

Introduction aux Big Data

Chapitre 4 : Apache Spark

Pr. Dounia ZAIDOUNI

Filière: DATA, S3, INE2
Institut National des Postes et Télécommunications (INPT)

31 Octobre 2024

Présentation du syllabus du cours

- Chapitre 1: Introduction aux Big Data
- Chapitre 2: Solutions et Architectures de Big Data
- Chapitre 3: Apache Hadoop
- **Chapitre 4 : Apache Spark**
 - Présentation Spark
 - Architecture Spark
 - Concepts de base de Spark
 - Tolérance aux pannes dans Spark
 - Implémentation et exécution des applications sous Spark

Outline

- 1 Présentation Spark
- 2 Architecture Spark
- 3 Concepts de base de Spark
- 4 Tolérance aux pannes dans Spark
- 5 Implémentation et exécution des applications sous Spark

Plan

- 1 Présentation Spark
- 2 Architecture Spark
- 3 Concepts de base de Spark
- 4 Tolérance aux pannes dans Spark
- 5 Implémentation et exécution des applications sous Spark

Historique

- En 2009, Spark fut conçu par Matei Zaharia lors de son doctorat au sein de l'université de Californie à Berkeley.
 - À l'origine, son développement est une solution pour accélérer le traitement des systèmes Hadoop. Les développeurs mettent notamment en avant la rapidité du produit en termes d'exécution des tâches par rapport à MapReduce.
- En 2013, transmis à la fondation Apache, Spark devient l'un des projets les plus actifs de cette dernière.

Historique

- En 2014, Spark a gagné le Daytona GraySort Contest dont l'objectif est de trier 100 To de données le plus rapidement possible.
 - Ce record était préalablement détenu par Hadoop. Pour ce faire, Spark a utilisé 206 machines obtenant un temps d'exécution final de 23 minutes alors que Hadoop avait lui utilisé 2100 machines pour un temps d'exécution final de 72 minutes.
 - La puissance de Spark fut démontrée en étant 3 fois plus rapide et en utilisant approximativement 10 fois moins de machines.
- Les contributeurs qui participent à son développement sont nombreux, et issus d'environ 200 sociétés différentes, comme Intel, Facebook, IBM et Netflix). Ainsi depuis 2015 on recense plus de 1000 contributeurs.

Présentation Spark

- Spark est un framework open source de calcul distribué. Il permet les traitements Batch et streaming.
- Il s'agit d'un ensemble d'outils et de composants logiciels structurés selon une architecture définie.
- Ce framework est un cadre applicatif de traitements big data pour effectuer des analyses complexes à grande échelle.
- Spark fournit des APIs de haut niveau en **Java, Scala, Python et R**, et un moteur optimisé qui supporte l'exécution des **graphes**.



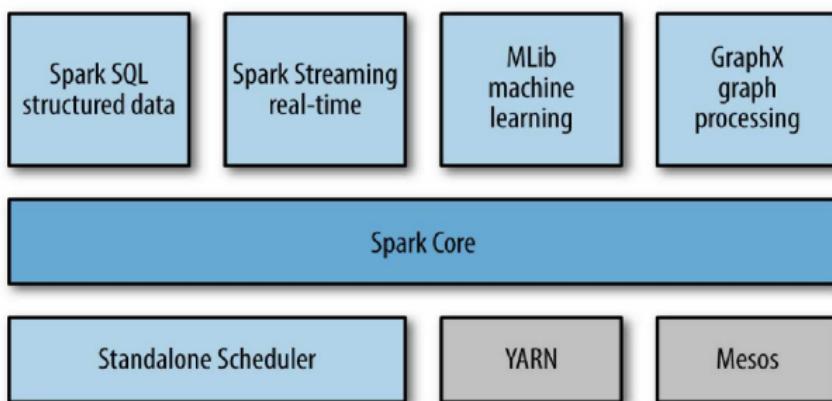
Présentation Spark

- Il supporte également un ensemble d'outils de haut niveau tels que :
 - Spark SQL** : pour le support du traitement de données structurées,
 - <https://spark.apache.org/docs/latest/sql-programming-guide.html>
 - Mlib** : librairie de machine learning pour l'apprentissage des données,
 - <https://spark.apache.org/docs/latest/ml-guide.html>
 - GraphX** : pour le traitement des graphes,
 - <https://spark.apache.org/docs/latest/graphx-programming-guide.html>
 - Spark Streaming** : pour le traitement des données en streaming.
<https://spark.apache.org/docs/latest/streaming-programming-guide.html>
 - Spark Core** : comprend un moteur d'exécution général pour la plate-forme Spark, requis par d'autres fonctionnalités. Il fournit une mémoire intégrée et des jeux de données de référence.

Présentation Spark

- Spark peut s'exécuter sur plusieurs plateformes:
 - **Hadoop**,
 - **Mesos**,
 - **Standalone**,
 - **Cloud**.
- Il peut également accéder diverses sources de données, comme :
 - **HDFS, Cassandra, HBase et S3.**

Présentation des composants de Spark



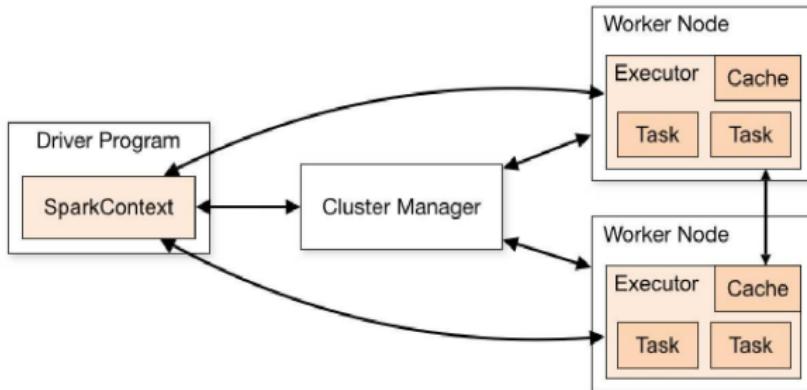
Plan

- 1 Présentation Spark
- 2 Architecture Spark
- 3 Concepts de base de Spark
- 4 Tolérance aux pannes dans Spark
- 5 Implémentation et exécution des applications sous Spark

Architecture Spark

- Les applications Spark s'exécutent en tant qu'ensembles de processus indépendants sur un cluster, coordonnés par l'objet **SparkContext** du programme principal (main).
- Pour exécuter une application sur un cluster, SparkContext peut se connecter à plusieurs types de gestionnaires de cluster (Spark **standalone** ou Mesos ou YARN), qui allouent des ressources entre les applications.
 - Une fois connecté, Spark acquiert les exécuteurs sur les nœuds du cluster, qui sont responsables des calculs et du stockage des données de l'application.
 - Ensuite, le travail est divisé en plusieurs tâches plus petites et SparkContext envoie ces tâches aux exécuteurs pour exécution.
 - Les exécuteurs exécutent les tâches attribuées par le gestionnaire du cluster et les renvoient au contexte Spark.

