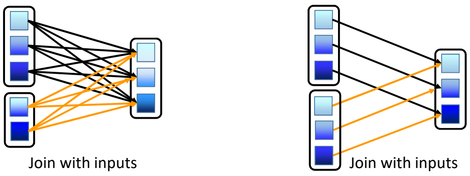


2 - RDD concepts and operations

Avoiding wide transformations with co-partitioning

- With identical partitioning of inputs:

  wide transformation → narrow transformation



Join with inputs **not** co-partitioned

Join with inputs **co-partitioned**

- less expensive communications
- possible pipelining
- less expensive fault tolerance

} Control RDD partitioning
Force co-partitioning
(using the same partition map)

2 - RDD concepts and operations

Persistence of the RDD

RDD are stored:

- in the memory space of the Spark Executors
- or on disk (of the node) when memory space of the Executor is full

By default: an old RDD is removed when memory space is required (*Least Recently Used* policy)

→ An old RDD has to be re-computed (using its *lineage*) when needed again

→ Spark allows to make a « persistent » RDD to avoid to recompute it

2 - RDD concepts and operations

Persistence of the RDD to improve Spark application performances

Spark application developper has to add instructions to force RDD storage, and to force RDD forgetting:

```
myRDD.persist(StorageLevel)     // or myRDD.cache()
… // Transformations and Actions
myRDD.unpersist()
```

Available *storage levels*:

- `MEMORY_ONLY` : in Spark Executor memory space
- `MEMORY_ONLY_SER` : + serializing the RDD data
- `MEMORY_AND_DISK` : on local disk when no memory space
- `MEMORY_AND_DISK_SER` : + serializing the RDD data in memory
- `DISK_ONLY` : always on disk (and serialized)

RDD is saved in the Spark executor memory/disk space
→ limited to the Spark session

2 - RDD concepts and operations

Persistence of the RDD to improve fault tolerance

To face *short term failures*: Spark application developper can force RDD storage with replication in the local memory/disk of several Spark Executors

```
myRDD.persist(storageLevel.MEMORY_AND_DISK_SER_2)
… // Transformations and Actions
myRDD.unpersist()
```

To face *serious failures*: Spark application developper can checkpoint the RDD outside of the Spark data space, on HDFS or S3 or…

```
myRDD.sparkContext.setCheckpointDir(directory)
myRDD.checkpoint()
… // Transformations and Actions
```

→ Longer, but secure!

Spark Technology

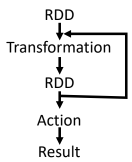
1. Spark main objectives
2. RDD concepts and operations
3. **SPARK application scheme and execution**
4. Application execution on clusters and clouds
5. Basic programming examples
6. Basic examples on pair RDDs
7. PageRank with Spark

3 – SPARK application scheme and execution

Transformations are **lazy** operations: saved and executed further

Actions **trigger** the execution of the sequence of transformations

A *job* is a sequence of RDD transformations, ended by an action



A *Spark application* is a set of jobs to run sequentially or in parallel

3 – SPARK application scheme and execution

The *Spark application driver* controls the application run

- It creates the Spark context
- It analyses the Spark program

↓

- It creates a DAG of tasks for each job
- It optimizes the DAG
  - pipelining narrow transformations
  - identifying the tasks that can be run in parallel

↓

- It schedules the DAG of tasks on the available worker nodes (the *Spark Executors*) in order to maximize parallelism (and to reduce the execution time)

3 – SPARK application scheme and execution

The *Spark application driver* controls the application run

- It attempts to keep in-memory the intermediate RDDs
  → in order the input RDDs of a transformation are already in-memory (ready to be used)
- A RDD obtained at the end of a transformation can be explicitely kept in memory, when calling the persist() method of this RDD (interesting if it is re-used further).

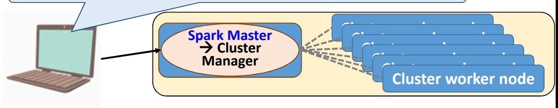
Spark Technology

1. Spark main objectives
2. RDD concepts and operations
3. SPARK application scheme and execution
4. **Application execution on clusters and clouds**
5. Basic programming examples
6. Basic examples on pair RDDs
7. PageRank with Spark

4 – Application execution on clusters and clouds

1 - with Spark Master as cluster manager (standalone mode)

```
spark-submit --master spark://node:port … myApp
```



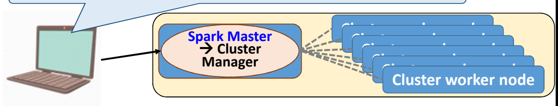
Spark cluster configuration:

- Add the list of cluster worker nodes in the Spark Master config.
- Specify the maximum amount of memory per Spark Executor
  `spark-submit --executor-memory XX …`
- Specify the total amount of CPU cores used to process one Spark application (through all its Spark executors)
  `spark-submit --total-executor-cores YY …`

4 – Application execution on clusters and clouds

1 - with Spark Master as cluster manager (standalone mode)


```
spark-submit --master spark://node:port … myApp
```



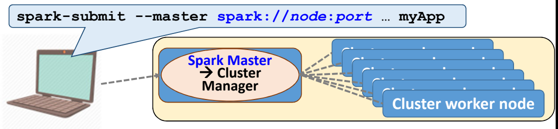
Spark cluster configuration:

- Default config :
  - (only) 1GB/Spark Executor
  - Unlimited nb of CPU cores per application execution
  - The Spark Master creates one mono-core Executor on all Worker nodes to process each job
- You can limit the total nb of cores per job
- You can concentrate the cores into few multi-core Executors

