

TP

M1 - Traitement par Lot avec Spark

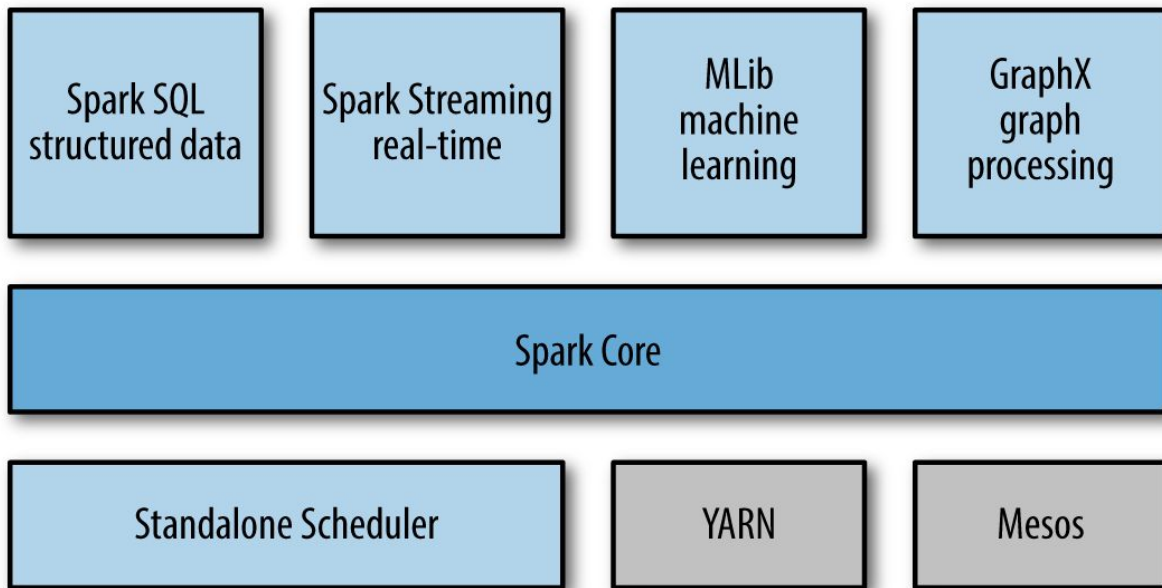
Nous allons utiliser la machine virtuelle cloudera précédemment installée dans ce TP.

Spark

Présentation

[Spark](#) est un système de traitement rapide et parallèle. Il fournit des APIs de haut niveau en Java, Scala, Python et R, et un moteur optimisé qui supporte l'exécution des graphes. Il supporte également un ensemble d'outils de haut niveau tels que [Spark SQL](#) pour le support du traitement de données structurées, [MLlib](#) pour l'apprentissage des données, [GraphX](#) pour le traitement des graphes, et [Spark Streaming](#) pour le traitement des données en streaming.





Spark et Hadoop

Spark peut s'exécuter sur plusieurs plateformes: Hadoop, Mesos, en standalone ou sur le cloud. Il peut également accéder diverses sources de données, comme HDFS, Cassandra, HBase et S3.

Dans ce TP, nous allons exécuter Spark sur Hadoop YARN. YARN s'occupera ainsi de la gestion des ressources pour le déclenchement et l'exécution des Jobs Spark.

