TECHNO DU BIG-DATA

Stéphane Derrode, Dpt MI / ECLyon
Stephane.derrode@ec-lyon.fr

Apache Spark - Sommaire

1. Programmation fonctionnelle en Python
2. Spark versus Map-Reduce
3. Principes de PySpark
4. Exemples PySpark
5. EcoSystème Spark

PROGRAMMATION FONCTIONNELLE

avec Python

Programmation fonctionnelle

La caractéristique essentielle d'un langage de programmation fonctionnelle (FP: *functional programing*) est l'absence d'effet de bord. En effet, tout est *immutable* (les données ne peuvent pas être modifiées).

Ainsi, par exemple:

```
element = liste.pop()
```

serait remplacé par :

```
liste, element = liste[:-1], liste[-1]
```

Egalement:

```
def size(l):
    s=0
    for x in l:
        s+=1
    return s
```

```
def size(l):
    if not l:
        return 0
    return 1+size(l[1:])
```

Le code ne dépend pas de données se trouvant à l'extérieur de la fonction courante et il ne modifie pas des données à l'extérieur de cette fonction.

Il devient alors plus facile de prouver qu'un programme marche, de le tester, et surtout, de travailler de manière concurrente : si rien ne se modifie, nul besoin de lock et de synchronisation car rien n'est partagé et toute instruction a toujours le même comportement quel que soit le contexte.

Python et la FP

Paradigmes de programmation de Python :

- structurée,
- objet et …
- FP

Caractéristiques fonctionnelles de Python :

- Bien que les listes, les sets, les objets et les dictionnaires soient mutables, les tuples, les chaînes et les entiers ne le sont pas.
- On peut manipuler les fonctions elles-mêmes, les créer dynamiquement, les retourner et les passer en paramètre.

Outils de base :

- ***map():*** applique une fct qui transforme chaque élément d'un itérable pour en obtenir un nouveau.
- ***filter():*** appliquer une fonction pour filtrer chaque élément d'un itérable et en obtenir un nouveau.
- ***reduce():*** appliquer une fonction aux deux premiers éléments d'un itérable, puis au résultat de cette fonction et à l'élément suivant, et ainsi de suite, jusqu'à obtenir un résultat.
- **Les fonctions anonymes :** des fonctions qui peuvent se définir sans bloc ni nom.
- **La récursion** : une fonction peut s'appeler elle-même.

Lambda function en python (fonction anonyme)

Python permet d'écrire des mini-fonctions, pas plus longue qu'une ligne, à la volée, sans nécessairement leur donner de nom. Il s'agit des *lambda functions:*

```
def f(x):
        return x*x

if __name__ == "__main__":
        # fonction classique
        print(f(3))

        # Lambda fonction nommée
        g=lambda x: x*x
        print(g(3))

        # Lambda fonction sans nom
        print((lambda x: x*x)(3))
```

Lambda_exemple.py

Remarque : Les fonctions lambda sont une question de style. Les utiliser n'est jamais une nécessité : partout vous pouvez utiliser une fonction ordinaire. On les utilise là où on veut incorporer du code spécifique et non réutilisable sans encombrer le code de multiples fonctions d'une seule ligne.

Programmation fonctionnelle : map

La fonction *map* prend en argument une fonction et une collection de données. Elle crée une nouvelle collection vide, applique la fonction à chaque élément de la collection d'origine et insère les valeurs de retour produites dans la nouvelle collection. Finalement, elle renvoie la nouvelle collection.

Exemple 1

```
squares=map(lambda x: x*x, [0,1,2,3,4])
print(list(squares))
# =>[0, 1, 4, 9, 16]
```

Exemple 2

```
import random

names=['Mary', 'Isla’, 'Sam']
code_names=['Mr. Pink', 'Mr. Orange', 'Mr. Blonde']

secret_names=map(lambda x: random.choice(code_names), names)
print(list(secret_names))
# => ['Mr. Orange', 'Mr. Orange', 'Mr. Orange']
```

map_exemples.py

Programmation fonctionnelle : map

Exemple 3

```
def multiply(x): return(x*x)
def add(x): return(x+x)

funcs=[multiply, add]
for i in range(2,5):
   value=list(map(lambda x: x(i), funcs))
   print(value)
#    [4, 4]
#    [9, 6]
#    [16, 8]
```

map_exemples.py

Exemple 4

```
a=[1,2,3,4]
b=[17,12,11,10]
c=[-1,-4,5,9]
print(list(map(lambda x,y: x+y, a,b)))
# =>[18, 14, 14, 14]
print(list(map(lambda x,y,z: x+y-z, a,b,c)))
# =>[19, 18, 9, 5]
```

Programmation fonctionnelle : filter

La fonction *filter* prend en entrée une fonction et une collection. Elle renvoie une nouvelle collection contenant tous les éléments de la collection d'origine pour laquelle la fonction renvoie une valeur vraie (*True*).