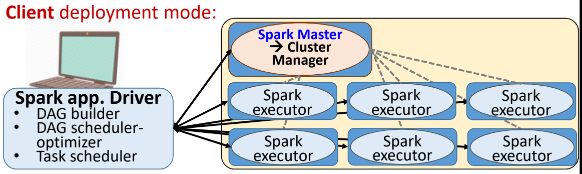
4 – Application execution on clusters and clouds

1 - with Spark Master as cluster manager (standalone mode)


```
spark-submit --master spark://node:port … myApp
```



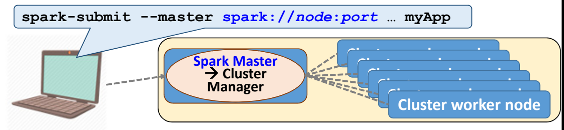
Client deployment mode:



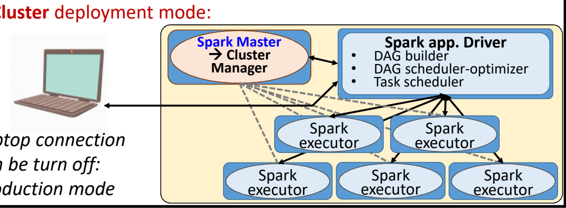
Interactive control of the application: development mode

4 – Application execution on clusters and clouds

1 - with Spark Master as cluster manager (standalone mode)



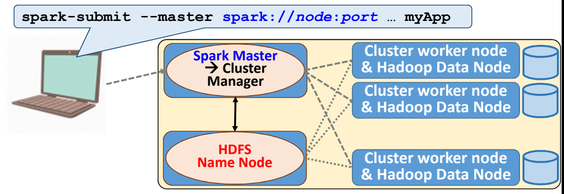
Cluster deployment mode:



Laptop connection can be turn off: production mode

4 – Application execution on clusters and clouds

1 - with Spark Master as cluster manager (standalone mode)



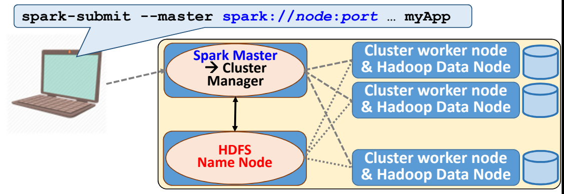
The Cluster Worker nodes should be the Data nodes, storing initial RDD values or new generated (and saved) RDD

→ Will improve the global data-computations locality

→ **When using HDFS**: the Hadoop data nodes should be re-used as worker nodes for Spark Executors

4 – Application execution on clusters and clouds

1 - with Spark Master as cluster manager (standalone mode)



The Cluster Worker nodes should be the Data nodes, storing initial RDD values or new generated (and saved) RDD

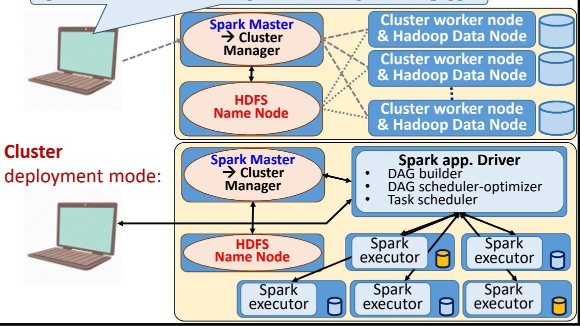
When using the Spark Master as Cluster Manager:

…there is no way to localize the Spark Executors on the data nodes hosting the right RDD blocks!

4 – Application execution on clusters and clouds

1 - with Spark Master as cluster manager (standalone mode)

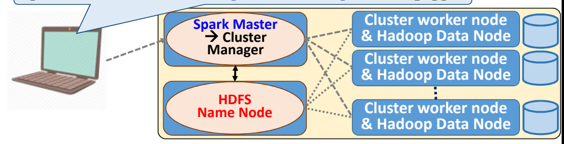
`spark-submit --master spark://node:port … myApp`



4 – Application execution on clusters and clouds

1 - with Spark Master as cluster manager (standalone mode)

`spark-submit --master spark://node:port … myApp`



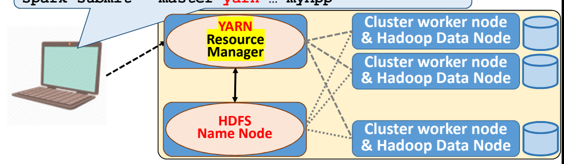
Strenght and weakness of standalone mode:

- Nothing more to install (included in Spark)
- Easy to configure
- Can run different jobs concurrently
- Can not share the cluster with non-Spark applications
- Can not launch Executors on data node hosting input data
- Limited scheduling mechanism (unique queue)

4 – Application execution on clusters and clouds

2 - with YARN cluster manager

```
export HADOOP_CONF_DIR = ${HADOOP_HOME}/conf
spark-submit --master yarn … myApp
```



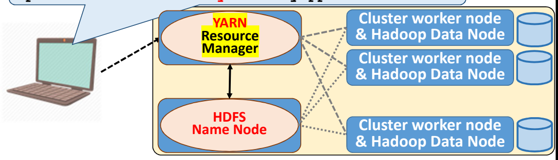
Spark cluster configuration:

- Add an env. variable defining the path to Hadoop conf directory
- Specify the maximum amount of memory per Spark Executor
- Specify the amount of CPU cores used per Spark executor
  `spark-submit --executor-cores YY …`
- Specify the nb of Spark Executors per job: `--num-executors`

4 – Application execution on clusters and clouds

2 - with YARN cluster manager

```
export HADOOP_CONF_DIR = ${HADOOP_HOME}/conf
spark-submit --master yarn … myApp
```



