Big Data

Spark optimizations & deployment

Stéphane Vialle & **Gianluca Quercini**

1

Spark optimizations & deployment

1. **Wide and Narrow transformations**
2. Optimizations
3. *Page Rank* example
4. Deployment on clusters & clouds

2

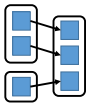
Wide and Narrow transformations

Narrow transformations

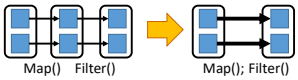
- Local computations applied to each partition block
  → no communication between processes (or nodes)
  → only local dependencies (between parent & son RDDs)

•Map()
•Filter()

•Union()



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



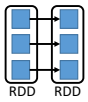
3

Wide and Narrow transformations

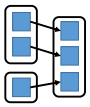
Narrow transformations

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

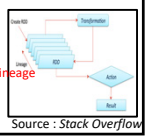
•Map()
•Filter()



•Union()



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



Source : *Stack Overflow*

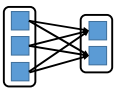
4

Wide and Narrow transformations

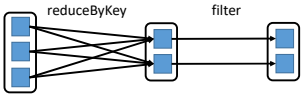
Wide transformations

- Computations requiring data from all parent RDD blocks
  → many comms between processes (and 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



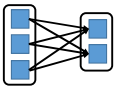
5

Wide and Narrow transformations

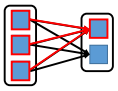
Wide transformations

- Computations requiring data from all parent RDD blocks
  → many comms between processes (and 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



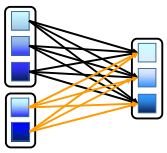
6

Wide and Narrow transformations

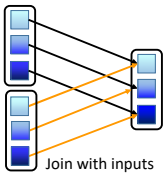
Avoiding wide transformations with co-partitioning

- 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)

7

Spark optimizations & deployment

1. Wide and Narrow transformations
2. **Optimizations**
   - **RDD Persistence**
   - RDD Co-partitionning
   - RDD controlled distribution
   - Traffic minimization
   - Maintaining parallelism
3. *Page Rank* example
4. Deployment on clusters & clouds

8

Optimizations: persistence

Persistence of the RDD

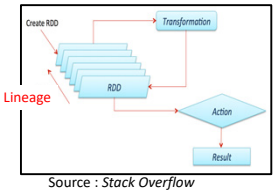
RDD are stored:

- 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



Source : *Stack Overflow*

9

Optimizations: persistence

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

10

Optimizations: persistence

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!

11

Spark optimizations & deployment

1. Wide and Narrow transformations
2. **Optimizations**
   - RDD Persistence
   - **RDD Co-partitionning**
   - RDD controlled distribution
   - Traffic minimization
   - Maintaining parallelism
3. *Page Rank* example
4. Deployment on clusters & clouds

12

Optimizations: RDD co-partitionning

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

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

To improve data locality with HDFS & YARN…

13

Optimizations: RDD co-partitionning

Specify a « partitioner »

```
val rdd2 = rdd1
        .partitionBy(new HashPartitioner(100))
        .persist()
```

Creates a new RDD (rdd2):

- partitionned according to hash partitionner strategy
- on 100 Spark Executors

→ Redistribute the RDD (rdd1 → rdd2)
→ WIDE (expensive) transformation

- Do not keep the original partition (rdd1) in memory / on disk
- keep the new partition (rrd2) in memory / on disk

→ to avoid to repeat a WIDE transformation when rdd2 is re-used

14

Optimizations: RDD co-partitionning

Specify a « partitioner »

```
val rdd2 = rdd1
        .partitionBy(new HashPartitioner(100))
        .persist()
```

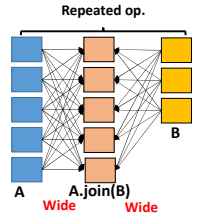
Partitionners:

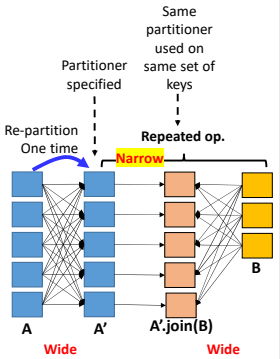
- *Hash partitioner :*
  Key0, Key0+100, Key0+200… on one Spark Executor
- *Range partitioner :*
  [Key-min ; Key-max] on one Spark Executor
- *Custom partitioner (develop your own partitioner) :*
  Ex : Key = URL, hash partitioned
  BUT : hash only the domain name of the URL
  → all pages of the same domain on the same Spark Executor because they are frequently linked