Exemple 1

```
number_list=range(-5,5)
less_than_zero=list(filter(lambda x: x<0, number_list))
print(less_than_zero)
# => [-5, -4, -3, -2, -1]
```

Exemple 2

```
carre=map(lambda x: x*x, filter(lambda x: x%2, range(10)))
print(list(carre))
# => [1, 9, 25, 49, 81]

carre2=[x*x for x in range(10) if x%2]
print(list(carre2))
# => [1, 9, 25, 49, 81]
```

filter_exemples.py

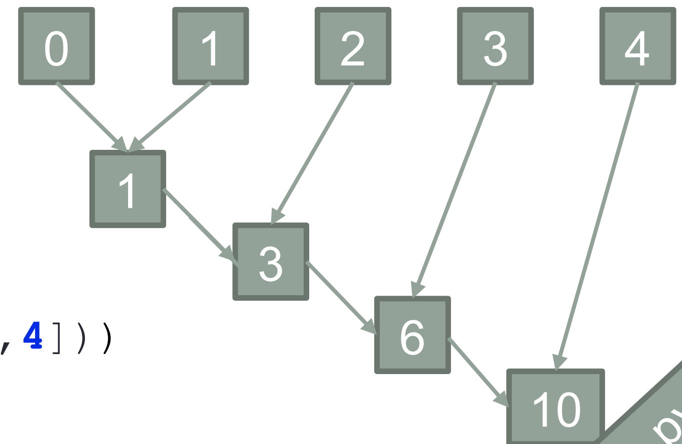
La fonction *filter* ressemble à une boucle mais c’est une fonction intégrée et elle est plus rapide!

Programmation fonctionnelle : reduce

La fonction *reduce* prend en entrée une fonction et une collection. Elle renvoie une valeur créée en combinant les éléments de la collection.
Attention : il faut que la fonction soit associative pour que l'opération de réduction fonctionne. P. ex., la moyenne ne l'est pas car :

moyenne(moyenne(a, b), c) != moyenne(a, moyenne(b, c))

La fonction doit prendre deux paramètres en entrée, et retourner une valeur. Au premier appel, les deux premiers éléments de l'itérable sont passés en paramètres. Ensuite, le résultat de cet appel et l'élément suivant sont passés en paramètre, et ainsi de suite.



Exemple 1

```
from functools import reduce
print(reduce(lambda a, x: a+x, [0,1,2,3,4]))
# =>10
```

La fonction *lambda* prend 2 paramètres *:*

- *x* est l'élément courant de l'itération et
- *a* est l'accumulateur.

Programmation fonctionnelle : reduce

La fonction *reduce* prend en entrée une fonction et une collection. Elle renvoie une valeur créée en combinant les éléments de la collection.
Attention : il faut que la fonction soit associative pour que l'opération de réduction fonctionne. P. ex., la moyenne ne l'est pas car :

moyenne(moyenne(a, b), c) != moyenne(a, moyenne(b, c))

La fonction doit prendre deux paramètres en entrée, et retourner une valeur. Au premier appel, les deux premiers éléments de l'itérable sont passés en paramètres. Ensuite, le résultat de cet appel et l'élément suivant sont passés en paramètre, et ainsi de suite.

Exemple 2

```
from functools import reduce
sentences=['Mary read a story to Sam and Isla.',
           'Isla cuddled Sam.',
           'Sam chortled.']
print(reduce(lambda a, x: a+x.count('Sam'), sentences, 0))
#  =>3
```

On peut préciser la valeur initiale de l'accumulateur (ici 0).

Programmation fonctionnelle : map + filter + reduce

Exemple : programmation classique

```
people=[{'name':'Mary','height':160},
        {'name':'Isla','height':80},
        {'name':'Sam'}]

height_total=0
height_count=0
for person in people:
   if 'height' in person:
       height_total+=person['height']
       height_count+=1

average_height=height_total/height_count
print(average_height)
```

mfr_exemples.py

Programmation fonctionnelle : map + filter + reduce

Exemple : programmation fonctionnelle

```
people=[{'name':'Mary','height':160},
        {'name':'Isla','height':80},
        {'name':'Sam'}]

heights=list(map   (lambda x: x['height'],
             filter(lambda x: 'height' in x, people)))

from operator import add
average_height=reduce(add, heights)/len(heights)
print(average_height)
```

mfr_exemples.py

Dans les sections suivantes, vous allez rencontrer non pas les fonctions Python « map() », « reduce() », « filter() », mais leur équivalent sous forme de méthodes PySpark qui s'appliquent sur objets de type RDD (*Resilient Distributed Data sets*). C'est donc de la programmation objet.

La syntaxe et le fonctionnement restent sensiblement les mêmes :

```
monRDD = sc.parallelize([1,2,3,4])
print(monRDD.map(lambda n: n+1).collect())
print(monRDD.filter(lambda n: (n%2)==0).collect())
```

La méthode *collect()* sera expliquée plus tard.