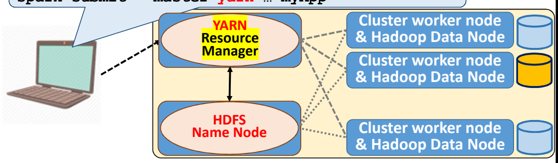
Spark cluster configuration:

- By default:
  - (only) 1GB/Spark Executor
  - (only) 1 CPU core per Spark Executor
  - (only) 2 Spark Executors per job
- Usually better with few large Executors (RAM & nb of cores)…

4 – Application execution on clusters and clouds

2 - with YARN cluster manager

```
export HADOOP_CONF_DIR = ${HADOOP_HOME}/conf
spark-submit --master yarn … myApp
```



Spark cluster configuration:

- Link Spark RDD meta-data « prefered locations » to HDFS meta-data about « localization of the input file blocks »

```
val sc = new SparkContext(sparkConf,
             InputFormatInfo.computePreferredLocations(
               Seq(new InputFormatInfo(conf,
                      classOf[org.apache.hadoop.mapred.TextInputFormat], hdfspath ))…
```

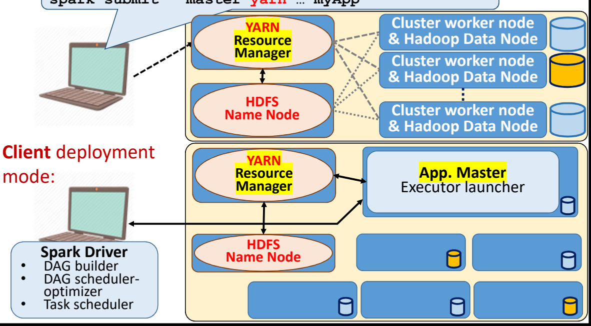
Spark Context construction

4 – Application execution on clusters and clouds

2 - with YARN cluster manager

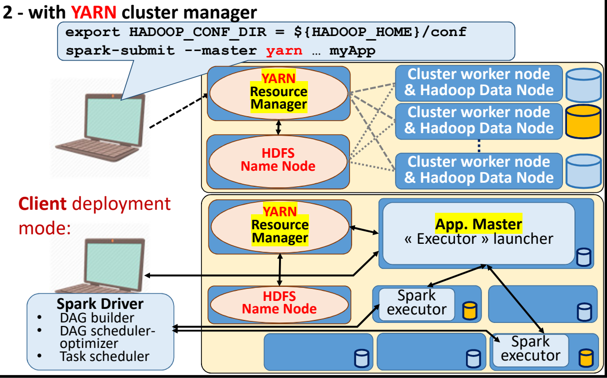
```
export HADOOP_CONF_DIR = ${HADOOP_HOME}/conf
spark-submit --master yarn … myApp
```

Client deployment mode:



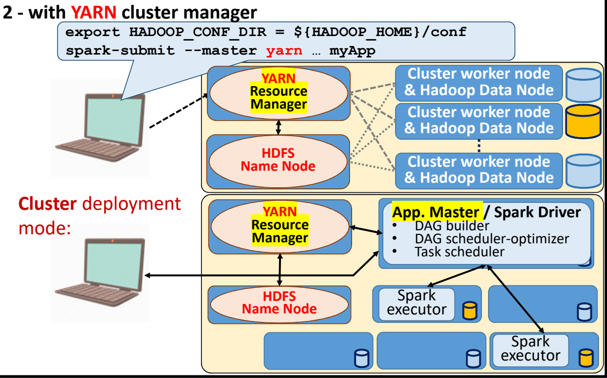
4 – Application execution on clusters and clouds

2 - with YARN cluster manager



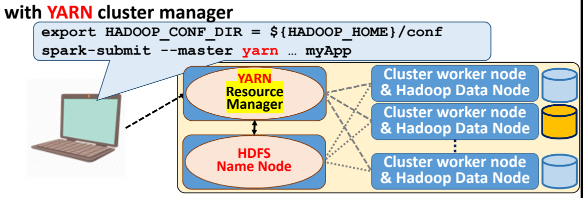
4 – Application execution on clusters and clouds

2 - with YARN cluster manager



4 – Application execution on clusters and clouds

2 - with YARN cluster manager



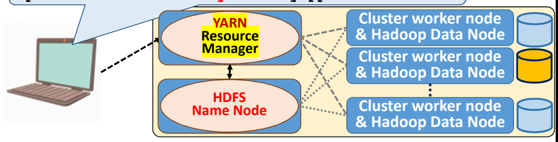
YARN *vs* standalone Spark Master:

- Usually available on HADOOP/HDFS clusters
- Allows to run Spark and other kinds of applications on HDFS
  (*better to share a Hadoop cluster*)
- Advanced application scheduling mechanisms
  (*multiple queues, managing priorities…*)

4 – Application execution on clusters and clouds

2 - with YARN cluster manager

```
export HADOOP_CONF_DIR = ${HADOOP_HOME}/conf
spark-submit --master yarn … myApp
```



YARN *vs* standalone Spark Master:

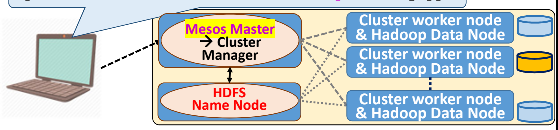
- Improvement of the data-computation locality…but is it critical ?
  - Spark reads/writes only input/output RDD from Disk/HDFS
  - Spark keeps intermediate RDD in-memory
  - With cheap disks: disk-IO time > network time

  → Better to deploy many Executors on unloaded nodes ?

