

# TECHNO DU BIG-DATA

Stéphane Derrode, Dpt MI -  
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 programming*) 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 elt d'un iterable pour en obtenir un nouveau.
- *filter()*: appliquer une fonction pour filtrer chaque elt d'un iterable et en obtenir un nouveau.
- *reduce()*: appliquer une fonction aux deux premiers éléments d'un iterable, 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)

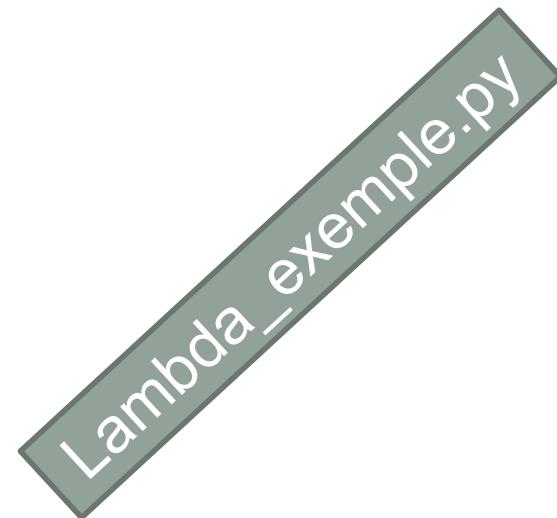
Python permet d'écrire des mini-fonctions, pas plus longue qu'une ligne, à la volée, sans nécessairement leur donner de nom. Il s'agit des *lambda functions*:

```
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))
```



**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]
```

## 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]
```

map\_exemples.py

# 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.

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.