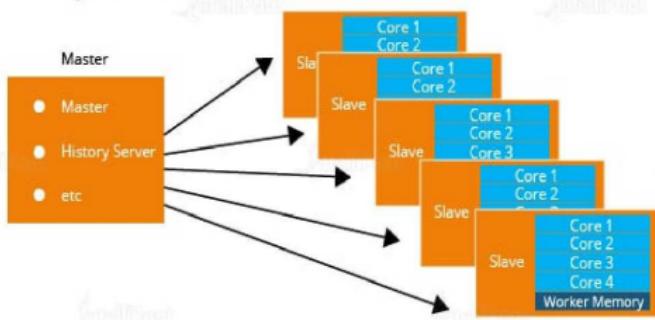
Architecture Spark



Cluster Standalone

- Le cluster Standalone est composé du **Standalone Master** et du **Standalone Worker**.
- Le standalone Master est le gestionnaire de ressources, alors que le standalone Worker est le worker du cluster.
- En mode cluster Standalone :
 - Il n'y a qu'un seul exécuteur pour exécuter les tâches sur chaque worker.
 - Un client établit une connexion avec le Master, demande des ressources et démarre le processus d'exécution sur le worker node.

Spark Standalone Architecture



Cluster Standalone

- On peut lancer un cluster standalone soit manuellement ou automatiquement :
 - Manuellement :
 - en démarrant un serveur master standalone :
`./sbin/start-master.sh`
 - et en démarrant des workers standalone et en les connectant au master:
`./sbin/start-slave.sh < master-spark-URL >`,
 - Automatiquement :
 - En utilisant les scripts de lancement fournis par Spark.
- Il est également possible d'exécuter le master et les workers sur une seule machine à des fins de test.
- Pour plus de détails sur l'installation d'un cluster Standalone, vous pouvez consulter :
 - <https://spark.apache.org/docs/latest/spark-standalone.html>

- Spark s'inspire fortement de Hadoop. Il fait partie de son écosystème et il privilégie le système de fichiers de stockage HDFS ou S3.
- Spark intègre son propre cluster manager SPARK standalone qui est interchangeable avec YARN ou MESOS.
- Les améliorations apportées par Spark :
 - Une évolution améliorée du moteur MapReduce,
 - Spark est un cluster dit in-memory (utilisant la mémoire pour plus de rapidité),
 - C'est un optimiseur paresseux et intelligent,
 - Développé en SCALA :
 - 100 lignes de code JAVA peut être traduit en une dizaine de lignes en SCALA,
 - Hérite des avantages de la programmation fonctionnelle.
- Le développement d'applications parallélisées est plus facile avec spark:
 - Concept de Resilient Distributed Dataset (RDD).

Plan

- 1 Présentation Spark
- 2 Architecture Spark
- 3 Concepts de base de Spark
- 4 Tolérance aux pannes dans Spark
- 5 Implémentation et exécution des applications sous Spark

Utilisation de Spark

Il existe deux manières pour utiliser Spark :

- Spark Shell :

- Le terminal spark est un terminal interactif pour apprendre ou explorer des données.
- Utilisation du Shell python ou du Shell Scala.

- Spark application :

- Pour le traitement des données à large échelle,
- Applications en Python, Scala ou Java,
- Utilisation de la commande : Spark-submit.

Spark Shell

Le terminal Spark est un terminal interactif pour apprendre ou explorer des données.

- Le terminal pour Spark en utilisant python :
 - Shell python: \$./bin/pyspark
- Le terminal pour Spark en utilisant Scala :
 - Shell Scala: \$./bin/spark-shell
 - Documentation language Scala :
<http://www.scala-lang.org/api/current/#package>
 - Tutoriel language Scala :
<https://docs.scala-lang.org/tutorials/>

Pour voir les nouveautés de Spark 3.5 :

- <https://spark.apache.org/releases/spark-release-3-5-0.html>

SparkContext

- Le SparkContext est la couche d'abstraction qui permet à Spark de savoir où il va s'exécuter.
- Un SparkContext standard sans paramètres correspond à l'exécution en local sur 1 CPU du code Spark qui va l'utiliser :

```
import sys
from pyspark import SparkContext

if __name__ == "__main__":
    # Create SparkContext
    sc = SparkContext()
```

- En Java pour plus de compatibilité, et même si on pourrait utiliser un SparkContext standard, on va privilégier l'utilisation d'un **JavaSparkContext** plus adapté pour manipuler des types Java.

```
public class SparkContextExample {
    public static void main(String[] args) {
        JavaSparkContext sc = new JavaSparkContext();
        JavaRDD<String> lines = sc.textFile("/path/to/file/*.txt");
    }
}
```

Le shell PySpark

- Dans le shell PySpark, le contexte SparkContext est déjà créé, dans la variable appelée sc.

```
hduser@zaidouni-VirtualBox:/usr/local/spark$ ./bin/pyspark
Python 3.10.4 (main, Jun 29 2022, 12:14:53) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
22/10/23 22:31:52 WARN Utils: Your hostname, zaidouni-VirtualBox resolves to a
loopback address: 127.0.1.1; using 10.0.2.15 instead (on interface enp0s3)
22/10/23 22:31:52 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another
address
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLev
el(newLevel).
22/10/23 22:31:54 WARN NativeCodeLoader: Unable to load native-hadoop library f
or your platform... using builtin-java classes where applicable
Welcome to

    \ _/ \
     \  \_ \ /_ / \_ / \
      \_ / . _\ \_ / / / \_ \
        /_ / / / / / / / \
                           version 3.3.0

Using Python version 3.10.4 (main, Jun 29 2022 12:14:53)
Spark context Web UI available at http://10.0.2.15:4040
Spark context available as 'sc' (master = local[*], app id = local-166655711557
0).
SparkSession available as 'spark'.
>>> █
```