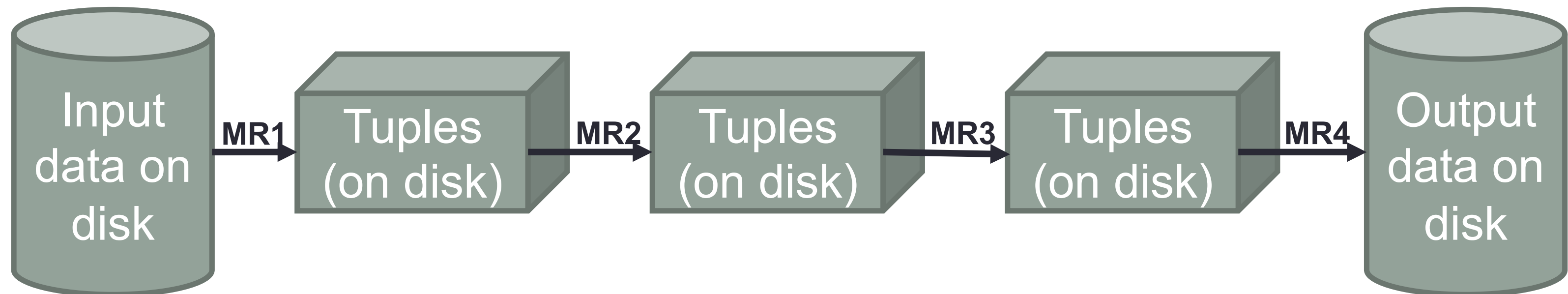
SPARK

Principes et fonctionnement

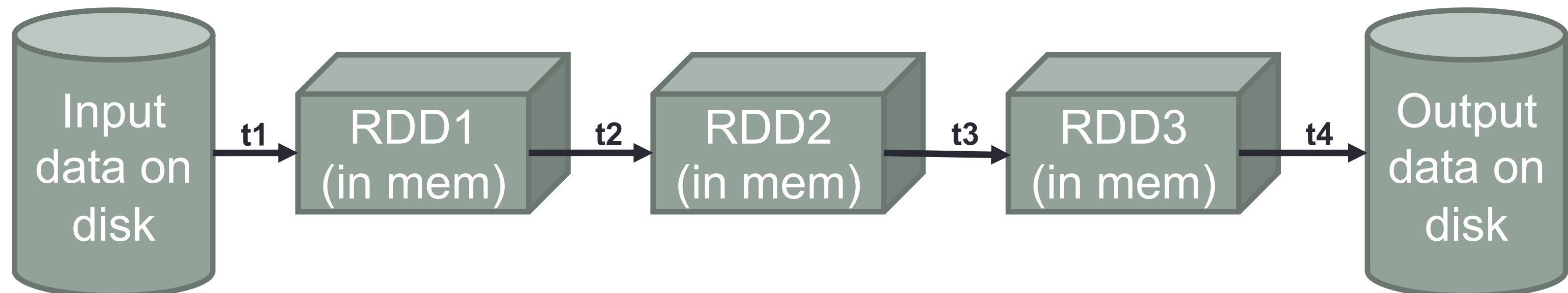
Pourquoi Spark?

- La plupart des algorithmes de Machine Learning sont itératifs : chaque itération améliore le résultat
- Avec des méthodes orientées Disque Dur (DD - Hadoop), le résultat de chaque itération est écrit sur le DD, ce qui ralenti le processus.

Hadoop execution flow



Spark execution flow



Pourquoi Spark?

Daytona Gray Sort 100 TB benchmark

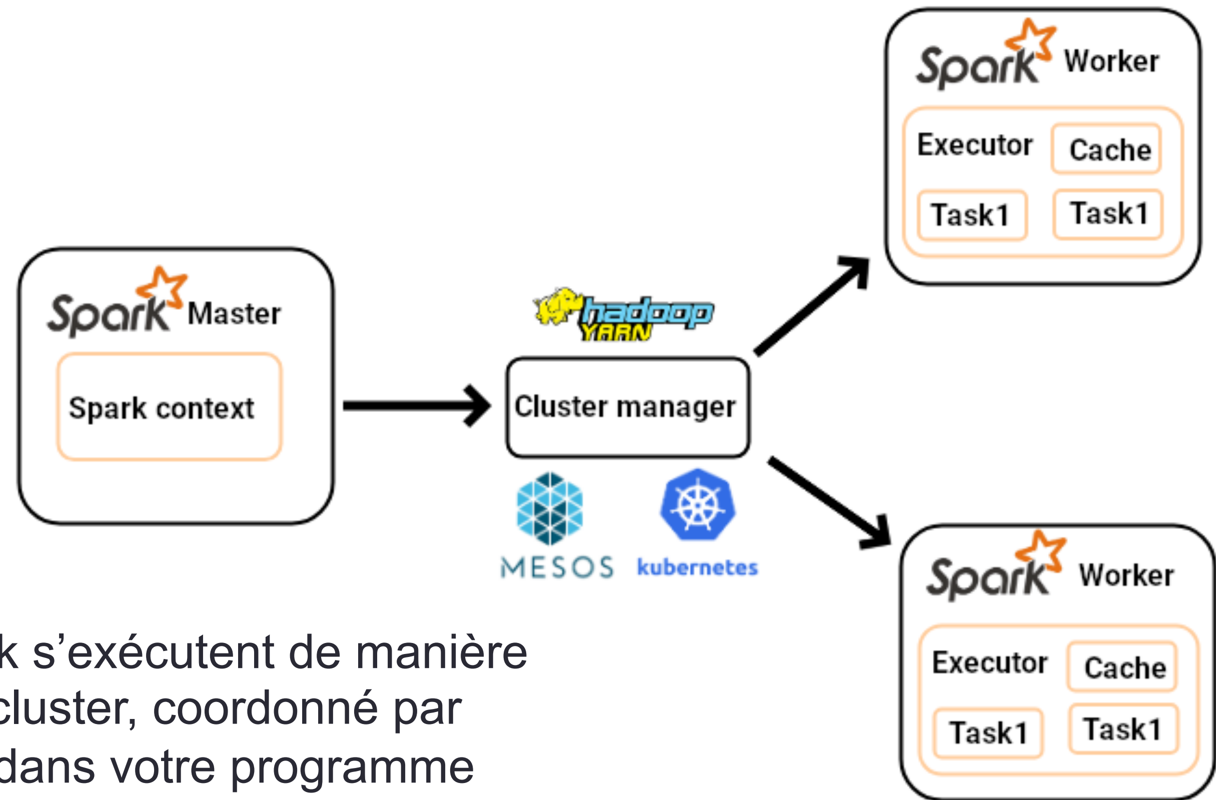
	Data Size	Time	Nodes	Cores
Hadoop MR (2013)	102.5 TB	72 min	2,100	50,400 physical
Apache Spark (2014)	100 TB	23 min	206	6,592 virtualized