15

Optimizations: RDD co-partitionning

Avoid repetitive WIDE transformations on large data sets



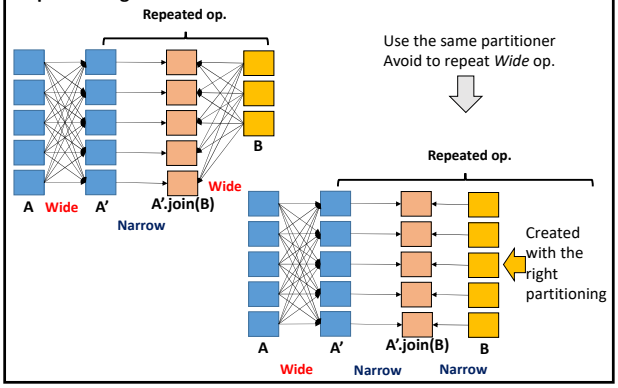


- Make ONE *Wide* op (one time) to avoid many *Wide* ops
- An explicit partitioning « propagates » to the transformation result
- Replace *Wide* op by *Narrow* op
- Do not re-partition a RDD to use only once!

16

Optimizations: RDD co-partitionning

Co-paritioning



17

Optimizations: RDD co-partitionning

PageRank with partitioner (see further)

```
val links = ……        // previous code
val links1 = links.partitionBy(new HashPartitioner(100)).persist()

var ranks = links1.mapValues(v => 1.0)

for (i <- 1 to iters) {
  val contribs =
       links1.join(ranks)
       .flatMap{ case (url (urlLinks, rank)) =>
                 urlLinks.map(dest => (dest,rank/urlLinks.size))}
  ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
}
```

- Initial **links** and **ranks** are co-partitioned
- Repeated **join** is Narrow-Wide
- Repeated **mapValues** is Narrow: respects the **reduceByKey** partitioning
- Pb: flatMap{…urlinks.map(…)} can change the partitionning ?!

18

Spark optimizations & deployment

1. Wide and Narrow transformations
2. **Optimizations**
   - RDD Persistence
   - RDD Co-partitionning
   - **RDD controlled distribution**
   - Traffic minimization
   - Maintaining parallelism
3. *Page Rank* example
4. Deployment on clusters & clouds

19

Optimization: RDD distribution

Create and distribute a RDD

- By default: level of parallelism 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

20

Optimization: RDD distribution

Control of the RDD distribution

- 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)
```

But better to use PARTITIONERS…

8 partition blocks imposed for the result of the reduceByKey

21

Spark optimizations & deployment

1. Wide and Narrow transformations
2. **Optimizations**
   - RDD Persistence
   - RDD Co-partitionning
   - RDD controlled distribution
   - **Traffic minimization**
   - Maintaining parallelism
3. *Page Rank* example
4. Deployment on clusters & clouds

22

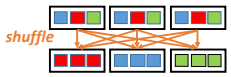
Optimization: traffic minimization

RDD redistribution:

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

Scala : `rdd.groupByKey()` → rdd: {(1, [2]), (3, [3, 4])}

Group values associated to the same key



→ Move almost all input data
→ Huge trafic in the shuffle step !!

`groupByKey` will be time consumming:
- no computation time…
- … but huge traffic on the network of the cluster/cloud

→ Optimize computations and communications in a Spark program

23

Optimization: traffic minimization

RDD reduction:

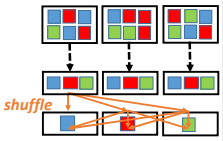
rdd : {(1, 2), (3, 3), (3, 4)}

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

Reduce values associated to the same key

`((x,y) => x+y)`:
`1 int + 1 int` → `1 int`

→ **Limited trafic in the shuffle step**



But: `((x,y) => x+y)`:
`1 list + 1 list` → `1 longer list`

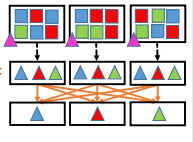
→ TD-1

24

Optimization: traffic minimization

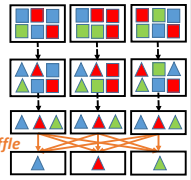
RDD reduction with different input and reduced datatypes:

```
Scala  : rdd.aggregateByKey(init_acc) (
           …, // mergeValueAccumulator fct
           …, // mergeAccumulators fct
         )
```



```
Scala  : rdd.combineByKey(
           …, // createAccumulator fct
           …, // mergeValueAccumulator fct
           …, // mergeAccumulators fct
         )
```

See further



25

Spark optimizations & deployment

1. Wide and Narrow transformations
2. **Optimizations**
   - RDD Persistence
   - RDD Co-partitionning
   - RDD controlled distribution
   - Traffic minimization
   - **Maintaining parallelism**
3. *Page Rank* example
4. Deployment on clusters & clouds

26

Optimization: maintaining parallelism

Computing an average value per key in parallel

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
```

ACTION → Break parallelism! Bad performances!