4 – Application execution on clusters and clouds

3 - with MESOS cluster manager

```
spark-submit --master mesos://node:port … myApp
```



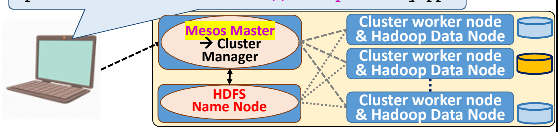
Mesos is a generic cluster manager

- Supporting to run both:
  - short term distributed computations
  - long term services (like web services)
- Compatible with HDFS

4 – Application execution on clusters and clouds

3 - with MESOS cluster manager

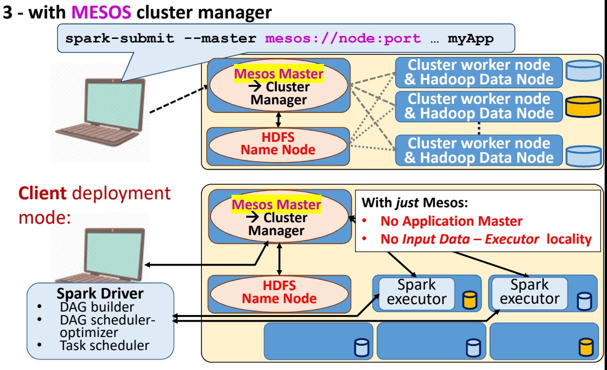
```
spark-submit --master mesos://node:port … myApp
```



- Specify the maximum amount of memory per Spark Executor
  `spark-submit --executor-memory XX …`
- Specify the total amount of CPU cores used to process one Spark application (through all its Spark executors)
  `spark-submit --total-executor-cores YY …`
- Default config:
  - create few Executors with max nb of cores (≠ standalone…)
  - use all available cores to process each job (like standalone…)

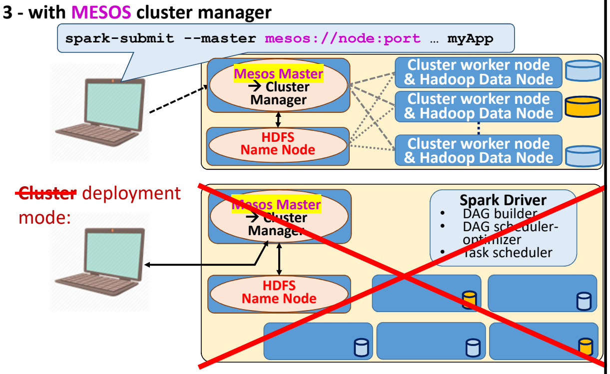
4 – Application execution on clusters and clouds

3 - with MESOS cluster manager



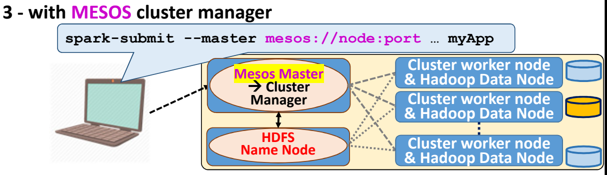
4 – Application execution on clusters and clouds

3 - with MESOS cluster manager



4 – Application execution on clusters and clouds

3 - with MESOS cluster manager

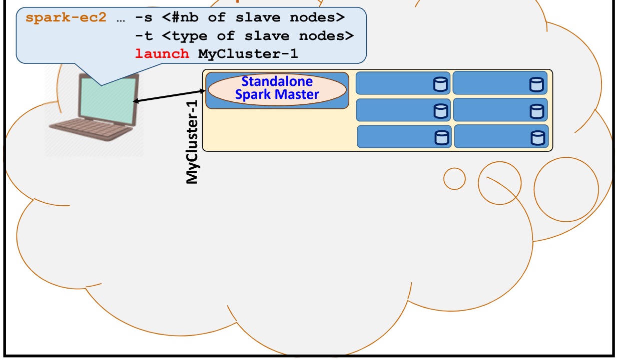


- **Coarse grained mode**: number of cores allocated to each Spark Executor are set at launching time, and cannot be changed
- **Fine grained mode**: number of cores associated to an Executor can dynamically change, function of the number of concurrent jobs and function of the load of each executor

→ Better solution/mechanism to support **many shell interpretors**
→ **But latency can increase** (*Spark Streaming* lib can be disturbed)

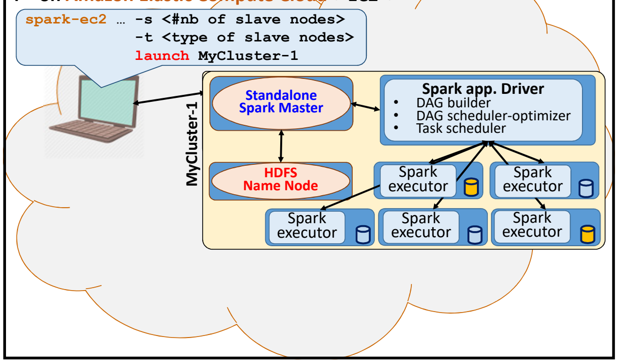
4 – Application execution on clusters and clouds

4 – on Amazon Elastic Compute Cloud « EC2 »