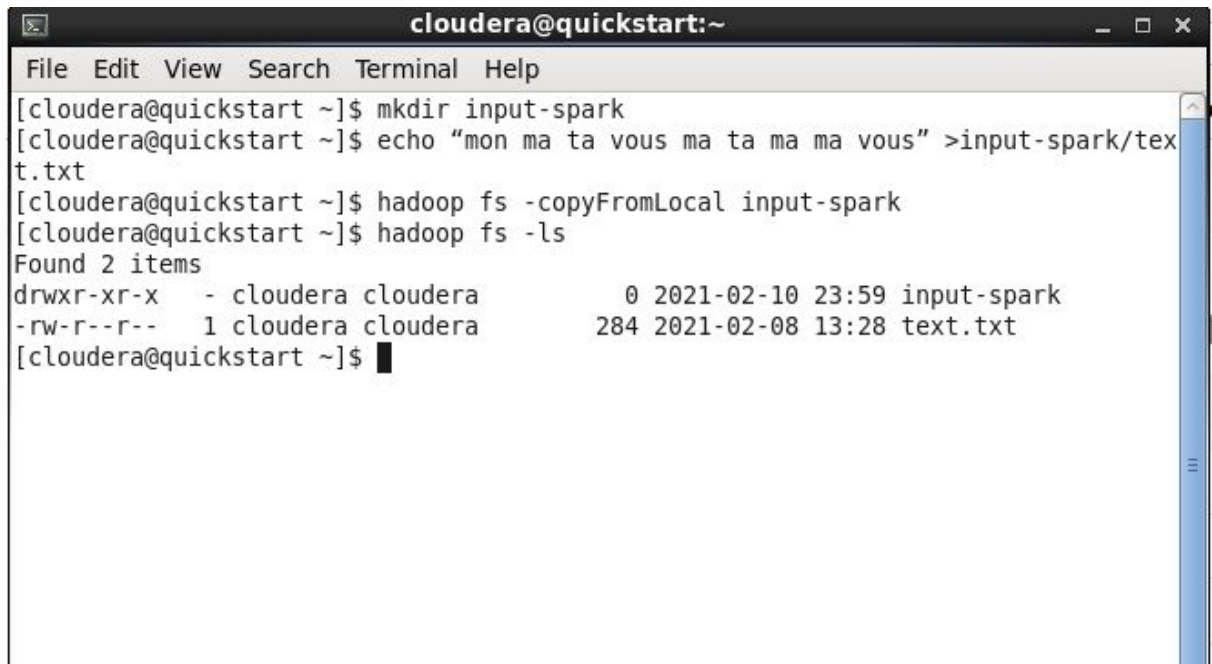
1- Ouvrir *un terminal* (Terminal 1)

- Saisir la commande *spark-shell*

```
cloudera@quickstart:~  
File Edit View Search Terminal Help  
SLF4J: Found binding in [jar:file:/usr/lib/parquet/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]  
SLF4J: Found binding in [jar:file:/usr/lib/avro/avro-tools-1.7.6-cdh5.12.0.jar!/org/slf4j/impl/StaticLoggerBinder.class]  
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.  
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]  
Welcome to  
 version 1.6.0  
Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_67)  
Type in expressions to have them evaluated.  
Type :help for more information.  
21/02/10 23:55:04 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable  
Spark context available as sc (master = local[*], app id = local-1613030113588).  
21/02/10 23:55:26 WARN shortcircuit.DomainSocketFactory: The short-circuit local reads feature cannot be used because libhadoop cannot be loaded.  
SQL context available as sqlContext.  
scala>
```

2- Créer le fichier (Ouvrir un autre terminal : Terminal 2)