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

Sequential computing !

27

Optimization: maintaining parallelism

Computing an average value per key in parallel

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

- **Solution 2: combineByKey + 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))
  .collectAsMap()
```

Type inference needs some help!

Still bad performances! (Break parallelism)

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

Still sequential !

28

Optimization: maintaining parallelism

Computing an average value per key in parallel

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)}
```

Transformation: compute in parallel and return a RDD

```
theSums.collectAsMap().foreach( Action: at the end (just to print)
      kv => println(kv._1 + " has average:" + kv._2))
```

29

Spark optimizations & deployment

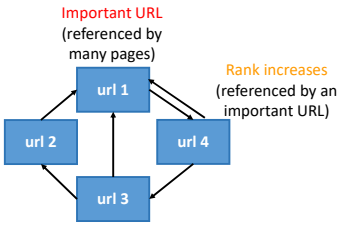
1. Wide and Narrow transformations
2. Optimizations
3. ***Page Rank* example**
4. Deployment on clusters & clouds
   - Task DAG execution
   - Spark execution on clusters
   - Ex of Spark execution on cloud

30

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

31

PageRank with Spark

PageRank principles

- Simplified algorithm:

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

$B(u)$: 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*

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

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

32

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}

33

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"

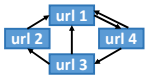
links RDD

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

34

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]

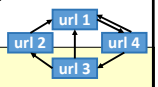
ranks RDD

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

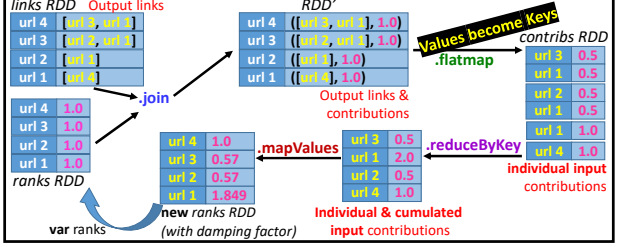
35

PageRank with Spark

PageRank third step in Spark (Scala)




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



36

PageRank with Spark

PageRank third step in Spark (Scala)

- Spark & 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)
       .flatMap{ case (url (urlLinks, rank)) =>
                 urlLinks.map(dest => (dest,rank/urlLinks.size))}
  ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
}
```

37

PageRank with Spark

PageRank third step in Spark (Scala): optimized with partitioner

```
val links = ……       // previous code
val links1 = links.partitionBy(new HashPartitioner(100)).persist()

var ranks = links1.mapValues(v => 1.0)

for (i <- 1 to iters) {
  val contribs =
       links1.join(ranks)
       .flatMap{ case (url (urlLinks, rank)) =>
                 urlLinks.map(dest => (dest,rank/urlLinks.size))}
  ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
}
```

- Initial **links** and **ranks** are co-partitioned
- Repeated **join** is Narrow-Wide
- Repeated **mapValues** is Narrow: respects the **reduceByKey** partitioning
- Pb: flatMap{…urlinks.map(…)} can change the partitionning ?!

38

Spark optimizations & deployment

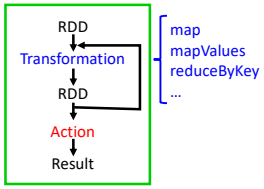
1. Wide and Narrow transformations
2. Optimizations
3. *Page Rank* example
4. **Deployment on clusters & clouds**
   - **Task DAG execution**
   - Spark execution on clusters
   - Ex of Spark execution on cloud

39

Task DAG execution

- *A RDD is a dataset distributed among the Spark compute nodes*
- *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
→ A DAG of tasks

40

Task DAG 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)

41

Task DAG execution

Spark job trace: on 10 Spark executors, with 3GB input file

```
DAGScheduler: Submitting 24 missing tasks from ShuffleMapStage 0 ...
TaskSchedulerImpl: Adding task set 0.0 with 24 tasks
...