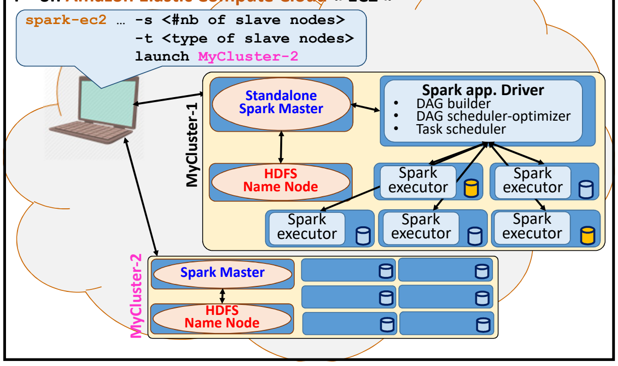
4 – Application execution on clusters and clouds

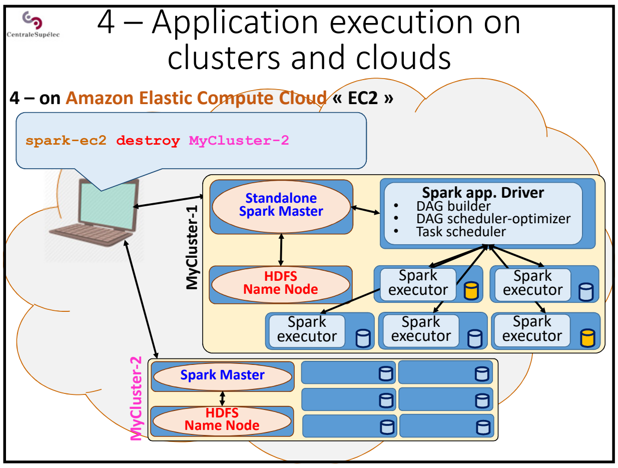
4 – on Amazon Elastic Compute Cloud « EC2 »

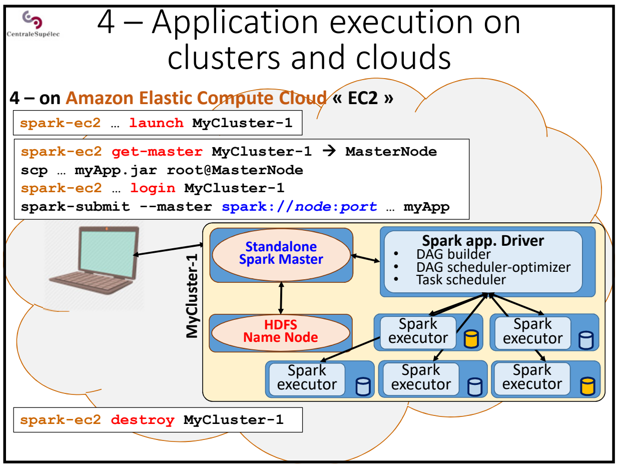


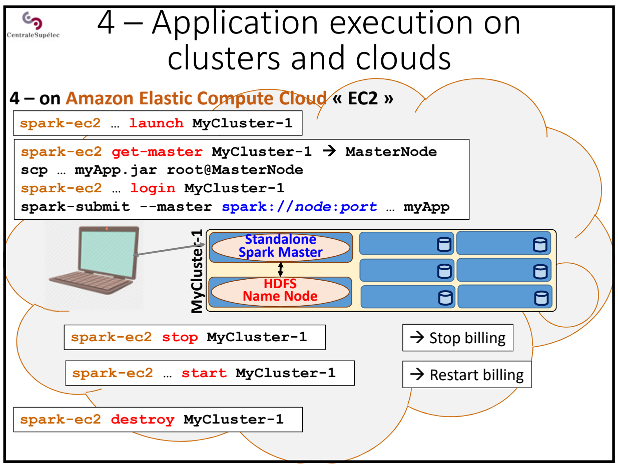
4 – Application execution on clusters and clouds

4 – on Amazon Elastic Compute Cloud « EC2 »




4 – Application execution on clusters and clouds
4 – on Amazon Elastic Compute Cloud « EC2 »
spark-ec2 destroy MyCluster-2
MyCluster-1
Standalone Spark Master
Spark app. Driver
DAG builder
DAG scheduler-optimizer
Task scheduler
HDFS Name Node
Spark executor
Spark executor
Spark executor
Spark executor
Spark executor
MyCluster-2
Spark Master
HDFS Name Node


4 – Application execution on clusters and clouds
4 – on Amazon Elastic Compute Cloud « EC2 »
spark-ec2 … launch MyCluster-1
spark-ec2 get-master MyCluster-1 → MasterNode
scp … myApp.jar root@MasterNode
spark-ec2 … login MyCluster-1
spark-submit --master spark://node:port … myApp
MyCluster-1
Standalone Spark Master
Spark app. Driver
DAG builder
DAG scheduler-optimizer
Task scheduler
HDFS Name Node
Spark executor
Spark executor
Spark executor
Spark executor
Spark executor
spark-ec2 destroy MyCluster-1


4 – Application execution on clusters and clouds
4 – on Amazon Elastic Compute Cloud « EC2 »
spark-ec2 … launch MyCluster-1
spark-ec2 get-master MyCluster-1 → MasterNode
scp … myApp.jar root@MasterNode
spark-ec2 … login MyCluster-1
spark-submit --master spark://node:port … myApp
MyCluster-1
Standalone Spark Master
HDFS Name Node
spark-ec2 stop MyCluster-1
→ Stop billing
spark-ec2 … start MyCluster-1
→ Restart billing
spark-ec2 destroy MyCluster-1

4 – Application execution on clusters and clouds

4 – on Amazon Elastic Compute Cloud « EC2 » : Bilan

Starting to learn to deploy HDFS and Spark architectures
Then, learn to deploy these architecture in a CLOUD
… or you can use a ''Spark Cluster service'' ready to use in a CLOUD!