Source : https://databricks.com/fr/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html

Pourquoi Spark ?

- Commence à Berkeley Univ. en 2009. Devenu un projet de la fondation Apache depuis 2013.
- http://spark.apache.org « Unified engine for large-scale data analytics ». *Apache Spark™ is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.*
- 10x (sur DD) – 100x (en mémoire) plus rapide
- Fournit des API pour Java, Scala, Python, R et SQL.
- S'intègre à Hadoop et son écosystème (notamment HDFS)
- Spark fonctionne :
  - **En mode « standalone »**
  - Sur Amazon EC2 (Service d'hébergement « cloud »)
  - **Sur Hadoop Yarn**
  - Sur Apache mesos

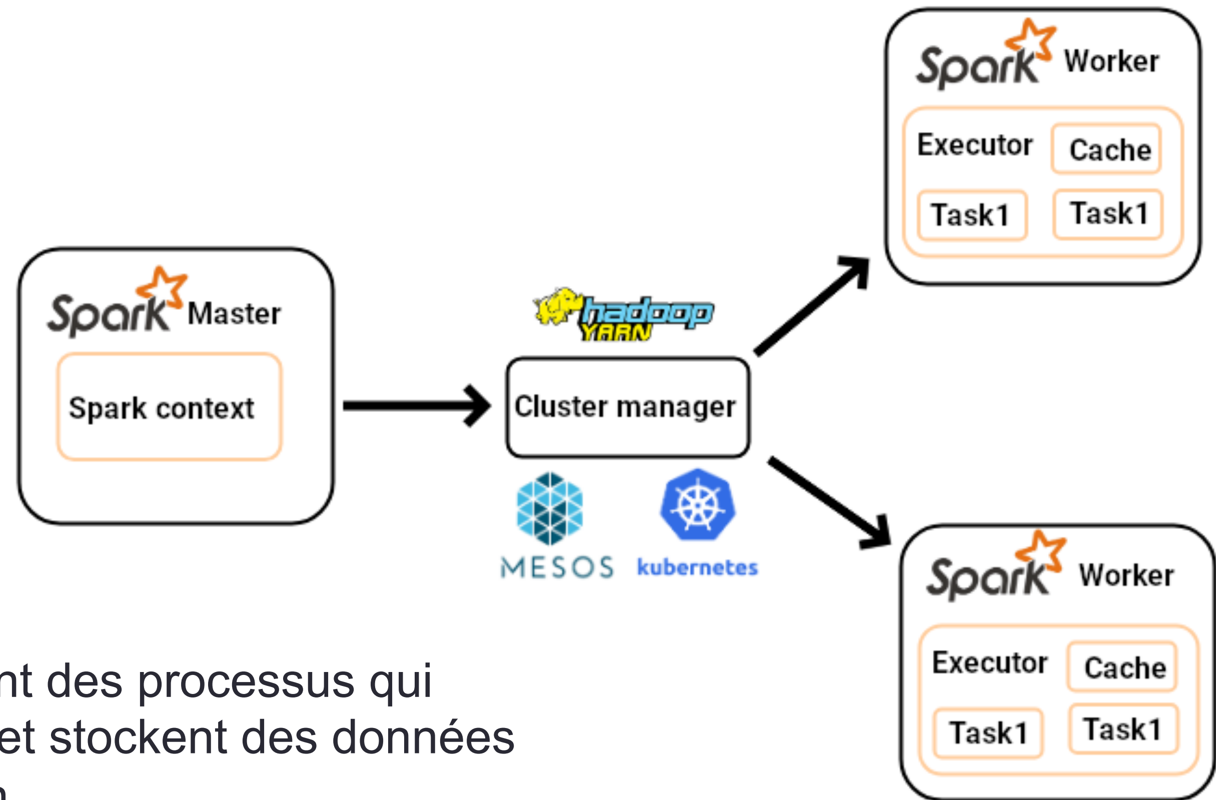
Architecture Spark



1. Les applications Spark s'exécutent de manière indépendante sur un cluster, coordonné par l'objet *SparkContext* dans votre programme principal.
2. *SparkContext* se connecte au *Cluster Manager* qui alloue des ressources entre les applications. Une fois connecté, Spark obtient des « executors » sur les nœuds du cluster.
3. Les « executors » sont des processus qui réalisent des calculs et stockent des données pour votre application.