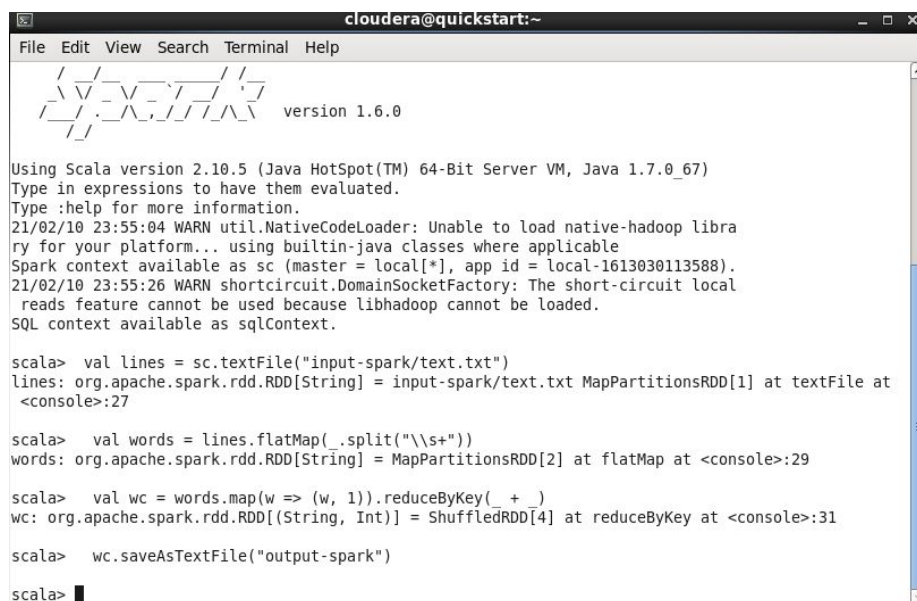
- `mkdir input-spark`
- `echo "mon ma ta vous ma ta ma ma vous" >input-spark/text.txt`
- `hadoop fs -copyFromLocal input-spark`



```
cloudera@quickstart:~  
File Edit View Search Terminal Help  
[cloudera@quickstart ~]$ mkdir input-spark  
[cloudera@quickstart ~]$ echo "mon ma ta vous ma ta ma ma vous" >input-spark/text.txt  
[cloudera@quickstart ~]$ hadoop fs -copyFromLocal input-spark  
[cloudera@quickstart ~]$ hadoop fs -ls  
Found 2 items  
drwxr-xr-x - cloudera cloudera 0 2021-02-10 23:59 input-spark  
-rw-r--r-- 1 cloudera cloudera 284 2021-02-08 13:28 text.txt  
[cloudera@quickstart ~]$
```

3- Sur le terminal 1

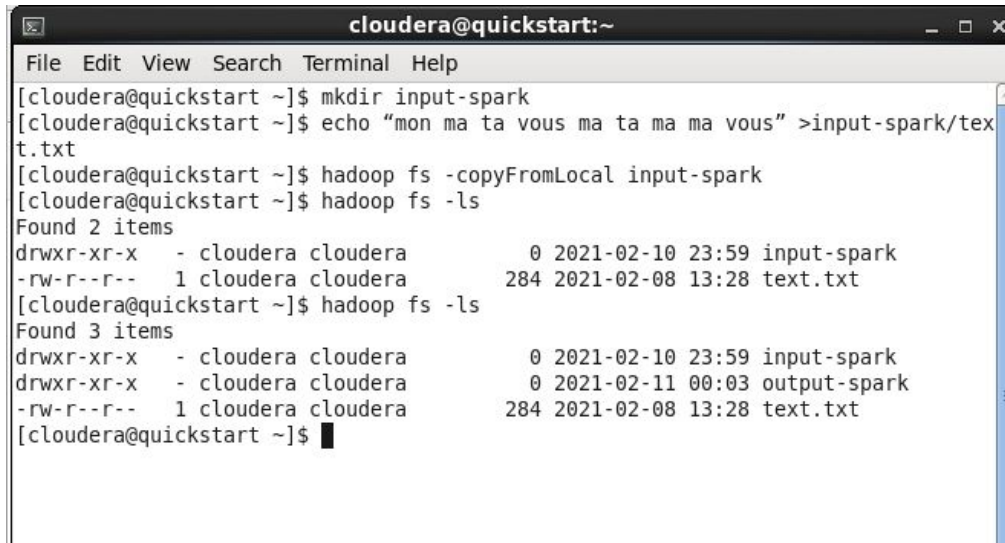
```
val lines = sc.textFile("input-spark/text.txt")  
val words = lines.flatMap(_.split("\\s+"))  
val wc = words.map(w => (w, 1)).reduceByKey(_ + _)  
wc.saveAsTextFile("output-spark")
```



```
cloudera@quickstart:~  
File Edit View Search Terminal Help  
version 1.6.0  
Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_67)  
Type in expressions to have them evaluated.  
Type :help for more information.  
21/02/10 23:55:04 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable  
Spark context available as sc (master = local[*], app id = local-1613030113588).  
21/02/10 23:55:26 WARN shortcircuit.DomainSocketFactory: The short-circuit local reads feature cannot be used because libhadoop cannot be loaded.  
SQL context available as sqlContext.  
  
scala> val lines = sc.textFile("input-spark/text.txt")  
lines: org.apache.spark.rdd.RDD[String] = input-spark/text.txt MapPartitionsRDD[1] at textFile at <console>:27  
  
scala> val words = lines.flatMap(_.split("\\s+"))  
words: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at flatMap at <console>:29  
  
scala> val wc = words.map(w => (w, 1)).reduceByKey(_ + _)  
wc: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:31  
  
scala> wc.saveAsTextFile("output-spark")  
  
scala>
```