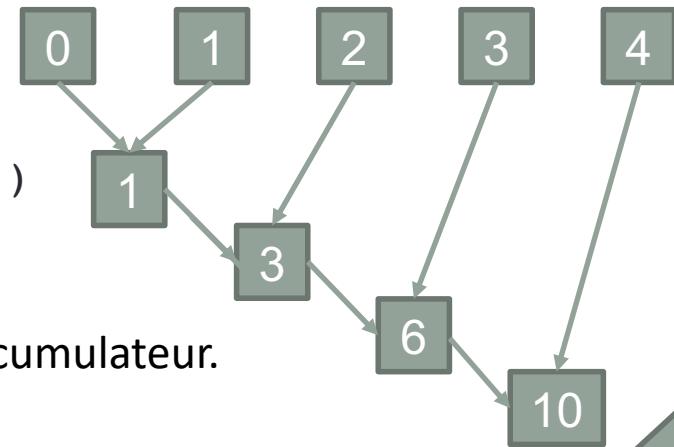
## Exemple 1

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

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

## Exemple 2

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



reduce\_exemples.py

# 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 des objets de type RDD (*Resilient Distributed Data sets*).

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())
```



# SPARK

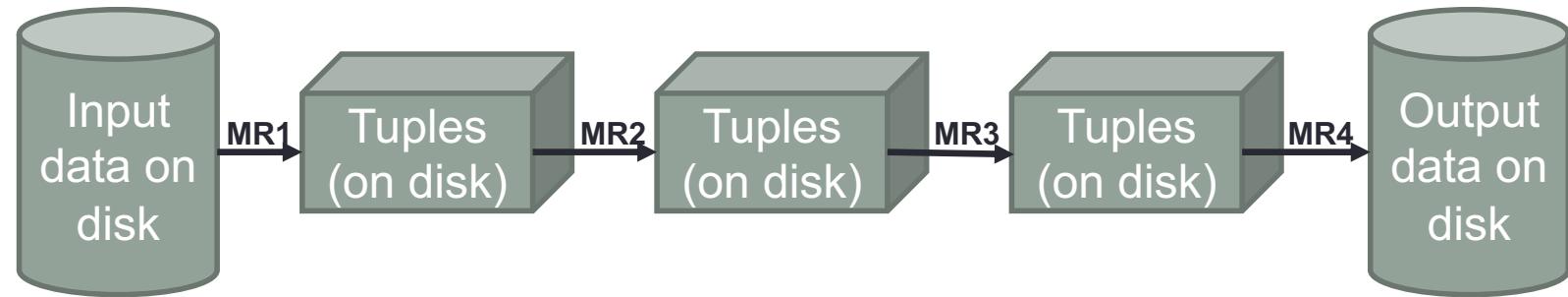
---

Principes et fonctionnement

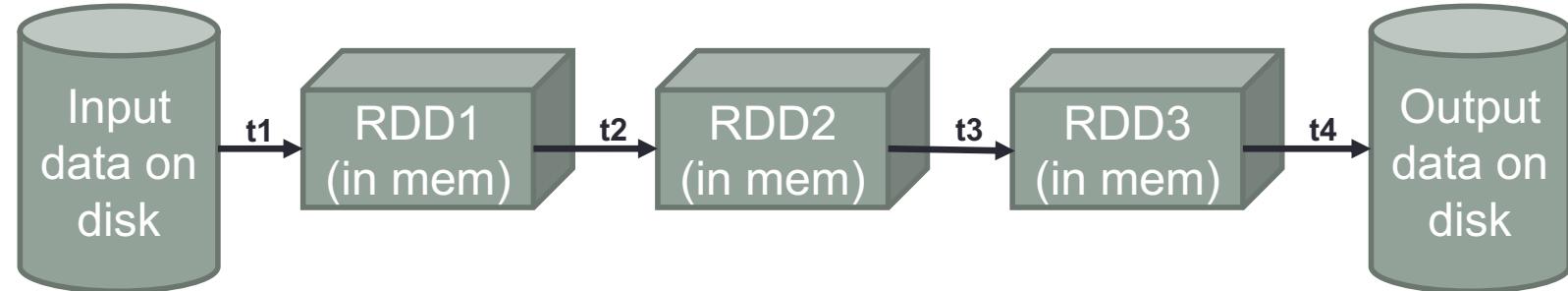
# Pourquoi Spark?

- La plupart des algorithmes de Machine Learning sont itératifs : chaque itération améliore le résultat.
- Avec des méthodes orientées Disque Dur (DD), 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