http://spark.apache.org/docs/latest/cluster-overview.html

Architecture Spark



4. Les « executors » sont des processus qui réalisent des calculs et stockent des données pour votre application.
5. Spark envoie votre code (Java JAR ou fichier Python) aux « executors ».
6. Finalement *SparkContext* envoie des tâches à exécuter aux « executors ».

http://spark.apache.org/docs/latest/cluster-overview.html

« wordcount » en Spark/Python (pyspark)

```python
import sys
from pyspark import SparkContext

if __name__ == "__main__":

    if len(sys.argv) != 2:
        print("Usage: PySpark_wc <file>", file=sys.stderr)
        sys.exit(-1)

    # Creation d'un contexte spark
    sc = SparkContext(appName="Spark Count")
    sc.setLogLevel("ERROR") # Valid log levels include: ALL, DEBUG, ERROR, FATAL,

    lines = sc.textFile(sys.argv[1])
    counts = lines.flatMap(lambda x: x.split(' ')) \
                  .map(lambda x: (x, 1)) \
                  .reduceByKey(lambda v1,v2 : v1 + v2)

    # Stockage du resultat sur HDFS
    # ne pas oublier "hadoop fs -rm -r -f sortie" entre 2 executions
    #count.saveAsTextFile("sortie")

    # Affichage
    output = counts.collect()
    for (word, count) in output:
        print("%s: %i" % (word, count))

    # Arret du contexte Spark
    sc.stop()
```

PySpark_wc.py

```
>> spark-submit --deploy-mode client --master local[2] PySpark_wc.py input/dracula
>> hadoop fs -ls sortie
>> hadoop fs -text sortie/part-00000
>> hadoop fs -rm -r -f sortie #prêt pour une nouvelle exécution
```

Concepts de Spark

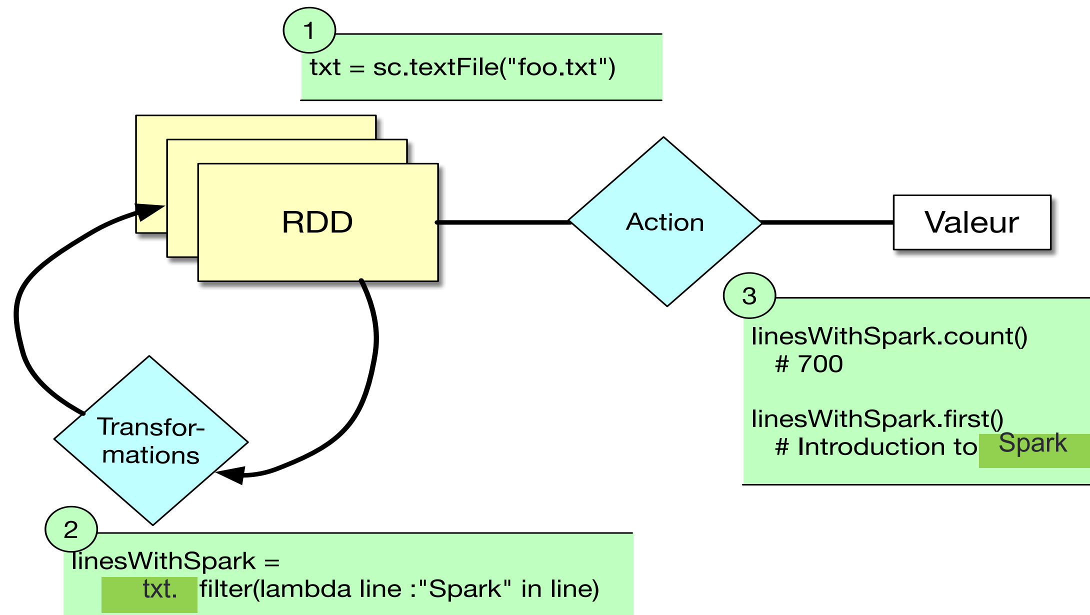
Pour comprendre Spark, trois concepts sont importants :

1. **RDD (*Resilient Distributed Data sets*) :** C'est une collection d'éléments partitionnés à travers les nœuds du cluster et qui peuvent être opérés en parallèle. C'est donc une structure de données flexible, et « *in-memory* ».
   Elle est tolérante aux pannes car un RDD sait comment recalculer son ensemble de données. Un RDD est immuable (*i.e.*, on ne peut pas le modifier) ; pour modifier son contenu, il faut créer un autre RDD. Ils supportent deux types d'opérations : les **transformations** et les **actions**.

1. **Transformations :** C'est ce que vous faite subir à un RDD pour obtenir un autre RDD. Par exemple, lire un fichier de données est une transformation qui crée un RDD. Les fonctions de transformation sont par exemple : *map*, *filter*, *flatMap*, *groupByKey*, *reduceByKey*, *aggregateByKey*, *pipe* et *coalesce*.