Le shell PySpark

- Vous pouvez définir le maître auquel le contexte se connecte à l'aide de l'argument **- -master**.
- Vous pouvez ajouter des fichiers JAR au classpath en passant une liste séparée par des virgules à l'argument du **- -jars**.
- Vous pouvez ajouter des dépendances (par exemple, Spark Packages) à votre session shell en fournissant une liste de coordonnées Maven séparées par des virgules à l'argument **- -packages**.
- Tous les référentiels supplémentaires dans lesquels des dépendances pourraient exister peuvent être passés à l'argument **- -repositories**.
- Par exemple, pour exécuter `./bin/pyspark` sur exactement quatre cœurs, utilisez :
 - `./bin/pyspark - -master local[4]`
- Pour une liste complète des options, exédez : **pyspark - -help**
<https://spark.apache.org/docs/latest/rdd-programming-guide.html#using-the-shell>

Passage des fonctions

- Exemple de passage des fonctions nommées (en python) :
> **def toUpper(s):**
 return s.upper()
> **mydata = sc.textFile("file1.txt")**
> **mydata.map(toUpper).take(2)**
- Exemple de passage des fonctions anonymes (en python) :
> **mydata.map(lambda line: line.upper()).take(2)**
- Exemple de passage des fonctions en java :

Language: Java 8

```
...
JavaRDD<String> lines = sc.textFile("file");
JavaRDD<String> lines_uc = lines.map(
    line -> line.toUpperCase());
...
```

Collections parallélisées

- Les collections parallélisées sont créées en appelant la méthode de **parallelize** de SparkContext sur une collection ou une collection existante dans votre programme main.
- Les éléments de la collection sont copiés pour former un dataset distribué pouvant être utilisé en parallèle.
- Par exemple, voici comment créer une collection parallélisée contenant les numéros 1 à 5 :

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

- Une fois créé, le dataset distribué (distData) peut être utilisé en parallèle.

Collections parallélisées

- Par exemple, nous pouvons appeler :
distData.reduce (lambda a, b: a + b) pour additionner les éléments de la liste.
- Un paramètre important pour les collections parallélisées est le nombre de **partitions** dans lesquelles on veut couper les données. Spark exécutera une tâche pour chaque partition du cluster.
- En règle générale, vous disposez de 2 à 4 partitions pour chaque processeur de votre cluster.
- Normalement, Spark essaie de définir automatiquement le nombre de partitions en fonction de votre cluster. Cependant, vous pouvez également le définir manuellement en le passant comme second paramètre à la fonction parallelize.
 - Par exemple : **sc.parallelize (data, 10)**.

- PySpark peut créer des datasets distribués à partir de toute source de stockage prise en charge par Hadoop, y compris votre système de **fichiers local, HDFS, Cassandra, HBase, Amazon S3, etc.**
- Spark prend en charge les **fichiers texte**, les **SequenceFiles** et tout autre fichier **Hadoop InputFormat**.
 - Par exemple pour charger des données de Hadoop :
`sc.hadoopFile(path/to/file).`
- Les RDD de fichier texte peuvent être créés à l'aide de la méthode **textFile** de SparkContext.
- `textFile` prend un URI pour le fichier (un chemin local sur la machine ou un URI `hdfs://`, `s3a://`, etc.) et le lit sous la forme d'une collection de lignes. Voici un exemple d'invocation :
 - `distFile = sc.textFile("data.txt")`.

Enregistrement et chargement des SequenceFiles

- SequenceFile est un fichier plat constitué de paires clé/valeur binaires.
- Il est largement utilisé dans MapReduce en tant que formats d'entrée/sortie.
- Comme pour les fichiers texte, SequenceFiles peut être enregistré et chargé en spécifiant le chemin.

```
>>> 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, u'a'), (2, u'aa'), (3, u'aaa')]
```

RDD (Resilient Distributed Dataset)

- Spark s'articule autour du concept du **RDD (Resilient Distributed Dataset)**, qui est un ensemble d'éléments tolérants aux pannes pouvant être exploités en parallèle.
 - **Resilient**: si des données en mémoire sont perdues, elles peuvent être regénérées,
 - **Distributed**: Les RDDs sont répartis sur plusieurs noeuds d'un cluster,
 - **Dataset**: les données initiales peuvent provenir d'une source telle qu'un fichier ou bien créé par programme.
- Les RDDS sont les unités fondamentales des données dans Spark.
- La plupart de la programmation Spark consiste à effectuer des opérations sur les RDD.

RDD (Resilient Distributed Dataset)

- Les RDD peuvent contenir n'importe quel type d'élément sérialisable :
 - Les types primitifs tels que les entiers, les caractères et les booléens
 - Les types de séquence tels que les chaînes, les listes, les tableaux, les n-uplets et les différents types de données imbriqués.
 - Objets Scala / Java (si sérialisables)

Création d'un RDD

- Vous pouvez créer des RDD de trois manières :
 - En parallélisant une collection existante dans votre programme main ou
 - En utilisant un dataset dans un système de stockage externe, tel qu'un système de fichiers partagé, HDFS, HBase ou toute source de données offrant un Hadoop InputFormat.
 - Un RDD peut être créé à partir d'un autre RDD existant.

Exemple création d'un RDD en utilisant textFile

file1.txt :

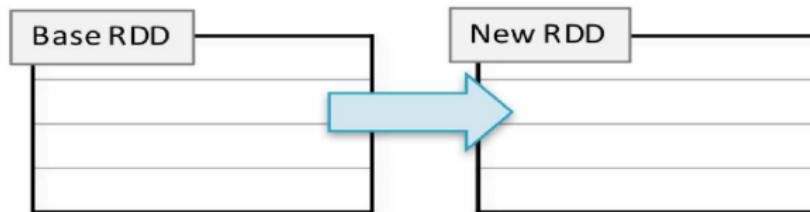
Nos valeurs à l'INPT sont :
Numérique par nature.
Renouvellement permanent.
Innovation et entrepreneuriat.
Ouverture sur l'écosystème.

```
mydata = sc.textFile("file1.txt")
```

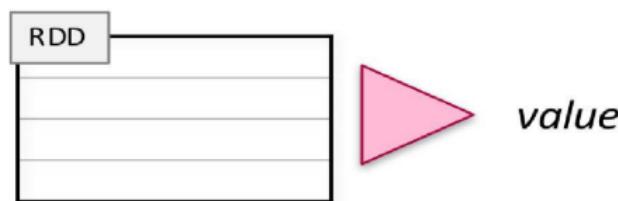
Nos valeurs à l'INPT sont :
Numérique par nature.
Renouvellement permanent.
Innovation et entrepreneuriat.
Ouverture sur l'écosystème.