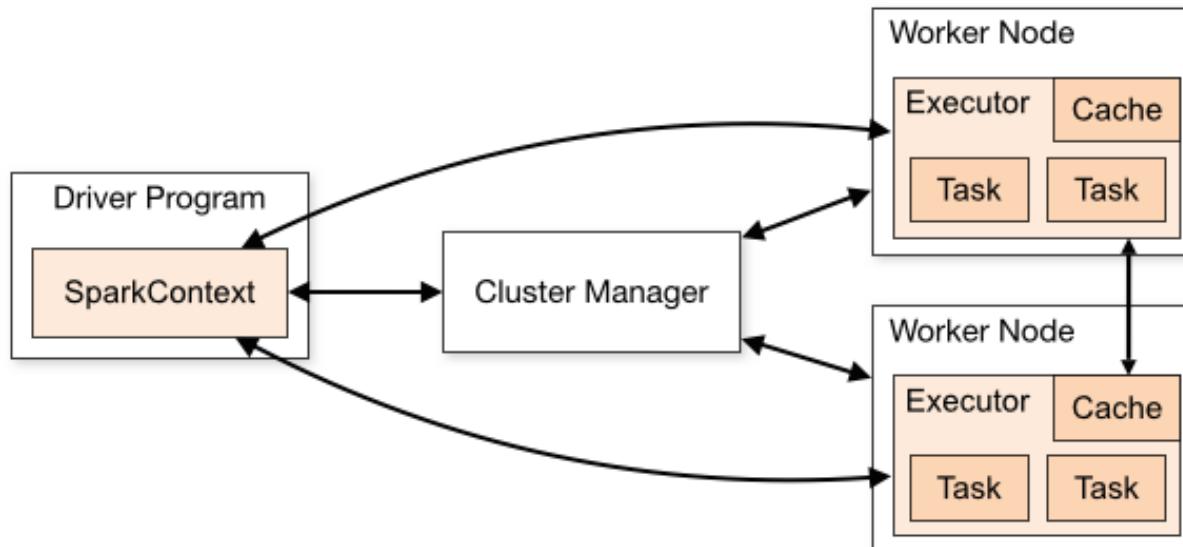
	Data Size	Time	Nodes	Cores
Hadoop MR (2013)	102.5 TB	72 min	2,100	50,400 physical
Apache Spark (2014)	100 TB	23 min	206	6,592 virtualized

# Pourquoi Spark ?

- Commence à Berkeley en 2009. Devenu un projet de la fondation Apache depuis 2013.
- <http://spark.apache.org> « *Apache Spark is a fast and general engine for large-scale data processing* »
- 10x (sur DD) – 100x (en mémoire) plus rapide
- Fournit des API pour Java, Scala, Python, R.
- 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

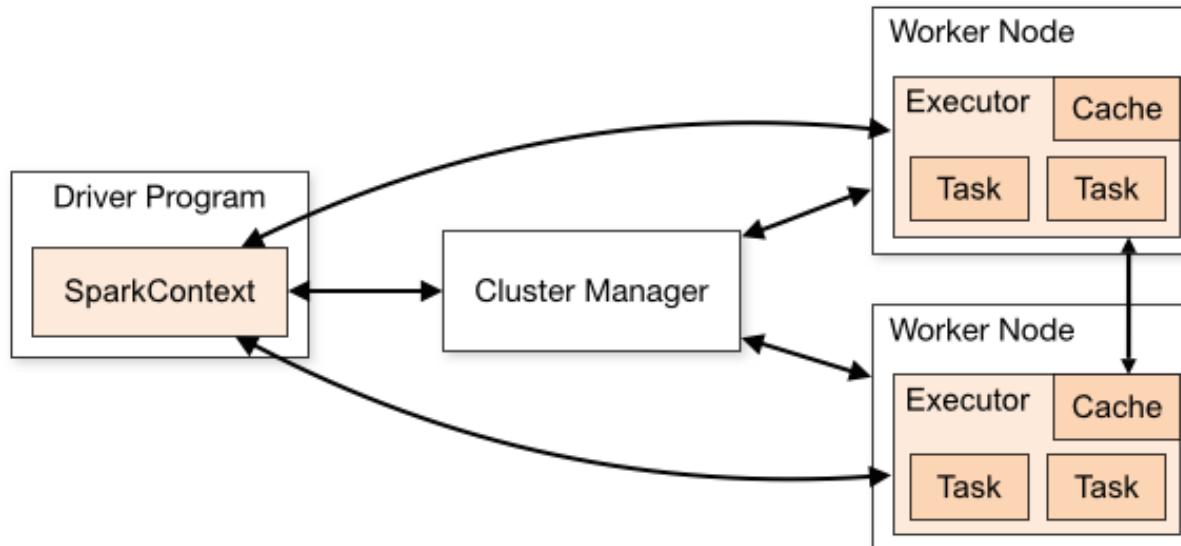
# Fonctionnement de Spark

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.



# Fonctionnement de 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 ».



# « wordcount » en Spark/Python (pyspark)