2. **Actions :** Ce sont les commandes qui doivent être mis en œuvre pour obtenir la réponse à la question que vous posez. Elles retournent une valeur. Par exemple, quelle est la première ligne du fichier ? Les actions sont par exemple *reduce*, *collect*, *count*, *first*, *take*, *countByKey* et *foreach*.

Comptage du nombre de lignes d'un fichier



1. La 1ière étape consiste à construire une spécification qui dit à Spark qu'il aura besoin de lire le fichier et de le stocker dans un RDD.
2. La 2ième étape consiste à préciser que l'on ne s'intéresse qu'aux lignes du fichier contenant le mot « Spark », grâce à la commande « filter ». Le résultat de cette transformation est un autre RDD.
3. A ce stade, aucun traitement n'est encore réalisé. Ils ne seront réalisés que lorsqu'une action sera demandée : par exemple compter le nombre de lignes avec le mot « Spark », ou donner la première ligne du fichier qui contient « Spark ». Cela sera réalisé de manière optimisée sur le cluster Hadoop.

RDD - Comptage du nombre de caractères d'un fichier

Resilient Distributed Datasets (RDD)

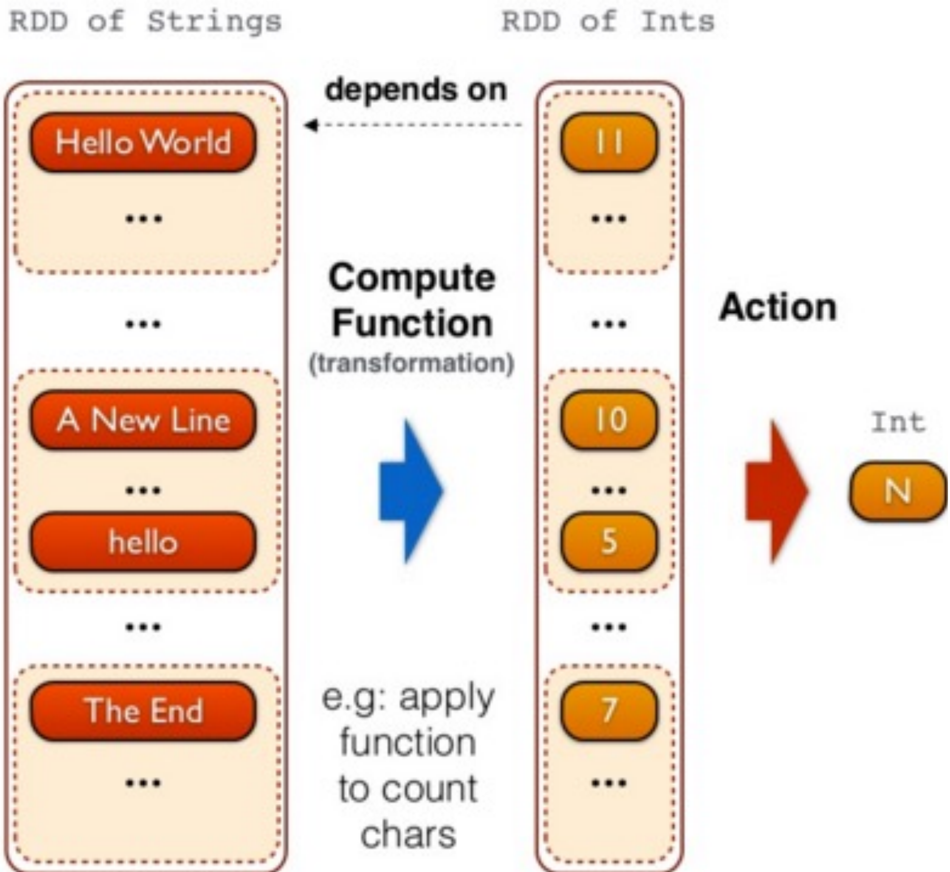
Immutable **Collection** of Objects

Partitioned and **Distributed**

Stored in **Memory**

Partitions **Recomputed on Failure**

RDD - Comptage du nombre de caractères d'un fichier



PYSPARK LIBRARY

Transformations et Actions

Collections parallélisées

```
from pyspark import SparkContext

if __name__ == "__main__":
        # Creation d’un contexte Spark
        sc=SparkContext(appName="parallelisation")

        data=[1,2,3,4,5]
        distData=sc.parallelize(data)

        # Arret du contexte Spark
        sc.stop()
```

PySpark_exemple1.py

Une fois créé, le RDD `distData` peut être opéré en parallèle.

Jeu de données externe : les fichiers de texte

```
from pyspark import SparkContext

if __name__=="__main__":

        sc=SparkContext(appName="Spark Count")

        lines=sc.textFile("README.md")
        lineLengths=lines.map(lambda s: len(s))
        # lineLengths.persist()
        totalLength=lineLengths.reduce(lambda a, b: a+b)
        print(totalLength)

        sc.stop()
```

PySpark_exemple2.py

- Le principe de « paresse » : rien n'est fait avant une action (*e.g. reduce(…)*). Spark décompose le calcul en tâches pour être exécutées sur des machines séparées. Chaque machine exécute sa « part de map() » et réalise une réduction locale, renvoyant ensuite le résultat au « *driver program* ».