TaskSetManager: Starting task 1.0 in stage 0.0 (TID 1, 172.20.10.14, executor 0, partition 1, ...)
TaskSetManager: Starting task 2.0 in stage 0.0 (TID 2, 172.20.10.11, executor 7, partition 2, ...)
...
TaskSetManager: Starting task 10.0 in stage 0.0 (TID 10, 172.20.10.11, executor 7, partition 10, ...)

TaskSetManager: Finished task 2.0 in stage 0.0 (TID 2) in 18274 ms … (executor 7) (1/24)
TaskSetManager: Starting task 11.0 in stage 0.0 (TID 11, 172.20.10.7, executor 8, partition 11, ...)
TaskSetManager: Finished task 8.0 in stage 0.0 (TID 8) in 18459 ms … (executor 8) (2/24)
...

TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks
                   have all completed, from pool
...
```

Submitting the 10 first tasks on the 10 Spark executor processes

Submitting a new task when a previous one has finished

End of task graph execution

42

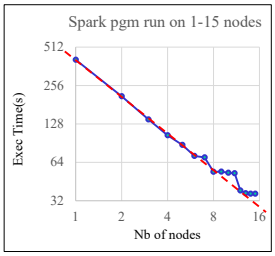
Task DAG execution

Execution time as a function of the number of Spark executors

Ex. of Spark application run:
- from 1 up to 15 executors
- with 1 executor per node

Good overall decrease but plateaus appear !

Probable **load balancing problem**…



Ex: a graph of 4 parallel tasks

on 1 node: T | on 2 nodes: T/2 | on 3 nodes: T/**2** → A plateau appears

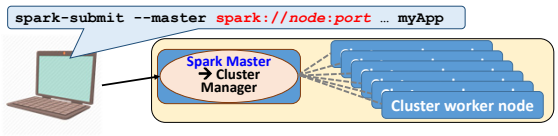
43

Spark optimizations & deployment

1. Wide and Narrow transformations
2. Optimizations
3. *Page Rank* example
4. **Deployment on clusters & clouds**
   - Task DAG execution
   - **Spark execution on clusters**