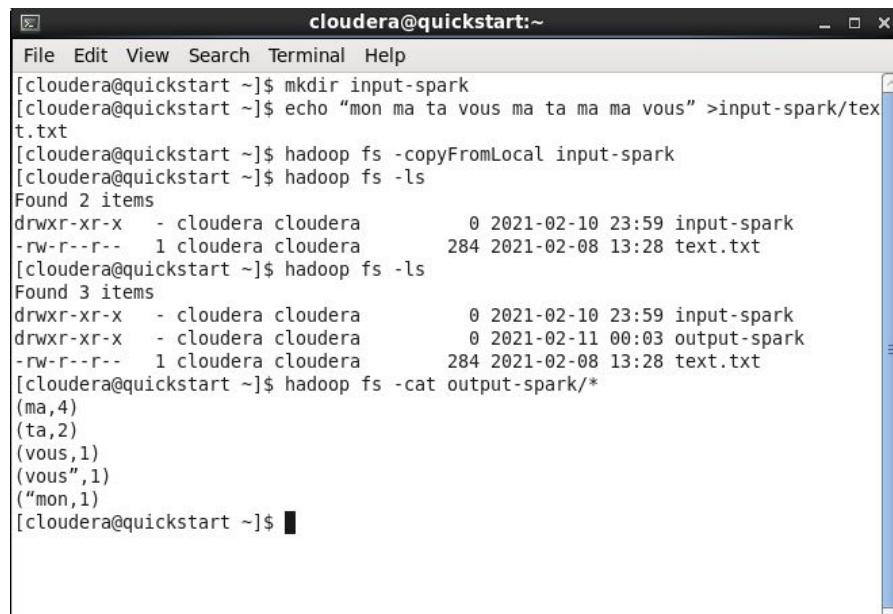
4- Vérification du résultat sur le terminal 2

- *hadoop fs -ls*



```
cloudera@quickstart:~  
File Edit View Search Terminal Help  
[cloudera@quickstart ~]$ mkdir input-spark  
[cloudera@quickstart ~]$ echo "mon ma ta vous ma ta ma ma vous" >input-spark/text  
t.txt  
[cloudera@quickstart ~]$ hadoop fs -copyFromLocal input-spark  
[cloudera@quickstart ~]$ hadoop fs -ls  
Found 2 items  
drwxr-xr-x - cloudera cloudera 0 2021-02-10 23:59 input-spark  
-rw-r--r-- 1 cloudera cloudera 284 2021-02-08 13:28 text.txt  
[cloudera@quickstart ~]$ hadoop fs -ls  
Found 3 items  
drwxr-xr-x - cloudera cloudera 0 2021-02-10 23:59 input-spark  
drwxr-xr-x - cloudera cloudera 0 2021-02-11 00:03 output-spark  
-rw-r--r-- 1 cloudera cloudera 284 2021-02-08 13:28 text.txt  
[cloudera@quickstart ~]$
```

- *hadoop fs cat output-spark/**



```
cloudera@quickstart:~  
File Edit View Search Terminal Help  
[cloudera@quickstart ~]$ mkdir input-spark  
[cloudera@quickstart ~]$ echo "mon ma ta vous ma ta ma ma vous" >input-spark/text  
t.txt  
[cloudera@quickstart ~]$ hadoop fs -copyFromLocal input-spark  
[cloudera@quickstart ~]$ hadoop fs -ls  
Found 2 items  
drwxr-xr-x - cloudera cloudera 0 2021-02-10 23:59 input-spark  
-rw-r--r-- 1 cloudera cloudera 284 2021-02-08 13:28 text.txt  
[cloudera@quickstart ~]$ hadoop fs -ls  
Found 3 items  
drwxr-xr-x - cloudera cloudera 0 2021-02-10 23:59 input-spark  
drwxr-xr-x - cloudera cloudera 0 2021-02-11 00:03 output-spark  
-rw-r--r-- 1 cloudera cloudera 284 2021-02-08 13:28 text.txt  
[cloudera@quickstart ~]$ hadoop fs -cat output-spark/*  
(ma,4)  
(ta,2)  
(vous,1)  
(vous",1)  
("mon,1)  
[cloudera@quickstart ~]$
```