Opérations sur les RDDs

- Les RDDs prennent en charge deux types d'opérations:
 - Les transformations : elles créent un nouveau RDD à partir d'un RDD existant.



- Les actions : elles renvoient une valeur au programme main après l'exécution d'un calcul sur le RDD.



Exemple d'opérations sur les RDDs

Quelques transformations communes :

- “**map**” passe chaque élément du dataset à une fonction et renvoie un nouveau RDD représentant les résultats.
- “**filter**“ crée un nouveau RDD en incluant ou en excluant chaque enregistrement dans le RDD de base selon une fonction.

Quelques actions communes :

- “**reduce**” regroupe tous les éléments du RDD en utilisant une fonction et renvoie le résultat final au programme main.
- “**count ()**“ renvoie le nombre d’éléments.
- “**take (n)**” retourne un tableau des n premiers éléments.
- “**collect ()**“ retourne un tableau de tous les éléments.
- “**saveAsTextFile (dir)**” enregistre dans un ou plusieurs fichiers texte.

Exemple RDDs opérations

```
> mydata = sc.textFile("ValeursINPT.txt")
> mydata.count()
5
> for line in mydata.take(2):
    print(line)
```

Nos valeurs à l'INPT sont :

Numérique par nature.

Exemple d'opérations sur les RDDs

Nos valeurs à l'INPT sont :
Numérique par nature.
Renouvellement permanent.
Innovation et entrepreneuriat.
Ouverture sur l'écosystème.

```
map(lambda line: line.upper())
```

NOS VALEURS À L'INPT SONT :
NUMÉRIQUE PAR NATURE.
RENOUVELLEMENT PERMANENT.
INNOVATION ET ENTREPRENEURIAT.
OUVERTURE SUR L'ÉCOSYSTÈME.

Exemple d'opérations sur les RDDs

NOS VALEURS À L'INPT SONT :
NUMÉRIQUE PAR NATURE.
RENOUVELLEMENT PERMANENT.
INNOVATION ET ENTREPRENEURIAT.
OUVERTURE SUR L'ÉCOSYSTÈME.

```
filter(lambda line: line.startswith('N'))
```

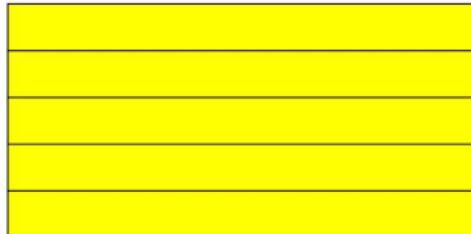
NOS VALEURS À L'INPT SONT :
NUMÉRIQUE PAR NATURE.

Transformations paresseuses (Lazy)

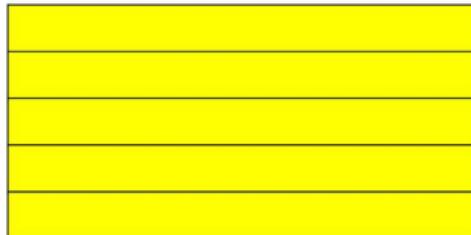
- Toutes les transformations dans Spark sont **paresseuses (Lazy)**, car elles ne calculent pas leurs résultats immédiatement.
- Au lieu de cela, ils se souviennent simplement des transformations appliquées à un dataset de base. Les transformations ne sont calculées que lorsqu'une action nécessite qu'un résultat soit renvoyé au programme main.
- Cette conception permet à Spark de fonctionner plus efficacement.
- Par défaut, chaque RDD transformé peut être recalculé chaque fois que vous exécutez une action dessus. Cependant, vous pouvez également conserver un RDD en mémoire à l'aide de la méthode **persist()** ou **cache()**. Dans ce cas, Spark conservera les éléments sur le cluster pour un accès beaucoup plus rapide la prochaine fois que vous l'interrogerez.
- Il existe également une prise en charge des RDD persistants sur le disque ou répliqués sur plusieurs nœuds.

Exemple de transformations paresseuses (1/3)

```
mydata = sc.textFile("ValeursINPT.txt")
mydata_uc = mydata.map(lambda s: s.upper())
mydata_filt = mydata_uc.filter(lambda s: s.startswith('N'))
RDD: mydata
```



RDD: mydata_uc



Exemple de transformations paresseuses (2/3)

RDD: mydata_filt



```
> mydata = sc.textFile("ValeursINPT.txt")
> mydata_uc = mydata.map(lambda s: s.upper())
> mydata_filt = mydata_uc.filter(lambda s: s.startswith('N'))
> mydata_filt.count()
```

2

RDD: mydata

Nos valeurs à l'INPT sont :
Numérique par nature.
Renouvellement permanent.
Innovation et entrepreneuriat.
Ouverture sur l'écosystème.

Exemple de transformations paresseuses (3/3)

RDD: mydata_uc

NOS VALEURS À L'INPT SONT :
NUMÉRIQUE PAR NATURE.
RENOUVELLEMENT PERMANENT.
INNOVATION ET ENTREPRENEURIAT.
OUVERTURE SUR L'ÉCOSYSTÈME.

RDD: mydata_filt

NOS VALEURS À L'INPT SONT :
NUMÉRIQUE PAR NATURE.

Chaînage des transformations (Python)

- Les transformations peuvent être enchaînées :

```
> mydata = sc.textFile("ValeursINPT.txt")
> mydata_uc = mydata.map(lambda s: s.upper())
> mydata_filt = mydata_uc.filter(lambda s: s.startswith('N'))
> mydata_filt.count()
```

2

est équivalent à :

```
> sc.textFile("ValeursINPT.txt").map(lambda line: line.upper()) \
.filter(lambda line: line.startswith('N')).count()
```

2

Création de RDD à partir de collections

Vous pouvez créer des RDD à partir de collections au lieu de fichiers :

```
> myData = ["Younes","Amine","Othman","Mohamed"]  
> myRdd = sc.parallelize(myData)  
> myRdd.take(2)  
['Younes', 'Amine']
```

Cela est utile dans les cas suivants :

- Test,
- Génération de données par programmation,
- Apprentissage.