```
if __name__ == "__main__":  
  
    if len(sys.argv) != 2:  
        print("Usage: wordcount <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,  
  
    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 "hdfs dfs -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\_wc1.py

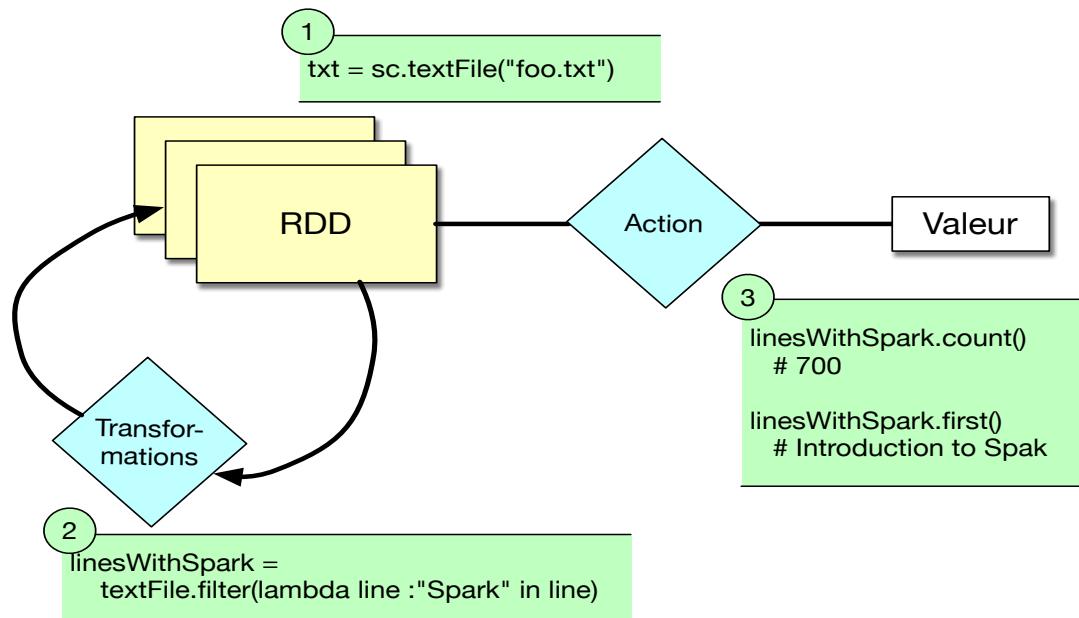
```
>> spark-submit --master local[2] PySpark_wc.py input/dracula  
>> hadoop fs -ls sortie  
>> hadoop fs -text sortie/part-00000  
>> hadoop fs -rm -r -f sortie # prêt pour une nouvelle exécution
```

# Concepts de Spark

Pour comprendre Spark, trois concepts sont importants :

1. **RDD (*Resilient Distributed Data sets*)** : C'est une 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 le modifier, il faut créer un autre RDD. Ils supportent deux types d'opérations : les **transformations** et les **actions**.
2. **Transformations** : C'est ce que vous faites 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*.
3. **Actions** : Ce sont les commandes qui doivent être mises 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 1<sup>ière</sup> 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 2<sup>ième</sup> 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.