Lear to minimize the cost (€) of a Spark cluster

- Allocate the right number of nodes
- Stop when you do not use, and re-start further

Choose to allocate reliable or preemptible machines:

- Reliable machines during all the session (standard)
- Preemptibles machines (5x less expensive!)
  → require to support to loose some tasks, or to checkpoint…
- *Machines in a HPC cloud (more expensive)*

Spark Technology

1. Spark main objectives
2. RDD concepts and operations
3. SPARK application scheme and execution
4. Application execution on clusters and clouds
5. **Basic programming examples**
6. Basic examples on pair RDDs
7. PageRank with Spark

5 – Basic programming examples

Ex. of transformations on one RDD:

rdd : {1, 2, 3, 3}

Python: `rdd.map(lambda x: x+1)` → rdd: {2, 3, 4, 4}
Scala : `rdd.map(x => x+1)` → rdd: {2, 3, 4, 4}

Scala : `rdd.map(x => x.to(3))` → rdd: {(1,2,3), (2,3), (3), (3)}
Scala : `rdd.flatMap(x => x.to(3))` → rdd: {1, 2, 3, 2, 3, 3, 3}

Scala : `rdd.filter(x => x != 1)` → rdd: {2, 3, 3}

Scala : `rdd.distinct()` → rdd: {1, 2, 3}

Some sampling functions exist:
Scala : `rdd.sample(false,0.5)` → rdd: {1} or {2,3} or …
with replacement = false

Sequence of transformations:

Scala: `rdd.filter(x => x != 1).map(x => x+1)` → rdd: {3, 4, 4}

5 – Basic programming examples

Ex. of transformations on two RDDs:

rdd : {1, 2, 3}
rdd2: {3, 4, 5}

Scala : `rdd.union(rdd2)` → rdd: {1, 2, 3, 3, 4, 5}

Scala : `rdd.intersection(rdd2)` → rdd: {3}

Scala : `rdd.subtract(rdd2)` → rdd: {1, 2}

Scala : `rdd.cartesian(rdd2)` → rdd: {(1,3), (1,4), (1,5), (2,3), (2,4), (2,5), (3,3), (3,4), (3,5)}

5 – Basic programming examples

Ex. of actions on a RDD:

Examples of « aggregations »: **computing a sum**

rdd : {1, 2, 3, 3}

Computing the sum of the RDD values:

Python : `rdd.reduce(lambda x,y: x+y)` → 9
Scala : `rdd.reduce((x,y) => x+y)` → 9

Results are NOT RDD

Specifying the initial value of the accumulator:

Scala : `rdd.fold(0)((accu,value) => accu+value)` → 9

Specifying to start to accumulate from Left or from Right:

Scala : `rdd.foldLeft(0)((accu,value) => accu+value)` → 9
Scala : `rdd.foldRight(0)((accu,value) => accu+value)` → 9

5 – Basic programming examples

Ex. of actions on a RDD:

Examples of « aggregations » :
computing an average value using aggregate(…)(…,…)

Scala:

- Specifying the initial value of the accumulator (0 = sum, 0 = nb)
- Specifying a function to add a value to an accumulator (in a rdd partition block)
- Specifying a function to add two accumulators (from two rdd partition blocks)

```
val SumNb = rdd.aggregate((0,0)) (
                (acc,v) => (acc._1+v, acc._2+1),
                (acc1,acc2) => (acc1._1+acc2._1,
                                acc1._2+acc2._2))
```

Type inference!

- Division of the sum by the nb of values

```
val avg = SumNb._1/SumNb._2.toDouble
```

5 – Basic programming examples

Ex. of actions on a RDD:

rdd : {1, 2, 3, 3}

Scala : `rdd.collect()` → {1, 2, 3, 3}

Scala : `rdd.count()` → 4

Scala : `rdd.countByValue()` → {(1,1), (2,1), (3,2)}

Scala : `rdd.take(2)` → {1, 2}

Scala : `rdd.top(2)` → {3, 3}

Scala : `rdd.takeOrdered(3,Ordering[Int].reverse)` → {3,3,2}

Scala : `rdd.takeSample(false,2)` → {?,?}

`takeSample(withReplacement, NbEltToGet, [seed])`

Scala :
```
var sum = 0
rdd.foreach(sum += _)
println(sum)
```
`rdd.foreach(sum += _)` → does not return any value

`println(sum)` → 9

Spark Technology

1. Spark main objectives
2. RDD concepts and operations
3. SPARK application scheme and execution
4. Application execution on clusters and clouds
5. Basic programming examples
6. **Basic examples on pair RDDs**
7. PageRank with Spark

6 – Basic examples on pair RDDs

Ex. of transformations on one RDD:

rdd : {(1, 2), (3, 3), (3, 4)}

Scala : `rdd.reduceByKey((x,y) => x+y)` → rdd: {(1, 2), (3, 7)}

Reduce values associated to the same key

Scala : `rdd.groupByKey((x,y) => x+y)` → rdd: {(1, [2]), (3, [3, 4])}

Group values associated to the same key

Scala : `rdd.mapValues(x => x+1)` → rdd: {(1, 3), (3, 4), (3, 5)}

Apply to each value (keys do not change)

Scala : `rdd.flatMapValues(x => x to 3)` → rdd: {(1,2), (1,3), (3,3)}

key: 1, 2 to 3 → (2, 3) → (1, 2), (1, 3),
key: 3, 3 to 3 → (3) → (3, 3)
key: 3, 4 to 3 → () → nothing
} (1,2), (1,3), (3,3)