Création de RDD à partir de fichiers texte

- Pour les RDD basés sur des fichiers, utilisez `SparkContext.textFile`.
- `textFile` accepte un seul fichier, un répertoire de fichiers, une liste générique de fichiers ou une liste de fichiers séparés par des virgules.

Exemples :

- `sc.textFile(" myfile.txt")`
- `sc.textFile(" mydata/")`
- `sc.textFile(" mydata/ *.log")`
- `sc.textFile(" myfile1.txt,myfile2.txt")`
- Chaque ligne de chaque fichier est un enregistrement séparé dans le RDD.
- Les fichiers sont référencés par un URI absolu ou relatif :
 - URI absolu: *file* : `/home/training/myfile.txt` ou `hdfs : //my/path/myfile.txt`
 - URI relatif (utilise le système de fichiers par défaut): `myfile.txt`

Création de RDD à partir de fichiers texte

- `textFile` associe chaque ligne d'un fichier à un élément RDD distinct.
- `textFile` ne fonctionne qu'avec les fichiers texte qui ont des lignes qui se terminent par des retours à la ligne. Qu'en est-il des autres formats?
- Spark utilise les classes Java Hadoop `InputFormat` et `OutputFormat`, par exemple :
 - `TextInputFormat` / `TextOutputFormat`,
 - `SequenceInputFormat` / `SequenceOutputFormat`,
 - `FixedLengthInputFormat`,
 - Plusieurs implémentations disponibles dans des bibliothèques supplémentaires comme `AvroKeyInputFormat` / `AvroKeyOutputFormat` dans la bibliothèque Avro.

Utilisation de “wholeTextFiles”

- sc.textFile mappe chaque ligne d'un fichier à un élément RDD séparé.
Qu'en est-il des fichiers avec une entrée de format multi-lignes tel que XML ou JSON?
- **sc.wholeTextFiles (répertoire) :**
 - Mappe tout le contenu de chaque fichier dans un répertoire à un seul élément RDD,
 - Fonctionne uniquement pour les petits fichiers (l'élément doit tenir dans la mémoire).

Exemple d'utilisation de “wholeTextFiles”

Appliquez **sc.wholeTextFiles (mydir)** où mydir est un répertoire contenant les deux fichiers json suivants :

file1.json :

```
{"firstName":"Fred",
"lastName":"Flintstone",
"userid":"123"}
```

file2.json :

```
{"firstName":"Barney",
"lastName":"Rubble",
"userid":"234"}
```

```
(file1.json,{"firstName":"Fred","lastName":"Flintstone","userid":"123" })
```

```
(file2.json,{"firstName":"Barney","lastName":"Rubble","userid":"234" })
```

```
(file3.json,... )
```

```
(file4.json,... )
```

Exemple d'utilisation de "wholeTextFiles"

```
> import json  
> myrdd1 = sc.wholeTextFiles(mydir)  
> myrdd2 = myrdd1.map(lambda a: json.loads(a[1]))  
> for record in myrdd2.take(2):  
>     print (record.get("firstName",None))
```

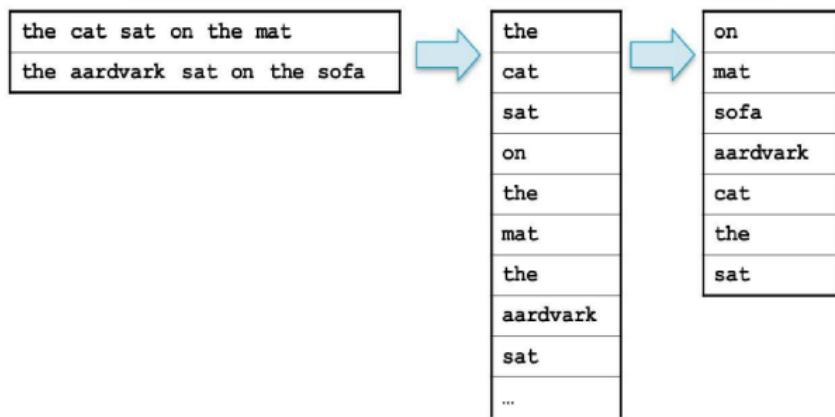
Fred
Barney

Autres transformations communes

- Transformations d'un seul RDD :
 - “**flatMap**” mappe un élément du RDD de base en plusieurs éléments.
 - “**distinct**” filtre les doublons.
 - “**sortBy**” utilise la fonction fournie pour trier.
- Transformations multi-RDD :
 - “**intersection**” crée un nouveau RDD avec tous les éléments des deux RDD d'origine.
 - “**union**” ajoute tous les éléments de deux RDD dans un seul nouveau RDD.
 - “**zip**” associe chaque élément du premier RDD avec l'élément correspondant du second.
 - “**subtract**” supprime les éléments de deuxième RDD du premier RDD.
- Pour avoir une liste plus complètes des différentes transformations, voir le lien :
<https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#transformations>

flatMap et distinct

```
> sc.textFile(file) \
.flatMap(lambda line: line.split(' ')) \
.distinct()
```



Transformations Multi-RDD

rdd1	rdd2
Chicago	San Francisco
Boston	Boston
Paris	Amsterdam
San Francisco	Mumbai
Tokyo	McMurdo Station

`rdd1.subtract(rdd2)`



Tokyo
Paris
Chicago

`rdd1.zip(rdd2)`



(Chicago, San Francisco)
(Boston, Boston)
(Paris, Amsterdam)
(San Francisco, Mumbai)
(Tokyo, McMurdo Station)

Transformations Multi-RDD

rdd1
Chicago
Boston
Paris
San Francisco
Tokyo

rdd2
San Francisco
Boston
Amsterdam
Mumbai
McMurdo Station

`rdd1.union(rdd2)`



Chicago
Boston
Paris
San Francisco
Tokyo
San Francisco
Boston
Amsterdam
Mumbai
McMurdo Station

`rdd1.intersection(rdd2)`



Boston
San Francisco

Pair RDD

Les Pair RDD sont une forme particulière de RDD :

- Chaque élément doit être une paire clé-valeur (un tuple à deux éléments)
- Les clés et les valeurs peuvent être n'importe quel type
- Ces Pair RDD sont très utilisés avec les algorithmes map-reduce.
- Création des pair RDD :
 - La première étape consiste à obtenir les données sous forme clé / valeur.
 - Fonctions couramment utilisées pour créer des Pair RDD : map, flatMap, keyBy.