# Comptage du nombre de lignes d'un fichier

## Resilient Distributed Datasets (RDD)

RDD of Strings



Immutable **Collection** of Objects

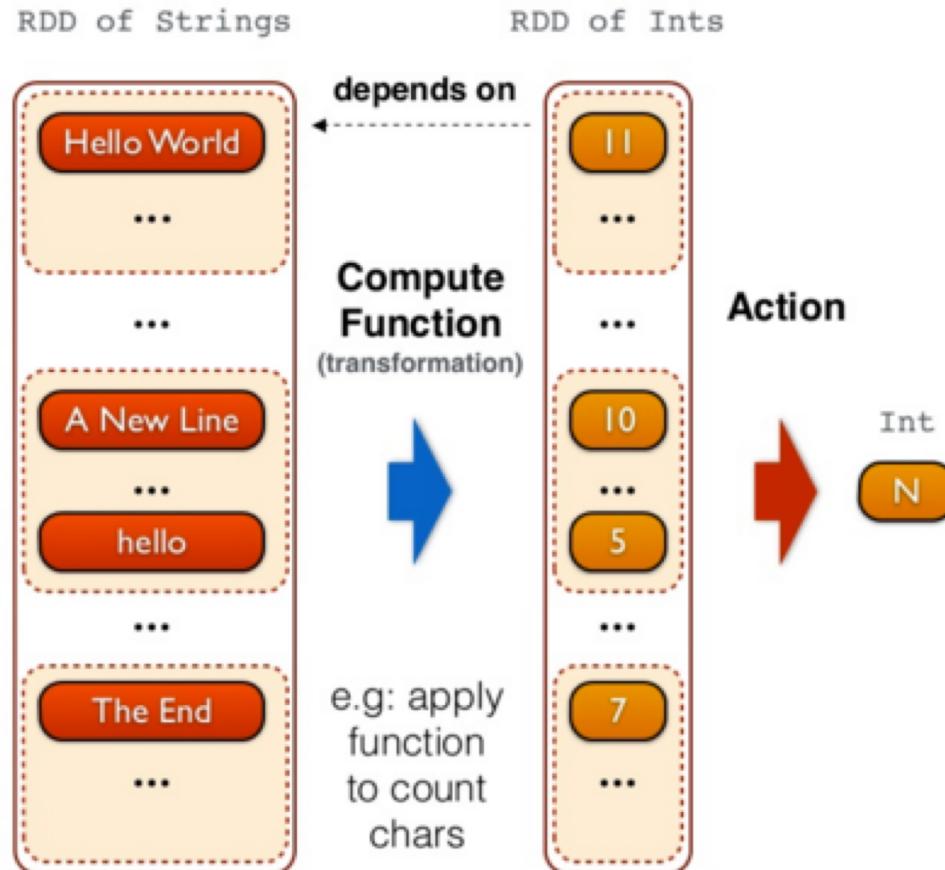
**Partitioned** and **Distributed**

Stored in **Memory**

Partitions **Recomputed on Failure**

# Comptage du nombre de lignes d'un fichier

## RDD Transformations and Actions



# Comptage du nombre de lignes d'un fichier

## Spark API

Scala

```
val spark = new SparkContext()

val lines      = spark.textFile("hdfs://docs/")      // RDD[String]
val nonEmpty = lines.filter(l => l.nonEmpty())      // RDD[String]

val count = nonEmpty.count
```

Java 8

```
SparkContext spark = new SparkContext();

JavaRDD<String> lines      = spark.textFile("hdfs://docs/")
JavaRDD<String> nonEmpty = lines.filter(l -> l.length() > 0);

long count = nonEmpty.count();
```

Python

```
spark = SparkContext()

lines = spark.textFile("hdfs://docs/")
nonEmpty = lines.filter(lambda line: len(line) > 0)

count = nonEmpty.count()
```

# 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. A ce stade, 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* ». Si on a besoin de lineLengths plus tard → persist() (sauvegarde en mémoire après le premier calcul).

# Common transformations 1/4

---

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

---

```
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]]
```

PySpark\_exemple3.py

# Common transformations 2/4

---

<b>union(<i>otherDataset</i>)</b>	Return a new dataset that contains the union of the elements in the source dataset and the argument.
<b>intersection(<i>otherDataset</i>)</b>	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
<b>distinct([<i>numTasks</i>])</b>	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

# Common transformations 3/4 - (clé, valeur)

---

<b>groupByKey([numTasks])</b>	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
<b>reduceByKey(func, [numTasks])</b>	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 <i>func</i> , which must be of type (V,V) => V.

**Baby\_names\_2013.csv** (from <https://www.healthdata.gov/dataset/baby-names-beginning-2007>)

```
>> 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
```

# Common transformations 3/4

```
baby_names=sc.textFile("baby_names_2013.csv")
rows=baby_names.map(lambda line: line.split(","))
rows.persist()

namesToC=rows.map(lambda n: (str(n[1]), str(n[2]))).groupByKey()
namesToC.map(lambda x : {x[0]: list(x[1])}).collect()
#print namesToC.take(20)
#=>[{'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
#=>[('GRIFFIN', 20), ('KALEB', 24), ('JOHNNY', 25), ('NAYELI', 11), ('ERIN', 58) ...]
```

**Remarque :** truc pour enlever la ligne de titre du fichier :

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

PySpark\_exemple5.py

# Common transformations 4/4

**join(otherDataset , [numTasks])** 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

# Common actions 1/1

<b>reduce(func)</b>	Aggregate the elements of the dataset using a function <i>func</i> (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
<b>collect()</b>	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.
<b>count()</b>	Return the number of elements in the dataset.
<b>first()</b>	Return the first element of the dataset (similar to take(1)).
<b>take(n)</b>	Return an array with the first <i>n</i> elements of the dataset.
<b>takeSample(<i>withReplacement</i>, <i>num</i>, [<i>seed</i>])</b>	Return an array with a random sample of <i>num</i> elts of the dataset, with or without replacement, optionally pre-specifying a random seed.
<b>takeOrdered(<i>n</i>, [<i>ordering</i>])</b>	Return the first <i>n</i> elements of the RDD using either their natural order or a custom comparator.
<b>saveAsTextFile(<i>path</i>)</b>	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call <code>toString</code> on each element.
<b>countByKey()</b>	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.