Apply to each value (keys do not change) and flatten

6 – Basic examples on pair RDDs

Ex. of transformations on one RDD:

rdd : {(1, 2), (3, 3), (3, 4)}

Scala : `rdd.keys()` → rdd: {1, 3, 3}
Return an RDD of just the keys

Scala : `rdd.values()` → rdd: {2, 3, 4}
Return an RDD of just the values

Scala : `rdd.sortByKeys()` → rdd: {(1, 2), (3, 3), (3, 4)}
Return a pair RDD sorted by the keys

Scala :
```
rdd. combineByKey(
    …, // createCombiner function
    …, // mergeValue function          ≈ Hadoop Combiner
    …, // mergeCombiners function)     ≈ Hadoop Reduce
```
Voir plus loin…

6 – Basic examples on pair RDDs

Ex. of transformations on two pair RDDs

rdd : {(1, 2), (3, 4), (3, 6)}
rdd2: {(3, 9)}

Scala : `rdd.subtractByKey(rdd2)` → rdd: {(1, 2)}
Remove pairs with key present in the 2nd pairRDD

Scala : `rdd.join(rdd2)` → rdd: {(3, (4, 9)), (3, (6, 9))}
Inner Join between the two pair RDDs

Scala : `rdd.cogroup(rdd2)` → rdd: {(1, ([2], [])),
(3, ([4, 6], [9]))}
Group data from both RDDs sharing the same key

6 – Basic examples on pair RDDs

Ex. of classic transformations applied on a pair RDD

rdd : {(1, 2), (3, 4), (3, 6)}

A pair RDD remains a RDD of tuples (key, values)
→ Classic transformations can be applied

Scala : `rdd.filter{case (k,v) => v < 5}` → rdd: {(1, 2), (3, 4)}

Scala : `rdd.map{case (k,v) => (k,v*10)}` → rdd: {(1, 20),
(3, 40),
(3, 60)}

6 – Basic examples on pair RDDs

Ex. of actions on pair RDDs

rdd : {(1, 2), (3, 4), (3, 6)}

Scala : `rdd.countByKey()` → {(1, 1), (3, 2)}

Return a tuple of couple, counting the number of pairs per key

Scala : `rdd.collectAsMap()` → Map{(1, 2), (3, 4), (3, 6)}

Return a 'Map' datastructure containing the RDD

Scala : `rdd.lookup(3)` → [4, 6]

Return an array containing all values associated with the provided key

6 – Basic examples on pair RDDs

Ex. of transformation: Computing an average value per key

theMarks: {("julie", 12), ("marc", 10), ("albert", 19), ("julie", 15), ("albert", 15),…}

- **Solution 1: mapValues + reduceByKey + collectAsMap + foreach**

```
val theSums = theMarks
  .mapValues(v => (v, 1))
  .reduceByKey((vc1, vc2) => (vc1._1 + vc2._1,
                              vc1._2 + vc2._2))
  .collectAsMap() // Return a 'Map' datastructure
```

Bad performances! Break parallelism!

```
theSums.foreach(
      kvc => println(kvc._1 +
                     " has average:" +
                     kvc._2._1/kvc._2._2.toDouble))
```

6 – Basic examples on pair RDDs

Ex. of transformation: Computing an average value per key

theMarks: {("julie", 12), ("marc", 10), ("albert", 19), ("julie", 15), ("albert", 15),…}

- **Solution 2: combineByKey + collectAsMap + foreach**

Type inference needs some help!

```
val theSums = theMarks
  .combineByKey(
    // createCombiner function
    (valueWithNewKey)    => (valueWithNewKey, 1),
    // mergeValue function (inside a partition block)
    (acc:(Int, Int), v) =>(acc._1 + v, acc._2 + 1),
    // mergeCombiners function (after shuffle comm.)
    (acc1:(Int, Int), acc2:(Int, Int)) =>
      (acc1._1 + acc2._1, acc1._2 + acc2._2))
  .collectAsMap()
```

Still bad performances! Break parallelism!

```
theSums.foreach(
      kvc => println(kvc._1 + " has average:" +
                     kvc._2._1/kvc._2._2.toDouble))
```

6 – Basic examples on pair RDDs

Ex. of transformation: Computing an average value per key

theMarks: {("julie", 12), ("marc", 10), ("albert", 19), ("julie", 15), ("albert", 15),...}

- **Solution 2: combineByKey + map + collectAsMap + foreach**

```
val theSums = theMarks
  .combineByKey(
    // createCombiner function
    (valueWithNewKey)    => (valueWithNewKey, 1),
    // mergeValue function (inside a partition block)
    (acc:(Int, Int), v) =>(acc._1 + v, acc._2 + 1),
    // mergeCombiners function (after shuffle comm.)
    (acc1:(Int, Int), acc2:(Int, Int)) =>
      (acc1._1 + acc2._1, acc1._2 + acc2._2))
  .map{case (k,vc) => (k, vc._1/vc._2.toDouble)}
```

```
theSums.collectAsMap().foreach(
      kv => println(kv._1 + " has average:" + kv._2))
```

6 – Basic examples on pair RDDs

Tuning the level of parallelism

- By default: level of paralelism set by the nb of partition blocks of the input RDD
- When the input is a in-memory collection (list, array…), it needs to be parallelized:

```
val theData = List(("a",1), ("b",2), ("c",3),……)
sc.parallelize(theData).theTransformation(…)
```

Or :

```
val theData = List(1,2,3,……).par
theData.theTransformation(…)
```

→ Spark adopts a distribution adapted to the cluster…
… but it can be tuned

6 – Basic examples on pair RDDs

Tuning the level of parallelism

- Most of transformations support an extra parameter to control the distribution (and the parallelism)
- **Example:**

Default parallelism:

```
val theData = List(("a",1), ("b",2), ("c",3),……)
sc.parallelize(theData).reduceByKey((x,y) => x+y)
```