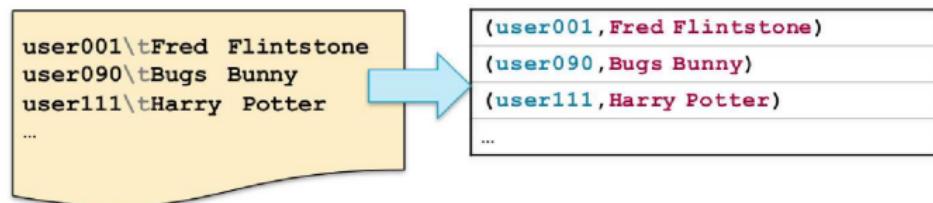
Pair RDD

(key1, value1)
(key2, value2)
(key3, value3)
...

Example 1

Création d'une paire RDD à partir d'un fichier séparé par des tabulations :

```
> users = sc.textFile(file) \
.map(lambda line: line.split('\t')) \
.map(lambda fields: ( fields[0], fields[1] ))
```



Pair RDD

Les Pair RDD sont une forme particulière de RDD :

- Chaque élément doit être une paire clé-valeur (un tuple à deux éléments)
- Les clés et les valeurs peuvent être n'importe quel type
- Ces Pair RDD sont très utilisés avec les algorithmes map-reduce.
- Création des pair RDD :
 - La première étape consiste à obtenir les données sous forme clé / valeur.
 - Fonctions couramment utilisées pour créer des Pair RDD : map, flatMap, keyBy.

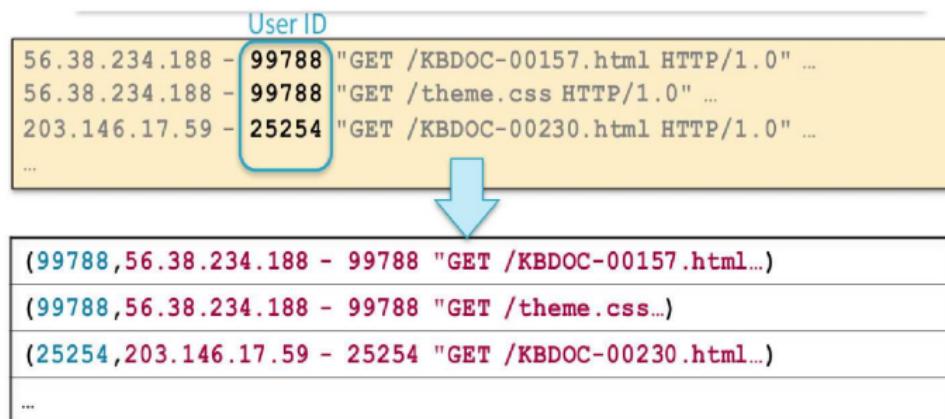
Pair RDD

(key1, value1)
(key2, value2)
(key3, value3)
...

Example 2

Keying des Web Logs par l'identifiant de l'utilisateur :

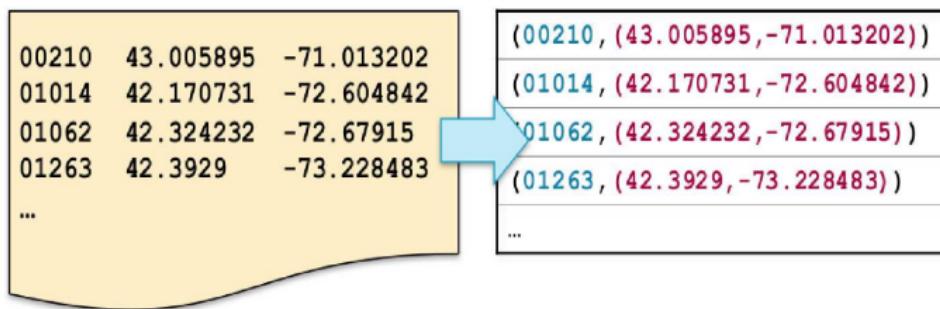
```
> sc.textFile(logfile) \
.keyBy(lambda line: line.split(' ')[2])
```



Example 3

Pair RDD avec des valeurs complexes :

```
> sc.textFile(file) \
.map(lambda line: line.split('\t')) \
.map(lambda fields: (fields[0],(fields[1],fields[2] )))
```



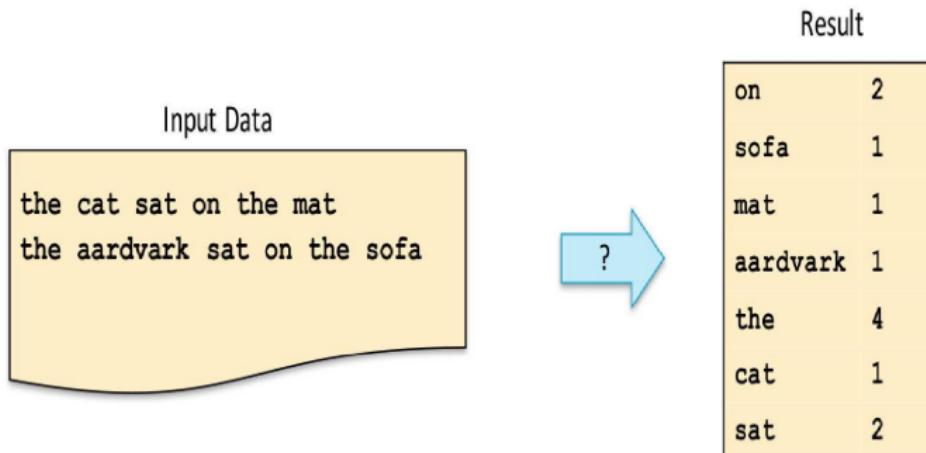
Fonction “keys()”

keys() permet l'extraction des Clés d'un PairRDD.

- Supposons que nous avons un RDD de paires représentant des produits et leurs prix :
 - `products = sc.parallelize([("Chaussures", 50), ("Chemise", 20), ("Pantalon", 40)])`
- Pour récupérer uniquement les noms de produits, on utilise `keys()` :
 - `product_names = products.keys()`
- En exécutant cela, `product_names` contiendra uniquement les noms des produits :
 - `print(product_names.collect())`
 - Sortie : `['Chaussures', 'Chemise', 'Pantalon']`

La méthode `keys()` permet d'extraire toutes les clés d'un PairRDD sans leurs valeurs associées. C'est utile pour des opérations où seules les clés sont nécessaires.