# Approximation de PI en Python

```
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 x**2+y**2<1

if __name__ == "__main__":
    sc=SparkContext(appName="PythonPi")
    partitions=int(sys.argv[1]) if len(sys.argv)>1 else 2
    n=100000*partitions

    count=sc.parallelize(range(1, n+1), partitions)
        .map(f)
        .reduce(add)

    print("Pi is roughly ", 4.0*count/n)
    sc.stop()
```

```
>> spark-submit --master local[2] pi.py 10
```

PySpark\_Pi.py

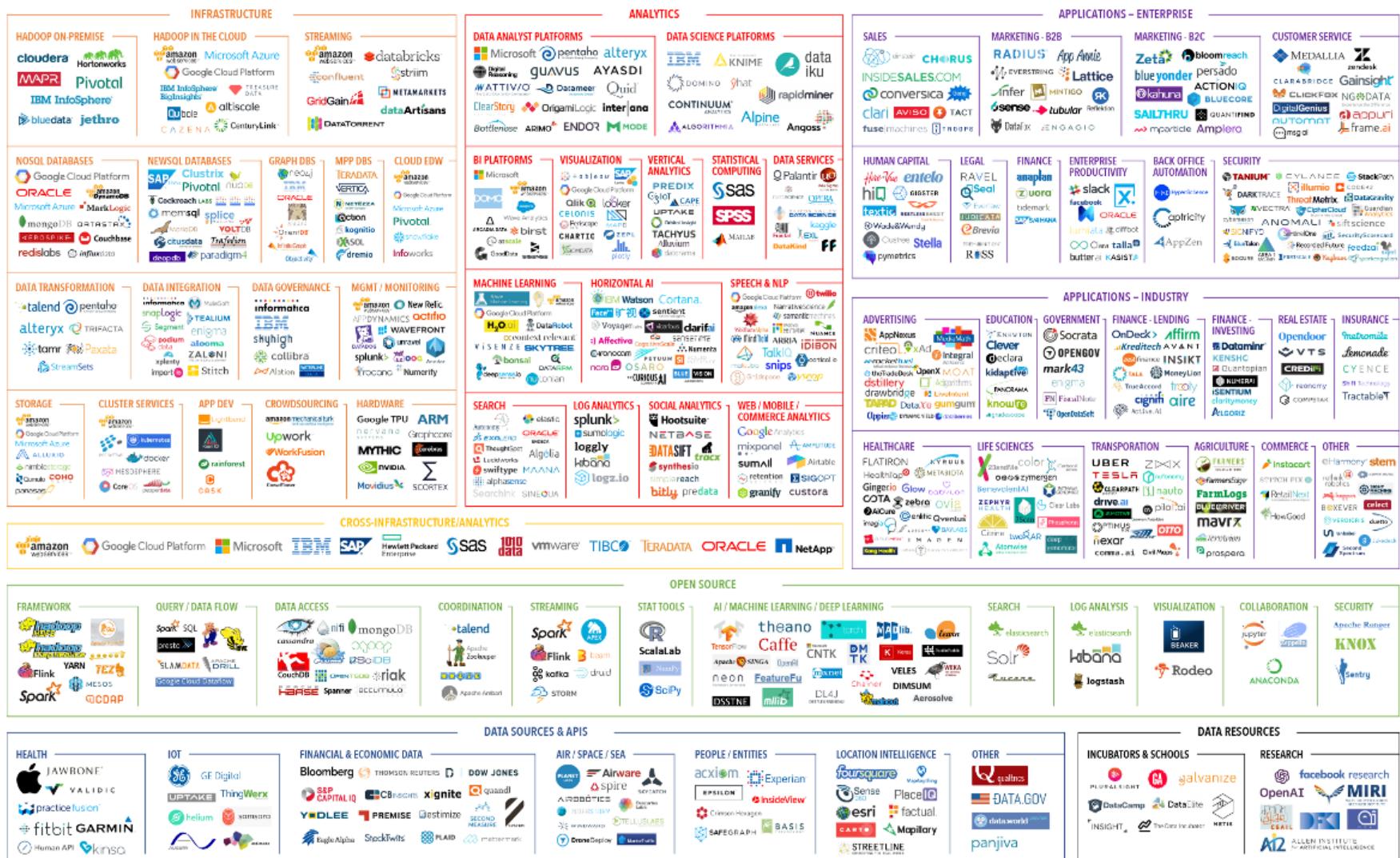
# ECOSYSTÈME DE SPARK

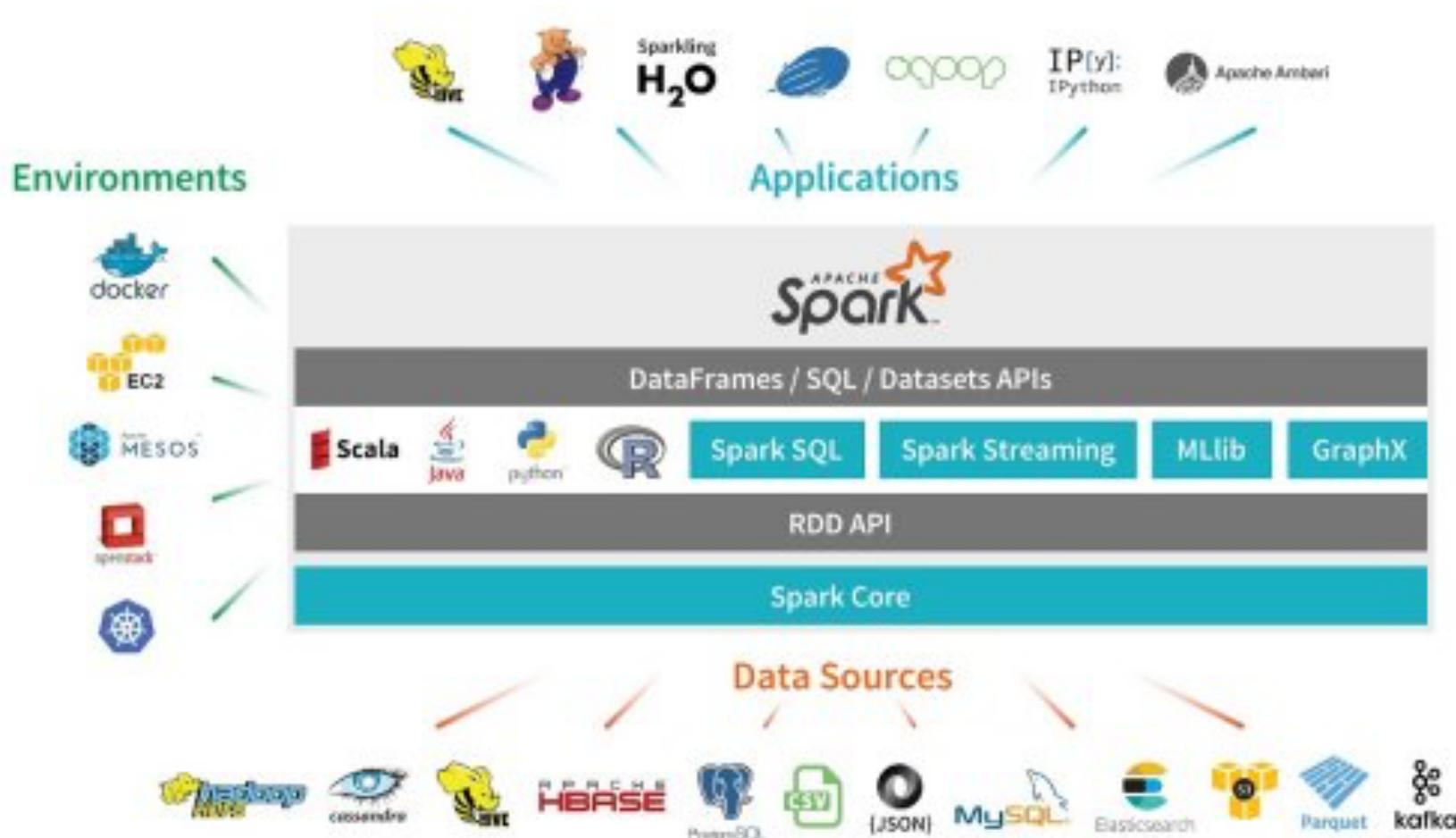
---

# Big Data landscape



## BIG DATA LANDSCAPE 2017





# Spark écosystème (en un coup d'œil)

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