Tuned parallelism:

```
val theData = List(("a",1), ("b",2), ("c",3),……)
sc.parallelize(theData).reduceByKey((x,y) => x+y,8)
```

8 partition blocks imposed for the result of the reduceByKey

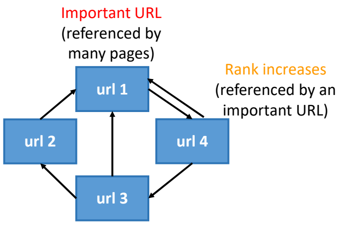
Spark Technology

1. Spark main objectives
2. RDD concepts and operations
3. SPARK application scheme and execution
4. Application execution on clusters and clouds
5. Basic programming examples
6. Basic examples on pair RDDs
7. **PageRank with Spark**

6 – PageRank with Spark

PageRank objectives

Compute the probability to arrive at a web page when randomly clicking on web links…



- If a URL is referenced by many other URLs then its rank increases (because being referenced means that it is important – ex: URL 1)
- If an important URL (like URL 1) references other URLs (like URL 4) this will increase the destination's ranking

6 – PageRank with Spark

PageRank principles

- Simplified algorithm:

$$PR(u) = \sum_{v \in B(u)} \frac{PR(v)}{L(v)}$$

Contribution of page *v* to the rank of page *u*

- $B(u)$: the set containing all pages linking to page *u*
- $PR(x)$: PageRank of page *x*
- $L(v)$: the number of outbound links of page *v*

- Initialize the PR of each page with an equi-probablity
- Iterate *k* times:
  compute PR of each page

6 – PageRank with Spark

PageRank principles

- The *damping* factor:
  the probability a user continues to click is a *damping* factor: *d*

$$PR(u) = \frac{1-d}{N_{pages}} + d.\sum_{v \in B(u)} \frac{PR(v)}{L(v)}$$

N_{pages}: Nb of documents in the collection

Usually : $d = 0.85$

Sum of all PR is 1

Variant:

$$PR(u) = (1-d) + d.\sum_{v \in B(u)} \frac{PR(v)}{L(v)}$$

Usually : $d = 0.85$

Sum of all PR is N_{pages}

6 – PageRank with Spark

PageRank first step in Spark (Scala)

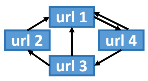
```
// read text file into Dataset[String] -> RDD1
val lines = spark.read.textFile(args(0)).rdd

val pairs = lines.map{ s =>
                      // Splits a line into an array of
                      // 2 elements according space(s)
                      val parts = s.split("\\s+")
                      // create the parts<url, url>
                      // for each line in the file
                      (parts(0), parts(1))
          }
// RDD1 <string, string> -> RDD2<string, iterable>
val links = pairs.distinct().groupByKey().cache()
```

"url 4 url 3"
"url 4 url 1"
"url 2 url 1"
"url 1 url 4"
"url 3 url 2"
"url 3 utl 1"

links RDD

url 4	[url 3, url 1]
url 3	[url 2, url 1]
url 2	[url 1]
url 1	[url 4]

6 – PageRank with Spark



PageRank second step in Spark (Scala)

Initialization with 1/N equi-probability:

```
// links <key, Iter> RDD → ranks <key,1.0/N_pages> RDD
var ranks = links.mapValues(v => 1.0/4.0)
```

`links.mapValues(…)` is an immutable RDD

`var ranks` is a mutable variable

```
var ranks = RDD1
ranks = RDD2
```

« ranks » is re-associated to a new RDD
RDD1 is forgotten …
…and will be removed from memory

Other strategy:

```
// links <key, Iter> RDD → ranks <key,one> RDD
var ranks = links.mapValues(v => 1.0)
```

links RDD

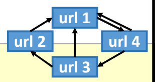
url 4	[url 3, url 1]
url 3	[url 2, url 1]
url 2	[url 1]
url 1	[url 4]

ranks RDD

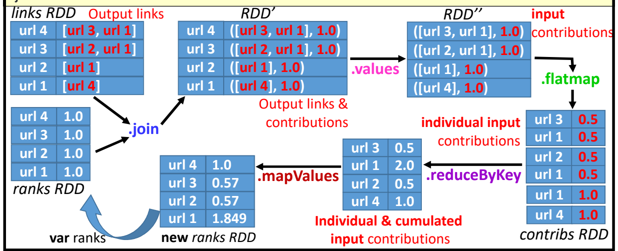
url 4	1.0
url 3	1.0
url 2	1.0
url 1	1.0

6 – PageRank with Spark

PageRank third step in Spark (Scala)




```
for (i <- 1 to iters) {
  val contribs =
        links.join(ranks)
        .values
        .flatMap{ case (urls, rank) =>
                      urls.map(url => (url, rank/urls.size )) }
   ranks = contribs.reduceByKey(_ + _)
           .mapValues(0.15 + 0.85 * _)
}
```



6 – PageRank with Spark

PageRank third step in Spark (Scala)

- Sparc & Scala allow a short/compact implementation of the PageRank algorithm
- Each RDD remains in-memory from one iteration to the next one

```
val lines = spark.read.textFile(args(0)).rdd
val pairs = lines.map{ s =>
                     val parts = s.split("\\s+")
                     (parts(0), parts(1)) }
val links = pairs.distinct().groupByKey().cache()

var ranks = links.mapValues(v => 1.0)

for (i <- 1 to iters) {
  val contribs =
       links.join(ranks)
       .values
       .flatMap{ case (urls, rank) =>
                    urls.map(url => (url, rank / urls.size )) }
  ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
}
```

Spark Technology