Exemple Map-Reduce : Word Count



Exemple Map-Reduce : Word Count

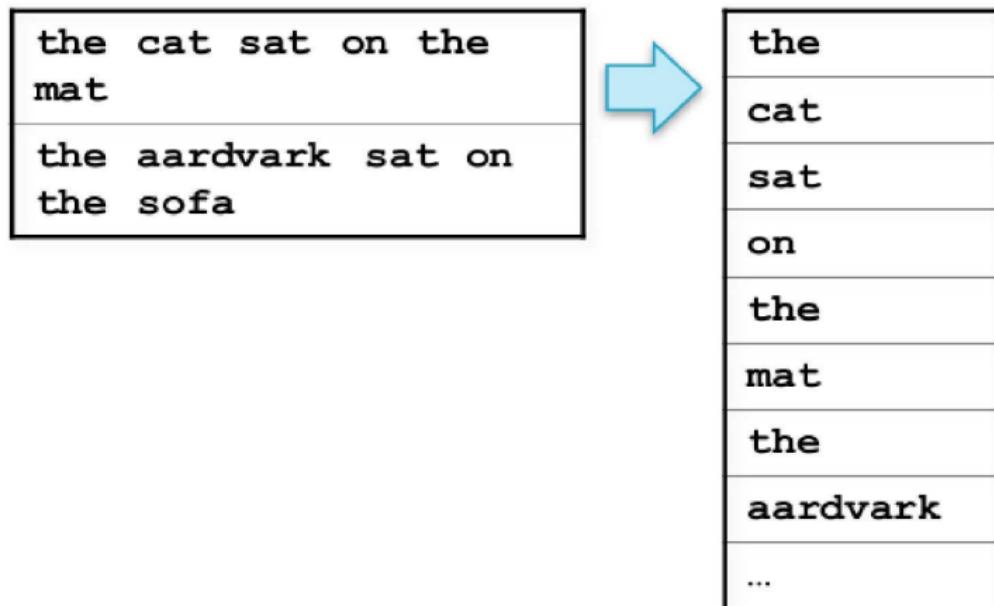
```
> counts = sc.textFile(file)
```

the cat sat on the
mat

the aardvark sat on
the sofa

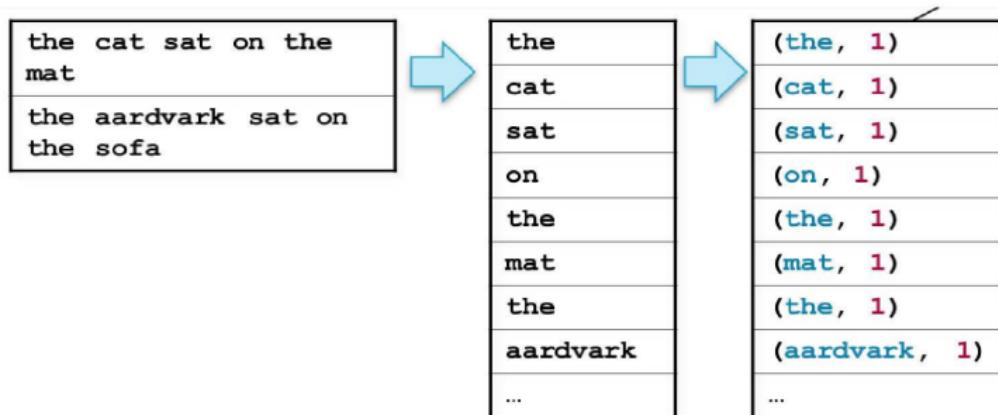
Exemple Map-Reduce : Word Count

```
> counts = sc.textFile(file) \  
.flatMap(lambda line: line.split(' '))
```



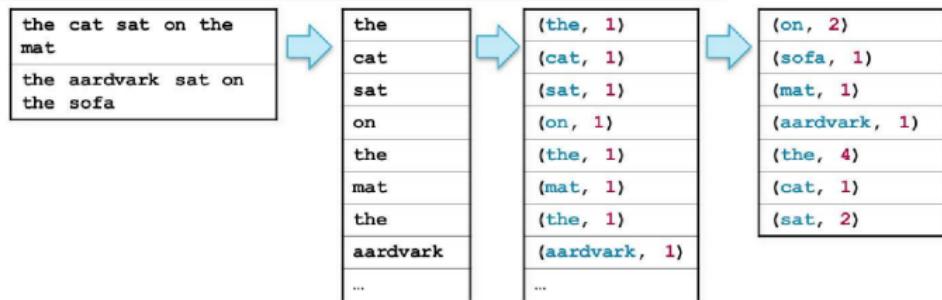
Exemple Map-Reduce : Word Count

```
> counts = sc.textFile(file) \  
.flatMap(lambda line: line.split(' ')) \  
.map(lambda word: (word,1))
```



Exemple Map-Reduce : Word Count

```
> counts = sc.textFile(file) \
.flatMap(lambda line: line.split(' ')) \
.map(lambda word: (word,1)) \
.reduceByKey(lambda v1,v2: v1+v2)
```



Plan

- 1 Présentation Spark
- 2 Architecture Spark
- 3 Concepts de base de Spark
- 4 Tolérance aux pannes dans Spark
- 5 Implémentation et exécution des applications sous Spark

Tolérance aux pannes dans Spark

Le mécanisme de tolérance aux pannes dans Apache Spark repose principalement sur la notion de RDDs et l'utilisation du DAG (Directed Acyclic Graph).

- **Immutabilité et Tolérance aux Pannes grâce aux RDDs** : Les RDD sont des structures de données immuables, ce qui signifie qu'une fois créées, elles ne peuvent pas être modifiées. Cela permet à Spark de conserver un historique de toutes les transformations appliquées à un RDD d'origine pour arriver à sa version finale.
- **Linéage** : Spark garde une trace de ce que l'on appelle le "linéage" des RDD, c'est-à-dire une série de transformations (comme `map()`, `filter()`, etc.) qui mènent à la génération des RDD finaux. Cette trace permet à Spark de réexécuter ces transformations à partir des données d'origine si une partition est perdue.