5- A la découverte de spark

L'API de Spark

A un haut niveau d'abstraction, chaque application Spark consiste en un programme *driver* qui exécute la fonction *main* de l'utilisateur et lance plusieurs opérations parallèles sur le cluster. L'abstraction principale fournie par Spark est un RDD (*Resilient Distributed Dataset*), qui représente une collection d'éléments partitionnés à travers les noeuds du cluster, et sur lesquelles on peut opérer en parallèle. Les RDDs sont créés à partir d'un fichier dans HDFS par exemple, puis le transforment. Les utilisateurs peuvent demander à Spark de sauvegarder un RDD en mémoire, lui permettant ainsi d'être réutilisé efficacement à travers plusieurs opérations parallèles.



Les RDDs supportent deux types d'opérations:

- les *transformations*, qui permettent de créer un nouveau Dataset à partir d'un Dataset existant (<https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations>)
- les *actions*, qui retournent une valeur au programme *driver* après avoir exécuté un calcul sur le Dataset. (<https://spark.apache.org/docs/latest/rdd-programming-guide.html#actions>)

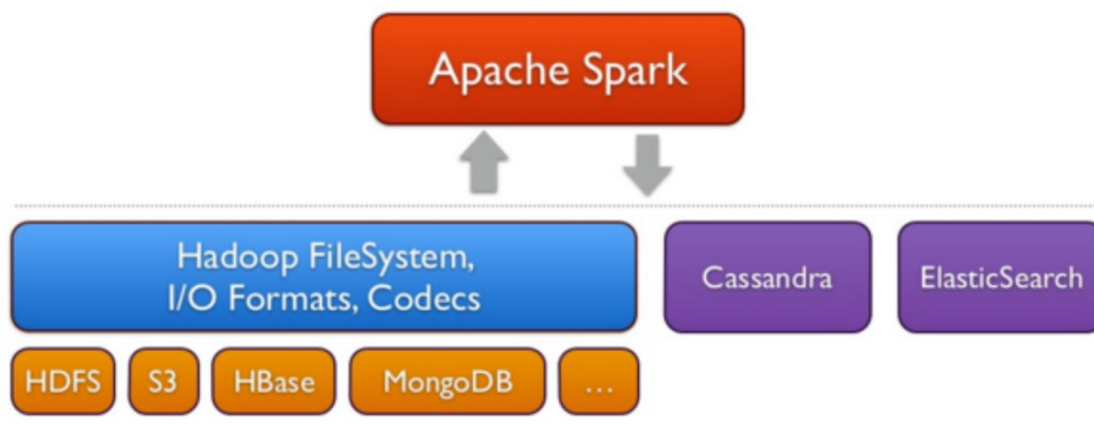
Par exemple, un *map* est une transformation qui passe chaque élément du dataset via une fonction, et retourne un nouvel RDD représentant les résultats. Un *reduce* est une action qui agrège tous les éléments du RDD en utilisant une certaine fonction et retourne le résultat final au programme.

Toutes les transformations dans Spark sont *lazy*, car elles ne calculent pas le résultat immédiatement. Elles se souviennent des transformations appliquées à un dataset de base (par ex. un fichier). Les transformations ne sont calculées que quand une action nécessite qu'un résultat soit retourné au programme principal. Cela permet à Spark de s'exécuter plus efficacement.