     - **Using the Spark cluster manager (standalone mode)**
     - Using YARN as cluster manager
     - Using Mesos as cluster manager
   - Ex of Spark execution on cloud

44

Using the Spark Master as cluster manager (standalone mode)

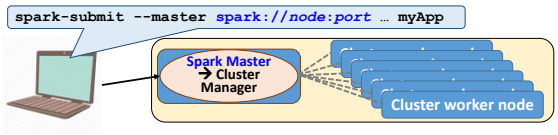


Spark cluster configuration:

- 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 …`

45

Using the Spark Master as cluster manager (standalone mode)

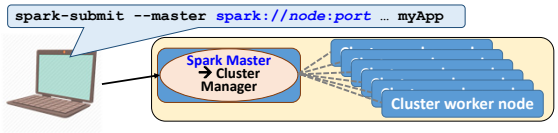


Spark cluster configuration:

- 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

46

Using the Spark Master as cluster manager (standalone mode)



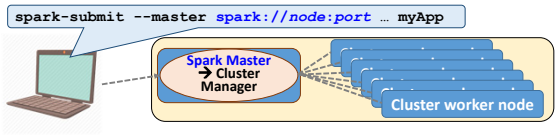
Spark cluster configuration:

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

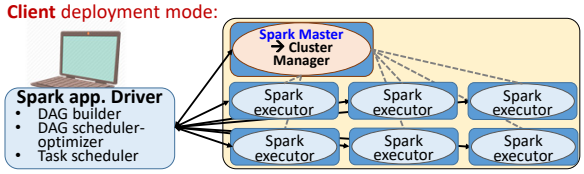
Exp 2019

47

Using the Spark Master as cluster manager (standalone mode)



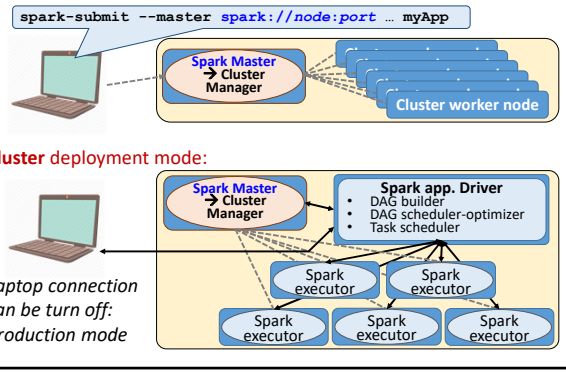
Client deployment mode:



Interactive control of the application: development mode

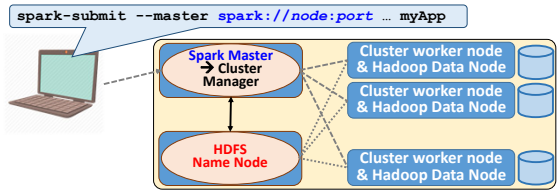
48

Using the Spark Master as cluster manager (standalone mode)



49

Using the Spark Master as cluster manager (standalone mode)



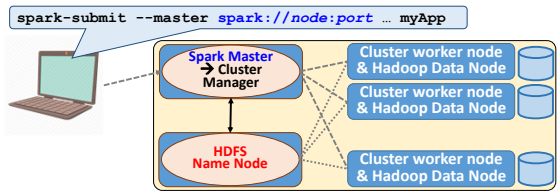
The Cluster Worker nodes should be the Data nodes, storing initial RDD values or new generated (and saved) RDD

→ Will improve the global data-computations locality

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

50

Using the Spark Master as cluster manager (standalone mode)