- Sauvegarde en mémoire Si on a besoin de `lineLengths` plus tard → `persist()`

- Remarque `sc.textFile` peut lire plusieurs fichiers lorsqu'on lui transmet un répertoire : `textFile("/mydirectory")`, `textFile("/mydirectory/*.txt")`, `textFile("/mydirectory/*.gz")`

Jeu de données externe : les fichiers de texte

Mise à part les fichiers "texte" (*i.e. textFile(…)*), l'API Python de Spark gère également les trois formats suivants :

- *sc.wholeTextFiles(…)* permet de lire un répertoire contenant plusieurs petits fichiers et retourne chacun d'entre eux comme un tuple (*NomDuFichier*, *ContenuDuFichier*). Contrairement à *textFile(…)* qui lui retourne les lignes de tous les fichiers.

- Format *Pickle : RDD.saveAsPickleFile()* et *sc.pickleFile()* permettent d'enregistrer et de lire un RDD au format pickle de Python (cf pickle – python object serialization).

- *Sequence Files* :

```
>>> rdd = sc.parallelize(range(1, 4)).map(lambda x: (x, "a" * x))
>>> rdd.saveAsSequenceFile("path/to/file")
>>> sorted(sc.sequenceFile("path/to/file").collect())
    [(1, 'a'), (2, 'aa'), (3, 'aaa')]
```

Imprimer les éléments d'un RDD

Si on souhaite imprimer à l'écran les éléments d'un RDD, on aura tendance à écrire :

rdd.foreach(println) ou *rdd.map(println)*

Cela fonctionne sur une simple machine mais pas sur un cluster (l'impression se fait sur chaque nœud)!

Pour imprimer tous les éléments sur le *Driver*, on doit utiliser la méthode *collect(…)* d'abord qui rapporte toutes les données vers le *Driver :*

rdd.collect().foreach(println)

Cela peut saturer la mémoire du *Driver car collect()* rapporte toutes les données sur une machine unique !

Si vous n'avez besoin que de quelques éléments, pensez à utiliser

rdd.take(100).foreach(println)

Transformations 1/5

map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).

```
sc=SparkContext(appName="Spark flatMap example")

A=sc.parallelize([2,3,4]).flatMap(lambda x:[x,x,x]).collect()
# =>[2, 2, 2, 3, 3, 3, 4, 4, 4]

B=sc.parallelize([1,2,3]).map(lambda x:[x,x,x]).collect()
# =>[[1, 1, 1], [2, 2, 2], [3, 3, 3]]

sc.stop()
```

PySpark_exemple3.py

[Guide détaillé : transformations]

Transformations 2/5

sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numPartitions*]))	Return a new dataset that contains the distinct elements of the source dataset.

```
one=sc.parallelize(range(1,10))
two=sc.parallelize(range(5,15))
one.persist()
two.persist()
U=one.intersection(two).collect()
# =>[5, 6, 7, 8, 9]

D=one.union(two).distinct().collect()
# =>[8, 12, 4, 1, 13, 5, 9, 2, 14, 10, 6, 11, 3, 7]
```

PySpark_exemple4.py

[Guide détaillé : transformations]

Transformations 3/5 - (clé, valeur)

groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance.
reduceByKey(*func*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.

```
>> head -5 baby_names_2013.csv
        Year,First Name,County,Sex,Count
        2013,GAVIN,ST LAWRENCE,M,9
        2013,LEVI,ST LAWRENCE,M,9
        2013,LOGAN,NEW YORK,M,44
        2013,HUDSON,NEW YORK,M,49
```

[Guide détaillé : transformations]

Transformations 3/5

PySpark_exemple5.py

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-

import sys
from pyspark import SparkContext

if __name__ == "__main__":

    sc=SparkContext(appName="baby example")
    sc.setLogLevel("ERROR") # Valid log levels include: ALL, DEBUG, ERROR, FATAL, INFO, OFF, TRA

    baby_names = sc.textFile("file:///root/pyspark/baby_names_2013.csv")    # en local
    #baby_names = sc.textFile("hdfs:///user/root/input/baby_names_2013.csv")  # sur HDFS

    rows = baby_names.filter(lambda x: 'County' not in x).map(lambda line: line.split(","))
    rows.persist()

    namesToCounties = rows.map(lambda n: (str(n[1]), str(n[2]))).groupByKey()
    res1 = namesToCounties.map(lambda x : {x[0]: list(x[1])}).collect()
    print(res1[:10])
    # => [{'GRIFFIN': ['ERIE', 'ONONDAGA', 'NEW YORK', 'ERIE', ...}, {...}]

    namesNumber = rows.map(lambda n: (str(n[1]), int(n[4]) ) ).reduceByKey(lambda v1,v2: v1 + v2).collect()
    print(namesNumber[:10])
    # => [('GRIFFIN', 20), ('KALEB', 24), ('JOHNNY', 25), ('NAYELI', 11), ('ERIN', 58) ...

    sc.stop()
```

Remarquez le truc pour enlever la ligne de titre :

```
baby_names.filter(lambda line:"County" not in line)
```

Transformations 4/5

sortByKey([*ascending*], [*numPartitions*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.
join(*otherDataset*, [*numPartitions*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.