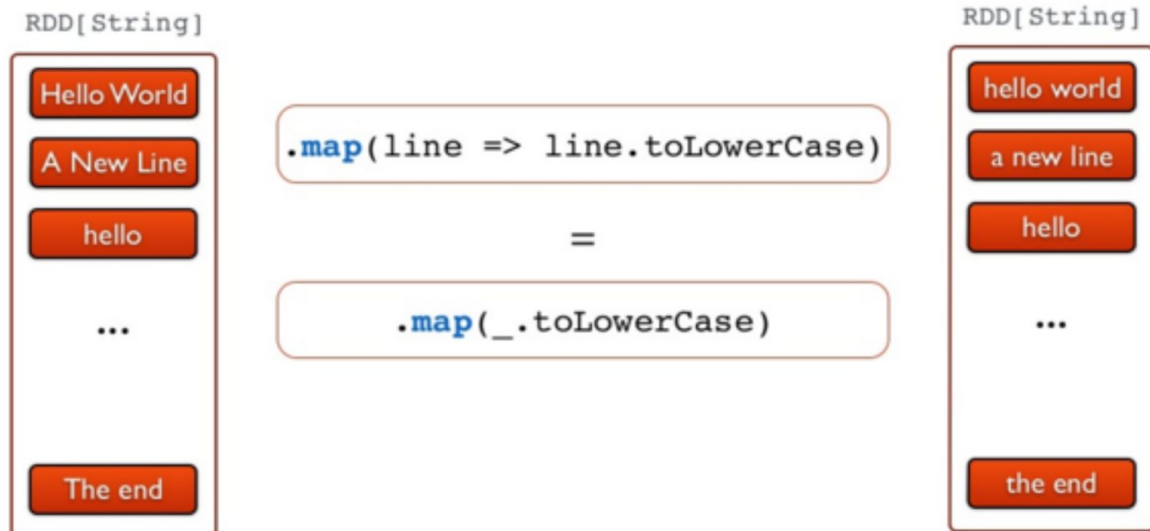
Exemple

L'exemple que nous allons présenter ici par étapes permet de relever les mots les plus fréquents dans un fichier. Pour cela, le code suivant est utilisé:

```
//Etape 1 - Créer un RDD à partir d'un fichier texte de Hadoop  
val docs = sc.textFile("input-spark/text.txt")
```

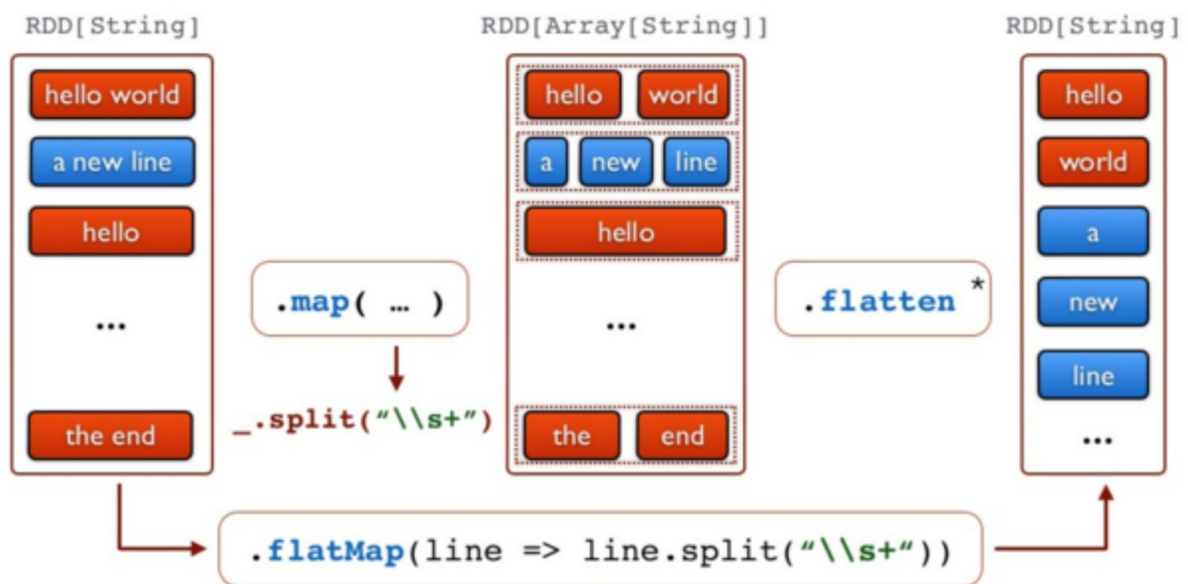


```
//Etape 2 - Convertir les lignes en minuscule  
val lower = docs.map(line => line.toLowerCase)
```