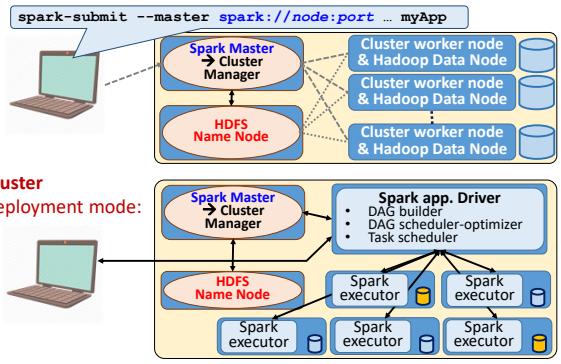
The Cluster Worker nodes should be the Data nodes, storing initial RDD values or new generated (and saved) RDD

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!

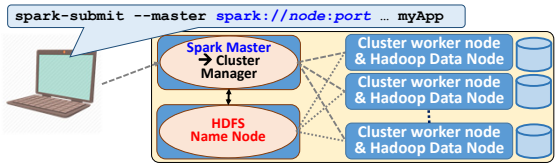
51

Using the Spark Master as cluster manager (standalone mode)



52

Using the Spark Master as cluster manager (standalone mode)



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 the data nodes hosting input data
- Limited scheduling mechanism (unique queue)

53

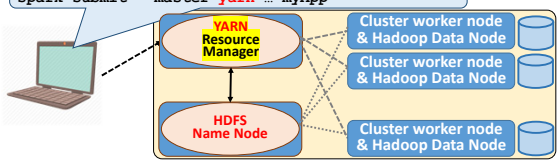
Spark optimizations & deployment

1. Wide and Narrow transformations
2. Optimizations
3. *Page Rank* example
4. **Deployment on clusters & clouds**
   - Task DAG execution
   - **Spark execution on clusters**
     - Using the Spark cluster manager (standalone mode)
     - **Using YARN as cluster manager**
     - Using Mesos as cluster manager
   - Ex of Spark execution on cloud

54

Using YARN as cluster manager

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



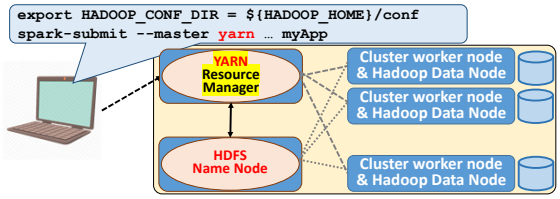
Spark cluster configuration:

- 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`

55

Using YARN as cluster manager

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



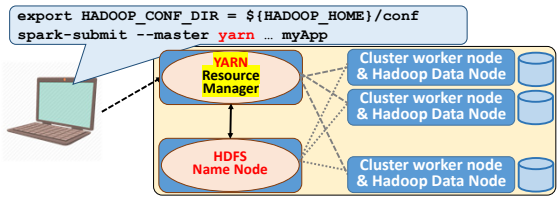
Spark cluster configuration:

- 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)…

56

Using YARN as cluster manager

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