Directed Acyclic Graph (DAG)

Définition d'un DAG (Directed Acyclic Graph) :

- Un DAG est une structure de données utilisée pour représenter des processus où les nœuds du graphique sont liés par des arêtes dirigées, et il n'y a pas de cycles (cela garantit qu'il n'y a pas de dépendances circulaires, ce qui serait problématique pour l'exécution).

Principaux rôles du DAG dans Spark

- **Représentation des Tâches :**

- Le DAG représente l'ensemble des transformations appliquées sur les données. Chaque nœud du DAG correspond à une opération sur les données (transformation ou action), tandis que les arêtes représentent les dépendances entre ces opérations.
- Le DAG permet à Spark de visualiser l'ensemble des calculs nécessaires pour produire le résultat final.

- **Optimisation de l'efficacité des exécutions :**

- En utilisant l'évaluation lazy pour retarder et optimiser l'exécution,
- En fusionnant les étapes ne nécessitant pas de shuffle (comme map, filter) pour minimiser les lectures/écritures. (Les étapes qui nécessitent un shuffle sont par ex. reduceByKey ou groupByKey, elles exigent une réorganisation des données entre les nœuds).
- En exécutant les tâches en parallèle grâce à une planification distribuée.

- **Tolérance aux Pannes :**

- Si une tâche échoue, Spark peut utiliser le DAG et le lineage pour retracer les étapes nécessaires à la recomposition des données perdues.

Tolérance aux pannes dan Spark

- Spark essaie de stocker les RDDs en mémoire si possible pour accélérer les calculs. Cependant, ces RDDs ne sont pas répliqués en mémoire par défaut.
- Si un exécuteur échoue et que des RDDs en mémoire sont perdus, Spark peut soit **réaffecter les tâches** à un autre exécuteur ou soit **refaire les calculs** grâce au DAG et au linéage des RDDs.

- **Réaffectation des tâches** : Les tâches en cours d'exécution sur ce nœud sont réaffectées à d'autres exécuteurs disponibles dans le cluster. Même si Spark ne réplique pas activement les RDD, les données sources sont souvent répliquées par le système de stockage sous-jacent (HDFS, S3, etc.). C'est à partir de là que Spark peut relire les blocs de données pour réexécuter les tâches manquantes.
- **Recalcul des partitions perdues (Recomputation)** : Spark réexécute les transformations à partir des données disponibles, en suivant le DAG et le lineage pour calculer les partitions manquantes (si celles-ci sont rapides et les données intermédiaires facilement recomputables).

Optimisation par la mise en cache

- Pour éviter d'avoir à recalculer des partitions coûteuses en cas de panne, Spark permet de mettre en cache (avec `cache()` ou `persist()`) les RDD. Cela stocke les RDD en mémoire (ou sur disque) pour un accès plus rapide.
- Ainsi, si un nœud échoue, Spark peut récupérer les partitions perdues directement depuis la mémoire ou le disque, sans devoir tout recalculer.

Plan

- 1 Présentation Spark
- 2 Architecture Spark
- 3 Concepts de base de Spark
- 4 Tolérance aux pannes dans Spark
- 5 Implémentation et exécution des applications sous Spark

Exemple de script python "WordCount.py"

```
import sys
from pyspark import SparkContext, SparkConf
if __name__ == "__main__":
    # create Spark context with necessary configuration
    sc = SparkContext("local", "PySpark Word Count Example")

    # read data from text file and split each line into words
    words = sc.textFile("/poeme.txt").flatMap(lambda line: line.split(" "))

    # count the occurrence of each word
    wordCounts = words.map(lambda word: (word, 1)).reduceByKey(lambda a,b:a +b)
    # save the counts to output
    wordCounts.saveAsTextFile("file0.count")
```

Pour exécuter ce script, tapez : **\$ spark-submit WordCount.py fileURL**

Exécution d'application sous Spark

- Spark peut s'exécuter :
 - localemement (Pas de traitement distribué)
 - localemement avec plusieurs threads de travail
 - sur un cluster
- Le mode local est utile pour le développement et les tests.
- Pour des utilisation de la production, on exécute presque toujours sur un cluster
- On peut choisir le gestionnaire de ressources : standalone, YARN ou MESOS.
- Exemple d'exécution en local avec 3 threads:

Language: Python

```
$ spark-submit --master 'local[3]' \
WordCount.py fileURL
```

Language: Scala/Java

```
$ spark-submit --master 'local[3]' --class \
WordCount MyJarFile.jar fileURL
```

Exécution d'application sous Spark

Utilisez spark-submit –master pour spécifier une option de cluster :

- yarn-client
- yarn-cluster
- *spark : //masternode : port* (Spark Standalone)
- *mesos : //masternode : port* (Mesos)

Language: Python

```
$ spark-submit --master yarn-cluster \
WordCount.py fileURL
```

Language: Scala/Java

```
$ spark-submit --master yarn-cluster --class \
WordCount MyJarFile.jar fileURL
```

Options couramment utilisées du "spark-submit"

- **--class**: chemin de la classe main de votre application
- **--master**: l'URL principale du cluster (par exemple, spark://23.195.26.187: 7077)
- **--deploy-mode**: Indique le mode de déploiement (mode client ou mode cluster)
- **--jars**: fichiers JAR supplémentaires (Scala et Java uniquement)
- **--py-files**: fichiers Python supplémentaires (Python uniquement)
- **--driver-java-options**: paramètres à transmettre à la machine virtuelle Java du driver.
- **--executor-memory**: Mémoire par exécuteur (par exemple: 1000m, 2g) (par défaut: 1g)
- **--packages**: coordonnées Maven d'une bibliothèque externe à inclure

Autres options spécifiques à YARN :

- **--num-executors**: Nombre d'exécuteurs à démarrer
- **--executor-cores**: nombre de cœurs à allouer pour chaque exécuteur
- **--queue**: file d'attente du YARN pour soumettre l'application

Pour afficher toutes les options disponibles :

- **--help**

- Les applications Scala ou Java Spark doivent être compilées et assemblées dans des dossiers JAR qui seront transmis aux nœuds workers.
- Apache Maven est un outil de construction populaire :
 - Pour des recommandations de réglage spécifiques avec spark, voir la documentation officielle de Spark
- Envisagez d'utiliser un environnement de développement intégré (IDE) :
 - IntelliJ ou Eclipse sont deux exemples populaires.
- Pour plus de détails sur la soumission et l'exécution des applications Spark, vous pouvez voir le TP2.