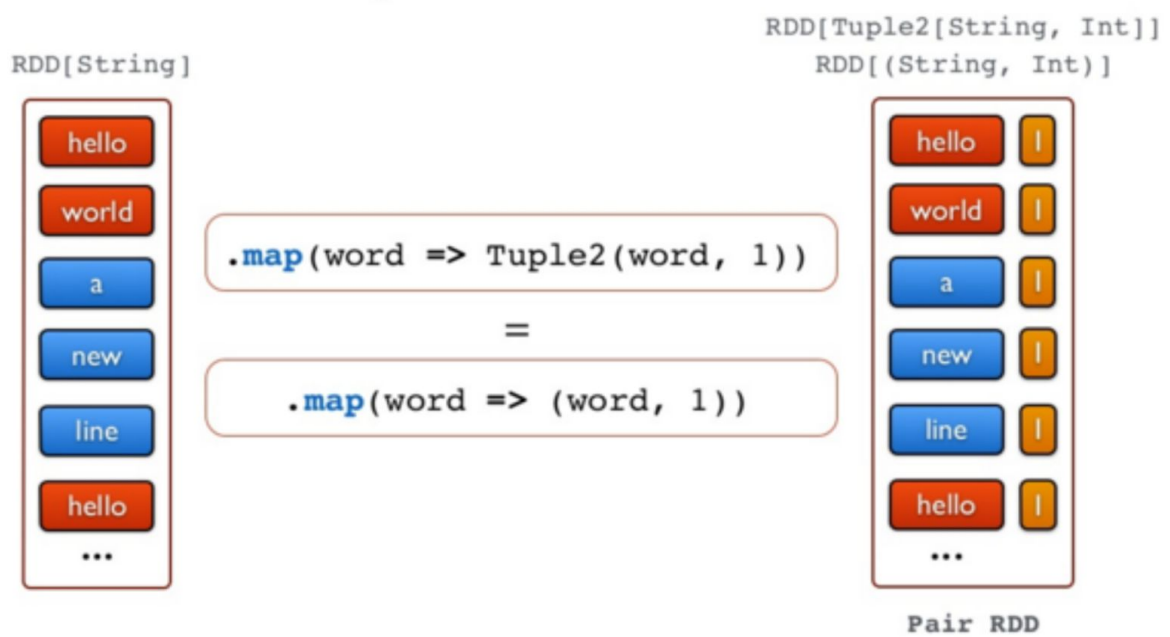
//Etape 3 - Séparer les lignes en mots

```
val words = lower.flatMap(line => line.split("\\s+"))
```