Spark cluster configuration:

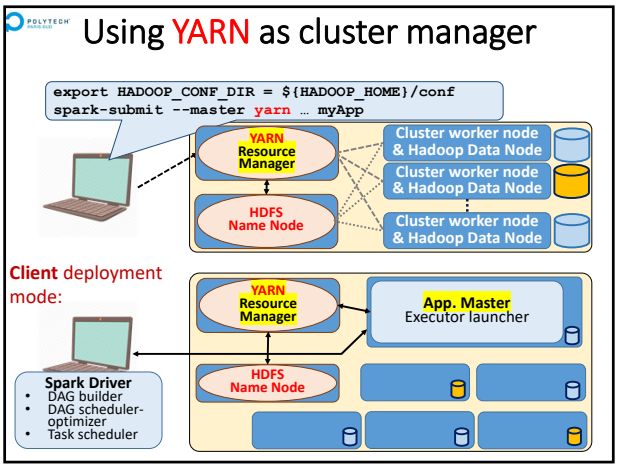
- Link Spark RDD meta-data « prefered locations » to HDFS meta-data about « localization of the input file blocks »

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

Spark Context construction

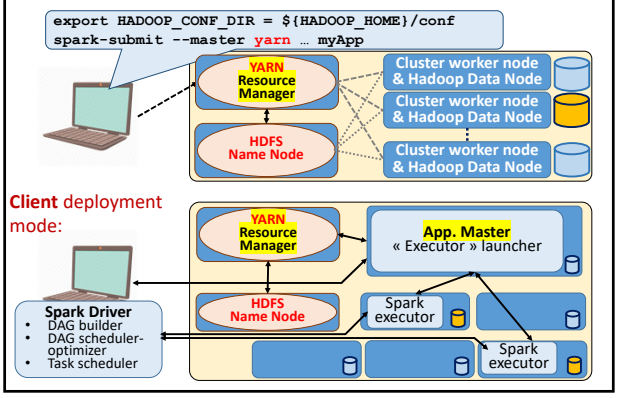
57

Using YARN as cluster manager



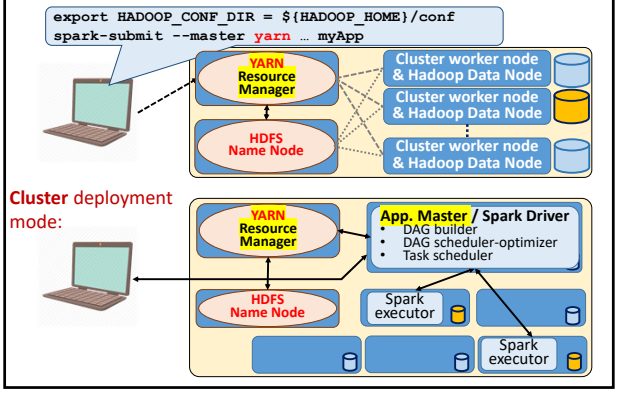
58

Using YARN as cluster manager



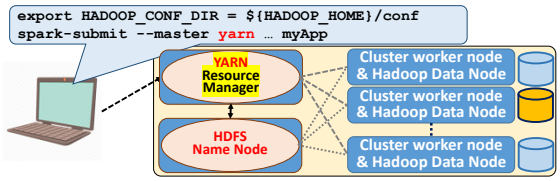
59

Using YARN as cluster manager



60

Using YARN as cluster manager

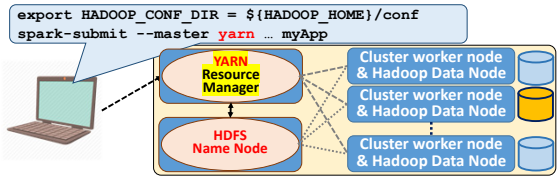


YARN *vs* standalone Spark Master:

- 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…*)

61

Using YARN as cluster manager



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 ?

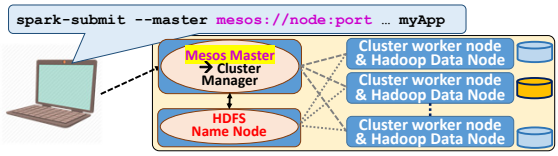
62

Spark optimizations & deployment

1. Wide and Narrow transformations
2. Optimizations
3. *Page Rank* example
4. **Deployment on clusters & clouds**
   - Task DAG execution
   - **Spark execution on clusters**
     - Using the Spark cluster manager (standalone mode)
     - Using YARN as cluster manager
     - **Using Mesos as cluster manager**
   - Ex of Spark execution on cloud

63

Using MESOS as cluster manager

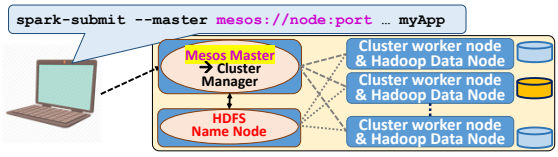


Mesos is a generic cluster manager

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

64

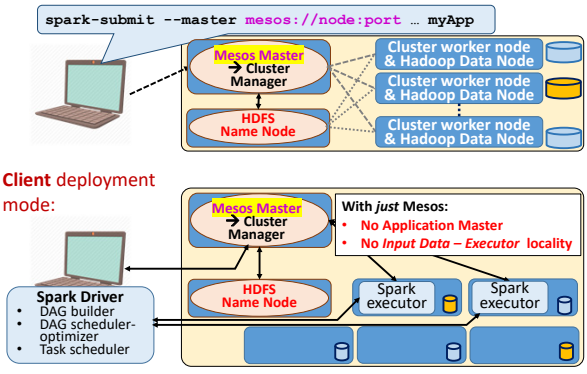
Using MESOS cluster manager



- 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 → like standalone…