```
names1=sc.parallelize(("abe","abby","apple")).map(lambda a: (a,1))
names2=sc.parallelize(("apple","beatty","beatrice")).map(lambda a: (a,1))

fulljoin=names1.join(names2).collect()
# =>[('apple', (1, 1))]

leftjoin=names1.leftOuterJoin(names2).collect()
# =>[('abe', (1, None)), ('apple', (1, 1)), ('abby', (1, None))]

rightjoin=names1.rightOuterJoin(names2).collect()
# =>[('apple', (1, 1)), ('beatrice', (None, 1)), ('beatty', (None, 1))]
```

PySpark_exemple6.py

Guide détaillé : transformations

Transformations 5/5

cogroup(*otherDataset*, [*numPartitions*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called groupWith.
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(*command*, *[envVars]*)	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(*numPartitions*)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
repartition(*numPartitions*)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.

[Guide détaillé : transformations]

Actions 1/2

reduce(*func*)	Aggregate the elements of the dataset using a function *func* (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(*n*)	Return an array with the first *n* elements of the dataset.
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.

Guide détaillé : actions

Actions 2/2

saveAsTextFile(*path*)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(*path*) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. **Note**: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See Understanding closures for more details.

Guide détaillé : actions

Approximation de PI en Python

```python
#!/usr/bin/env python3
#-*- coding: utf-8 -*-
import sys
from operator import add
from random import random

from pyspark import SparkContext

def f(_):
        x = random() * 2 - 1
        y = random() * 2 - 1
        return 1 if x ** 2 + y ** 2 < 1 else 0

if __name__ == "__main__":
    """
        Usage: pi
    """

    sc = SparkContext(appName="PySpark_Pi")
    sc.setLogLevel("ERROR")

    n = 100000
    count = sc.parallelize(range(1, n + 1)).map(f).reduce(add)
    print ("Pi is roughly ", 4.0 * count / n)

    sc.stop()
```

PySpark_Pi.py

```
>> spark-submit --deploy-mode client --master local[2] pi.py
```

Variable partagé (shared)

Normalement, lorsqu'une fonction, passée à une opération Spark (telle que *mapper* ou *reducer*), est exécutée sur un nœud de cluster distant, elle fonctionne sur des copies distinctes de toutes les variables utilisées dans la fonction.
Ces variables sont copiées sur chaque ordinateur et aucune mise à jour des variables sur l'ordinateur distant n'est propagée au programme pilote. La prise en charge de variables générales partagées en lecture-écriture entre les tâches serait inefficace.
Cependant, Spark fournit deux types limités de variables partagées pour deux modèles d'utilisation courants : **les variables de diffusion (*broadcast*) et les accumulateurs (*accumulator*).**

Broadcast variables : Les variables de diffusion permettent au programmeur de conserver une variable en lecture seule en cache sur chaque machine plutôt que d'en envoyer une copie. Ils peuvent être utilisés, par exemple, pour donner à chaque nœud une copie d'un grand ensemble de données d'entrée de manière efficace:

```
broadcastVar = sc.broadcast([1, 2, 3])
broadcastVar.value
[1, 2, 3]
```

Variable partagé (shared)

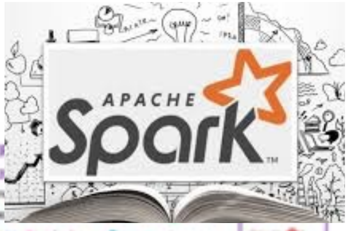
Accumulators: Les accumulateurs sont des variables qui sont uniquement « ajoutées » via une opération associative et commutative et peuvent donc être efficacement prises en charge en parallèle.
Ils peuvent être utilisés pour implémenter des compteurs (comme dans MapReduce) ou des sommes. Spark prend en charge nativement les accumulateurs de types numériques et les programmeurs peuvent ajouter la prise en charge de nouveaux types.

Exemple:

```
accum = sc.accumulator(0)
accum
    Accumulator<id=0, value=0>
sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))
accum.value
10
```

ECOSYSTÈME DE SPARK

Big Data landscape



DATA & AI LANDSCAPE 2020

INFRASTRUCTURE

ANALYTICS & MACHINE INTELLIGENCE

APPLICATIONS – ENTERPRISE

APPLICATIONS – INDUSTRY

OPEN SOURCE

DATA SOURCES & APIs

DATA RESOURCES

Version 1.0 - September 2020

© Matt Turck (@mattturck) & FirstMark (@firstmarkcap)

mattturck.com/data2020

FIRSTMARK

EARLY STAGE VENTURE CAPITAL

Environments

Applications

Data Sources

[Source]

Spark ecosystème

Data science and Machine learning

SQL analytics and BI

Storage and Infrastructure

Cf *third-party project*

Spark ecosystème

- **Spark QL**
  - Pour le traitement de données SQL et non structurées
- **MLlib**
  - Algorithmes de Machine Learning
- **GraphX**
  - Traitement de grands graphes
- **Spark streaming**
  - Traitement de flux de données « on-line »