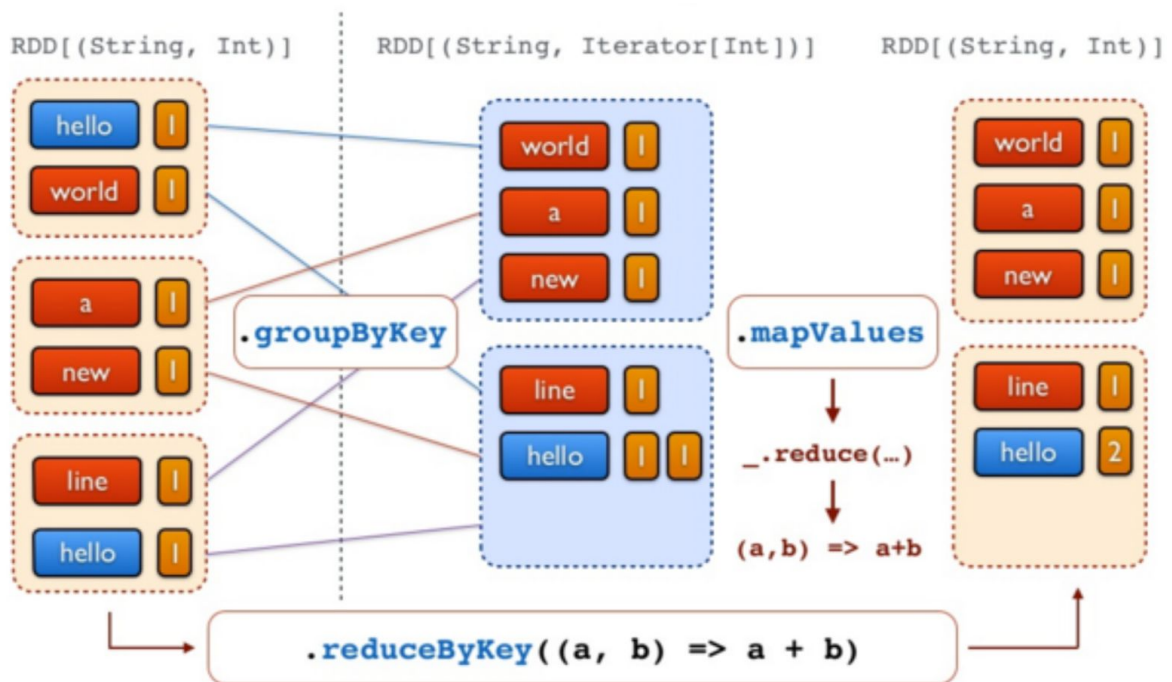


//Etape 4 - produire les tuples (mot, 1)

```
val counts = words.map(word => (word,1))
```

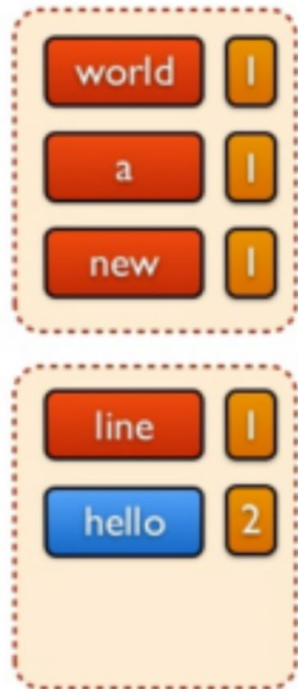



```
//Etape 5 - Compter tous les mots
val freq = counts.reduceByKey(_ + _)
```



```
//Etape 6 - Inverser les tuples (transformation avec swap)
freq.map(_._swap)
```


RDD[(String, Int)]



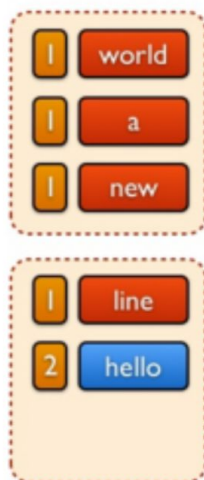
RDD[(Int, String)]



`.map(_._swap)`

//Etape 6 - Inverser les tuples (action de sélection des n premiers)
Remplacer N par 2 si vous voulez voir les 2 premiers elements
`val top = freq.map(_._swap).top(2)`

RDD[(Int, String)]



local top N *

`.top(N)`

local top N *



reduction

Array[(Int, String)]



NB. Ce TP est extrait de ce site web <https://insatunisia.github.io/TP-BigData/tp2/> .

6- Spark submit

Pour lancer un job spark, vous pouvez utiliser spark-submit

Nous allons exécuter le programme word count spark écrit en python.

Vous trouvez le programme sur le repo git : <https://github.com/elomedah/iris-2020> dans le dossier *Spark-word-count-python*

- `spark-submit chemin_de_votre_programme [arguments_du_programme]`

Exemple :

Vous pouvez télécharger le programme :

wget <https://raw.githubusercontent.com/elomedah/iris-2020/main/Spark-word-count-python/wordcounter.py>

Si le programme se trouve dans votre répertoire courant

- `spark-submit wordcounter.py input-spark/text.txt`