  - use all available cores to process each job …in 2019

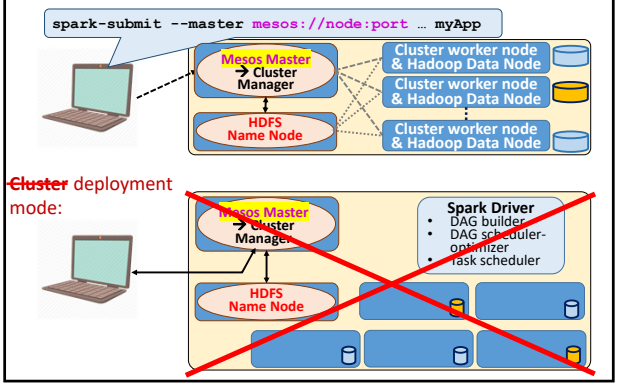
65

Using MESOS as cluster manager



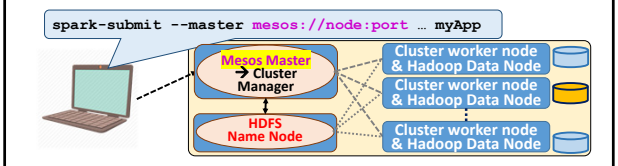
66

Using MESOS as cluster manager



67

Using MESOS as 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 **(specificity!)**

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

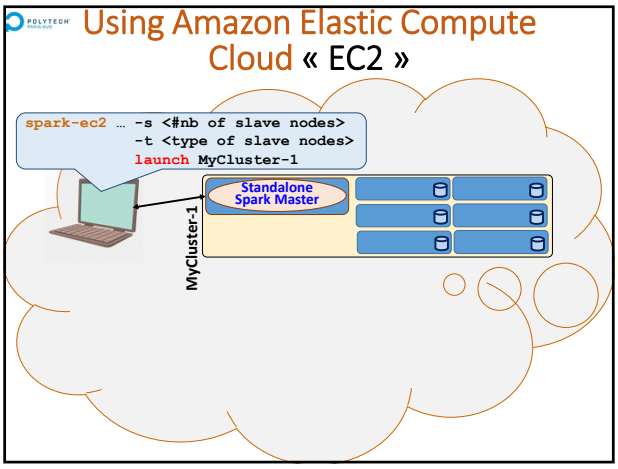
68

Spark optimizations & deployment

1. Wide and Narrow transformations
2. Optimizations
3. *Page Rank* example
4. **Deployment on clusters & clouds**
   - Task DAG execution
   - Spark execution on clusters
   - **Ex of Spark execution on cloud**

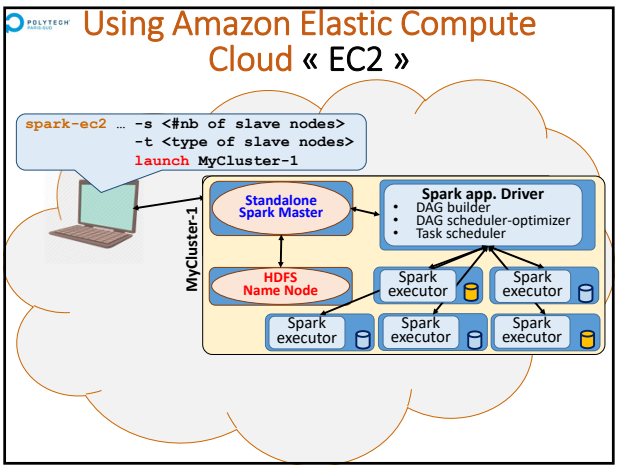
69

Using Amazon Elastic Compute Cloud « EC2 »



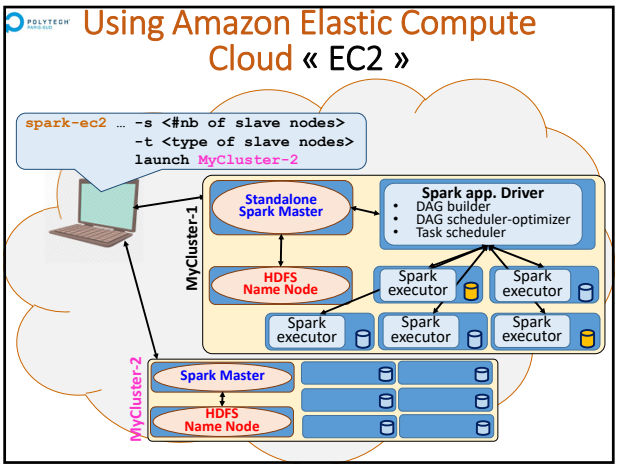
70

Using Amazon Elastic Compute Cloud « EC2 »



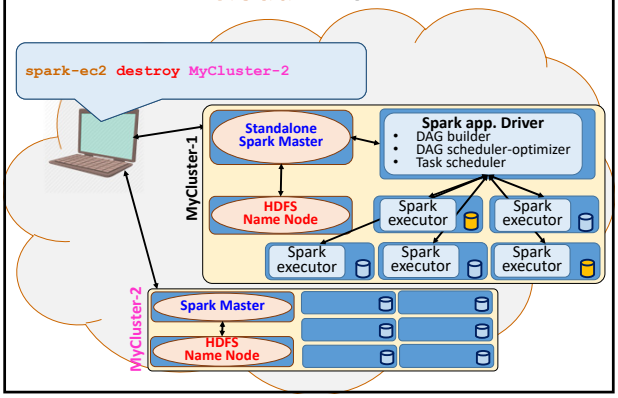
71

Using Amazon Elastic Compute Cloud « EC2 »



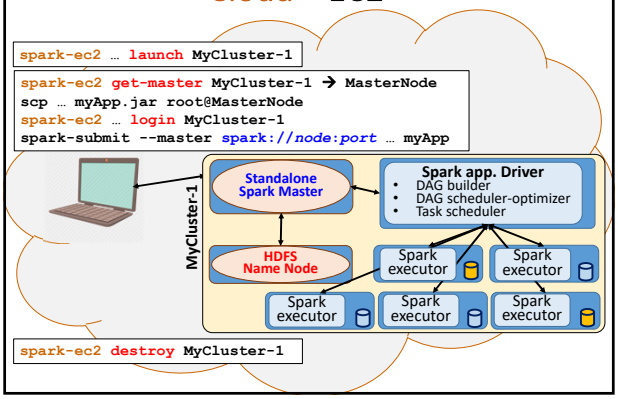
72

Using Amazon Elastic Compute Cloud « EC2 »



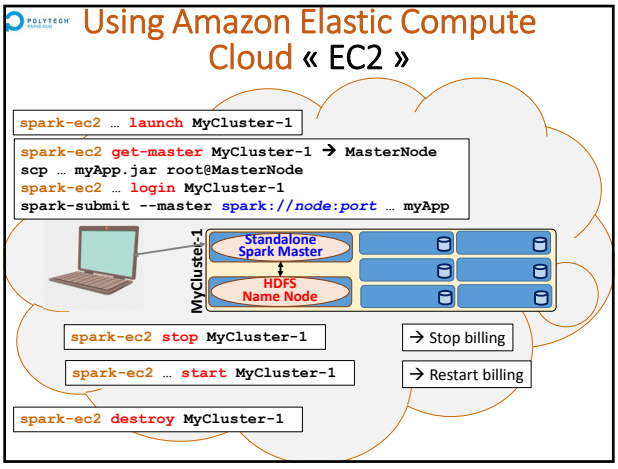
73

Using Amazon Elastic Compute Cloud « EC2 »



74

Using Amazon Elastic Compute Cloud « EC2 »



75

Using Amazon Elastic Compute Cloud « EC2 »

Start to learn to deploy HDFS and Spark architectures

Then, learn to deploy these architectecture in a CLOUD
… or use a ''Spark Cluster service'': ready to use in a CLOUD!

Learn 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…

76

Spark optimizations & deployment

77