TP Spark Python

L. Benyoussef & S. Derrode

Ce TP a pour objectif de vous faire découvrir Apache Spark (http://spark.apache.org), qui fait partie de l'écosystème Hadoop. Spark est écrit en Scala[1], et il supporte différents langages de programmation pour le développement d'applications : Java, R[2], Scala, et Python. C'est avec ce dernier langage que nous travaillerons ici.

Pour rappel, vous pouvez vous connecter au serveur grâce à la commande « *ssh login@156.18.90.100* », où *login* à la forme suivante : *mso31_X*, avec *X* le numéro qui vous a été affecté durant le premier TP. Si vous n'avez pas changé de mot de passe, alors celui-ci est identique au *login*.

Les fichiers nécessaires à ce TP sont a priori déjà dans le répertoire « tp-hadoop-python » que nous avions téléchargé grâce à *git* durant le premier TP (« *git clone …* »). Mais il n'est pas impossible que les sujets aient été modifiés depuis votre téléchargement. Nous allons donc mettre à jour votre copie locale, sans perdre les programmes que vous avez réalisés durant le premier TP. Pour cela, suivez scrupuleusement les étapes suivantes :

1. « *cd ~/tp-hadoop-python* »
2. « *git config --global user.name "votre nom"* » # enregistrement par git de votre nom (à ne faire qu'une fois pour toute).
3. « *git config --global user.email "votre adresse mail"* » # enregistrement par git de votre adresse mail (à ne faire qu'une fois pour toute).
4. « *git config --global core.editor nano* » # on définit nano comme éditeur de texte par défaut (à ne faire qu'une fois pour toute).

5. « *git add -A* » # Tous les fichiers que vous avez créés durant le premier TP sont ajoutés à la gestion par *git*.
6. « *git commit -a -m "mon travail de TP"* » # Enregistrement local de votre travail sur le 1er TP
7. « *git pull origin* » # récupération des nouveaux fichiers depuis le serveur github.

   Si cela est nécessaire, nano se lancera automatiquement. Dans ce cas, il vous suffit de quitter nano (*ctrl+w*, puis *ctrl+x*).

Si tout s'est bien passé, tapez « *cd Spark* ». Vous êtes prêt pour commencer le second TP.

COURANT AVRIL : vos comptes seront détruits sans préavis, n'oubliez pas de faire une sauvegarde de vos fichiers sur votre disque dur local si vous le souhaitez.

1. MAPREDUCE VS SPARK

Spark se définit comme un moteur général de traitement de données massives, qui affiche des temps d'exécution jusque 100 fois inférieurs à *MapReduce*, lorsque les traitements sont lancés en mémoire vive. Dans le cas où les traitements sont lancés sur le disque dur, les temps sont « seulement » 10 fois inférieurs.

[1] Scala est un langage de programmation fonctionnelle et orienté objet (https://www.scala-lang.org).

[2] R est un logiciel libre de traitement des données et d'analyse statistique, très utilisé par les statisticiens.

La principale raison vient du fait qu'Hadoop fait sur traitement en mode lot sur le disque dur, alors que *Spark* peut faire du traitement en mémoire, économisant ainsi tous les temps d'accès aux disque durs[3] :

> *« La **séquence de travail de MapReduce** ressemble à ceci : il lit les données au niveau du cluster, il exécute une opération, il écrit les résultats au niveau du cluster, il lit à nouveau les données mises à jour au niveau du cluster, il exécute l'opération suivante, il écrit les nouveaux résultats au niveau du cluster, etc. »*
> *« **Spark** lit les données au niveau du cluster, effectue toutes les opérations d'analyses nécessaires, écrit les résultats au niveau du cluster, et c'est tout »*

Notons cependant que *Spark* requiert de grosses quantités de ressources de mémoire vive (de la taille des données à traiter). Si ces ressources ne sont pas disponibles pour stocker les données en mémoires, alors les performances de *Spark* se dégradent considérablement puisqu'il utilise alors des accès aux disques durs. En conséquence, le principe *MapReduce* est plus souple que celui de *Spark* car il libère des ressources dès qu'il n'en a plus besoin, permettant à d'autres Job de s'exécuter en parallèle.

Spark ne devient indispensable que si on souhaite faire des traitements de données lourds, comme pour des applications numériques/scientifiques, notamment sur des données mis-à-jours en flux continu (données de *streaming* par exemple). Il est particulièrement intéressant d'utiliser *Spark* si on doit utiliser les données (stockées en mémoire vive) plusieurs fois, ce qui est typique des algorithmes itératifs que l'on utilise dans le domaine du *Machine Learning*.

Notons toutefois que du traitement de données en temps-réel peut être réalisé en *MapReduce* grâce à *Storm* (http://storm.apache.org) ou *Impala* (http://impala.apache.org), que le traitement *de* graph peut être fait grâce à *Giraph* (http://giraph.apache.org), et qu'Apache *Mahout* (http://mahout.apache.org) offre aussi une librairie de *Machine Learning*.

Spark n'a que peu d'intérêt si on souhaite faire du *reporting* d'activité commerciale par exemple. Également, si *MapReduce* peut être utilisé par de nombreux métiers grâce à l'écosystème (*Hive*, *Pig*), *Spark* nécessite des compétences techniques que seul un informaticien de métier peut maîtriser : il est donc moins accessible[4].

Notons finalement que *Spark* peut fonctionner en mode *standalone* mais bénéficie du système de stockage distribué proposé par Hadoop : il est capable de lire des données directement sur HDFS. C'est dans ce mode que nous allons travailler (grâce à « --master local[2] », par exemple). Le serveur hadoop étant en panne, nous ne pourrons pas utiliser le mode « --master hadoop » qui aurait permis d'exploiter le server de calculs. Mais nous pouvons utiliser l'espace de stockage HDFS.

2. LET'S START !

Comment utiliser Spark avec Python ?

Nous avons deux modes : mode interactif Shell, et mode script/job

- **Mode Spark Shell / python :** il suffit de taper : « *pyspark* » à l'invite de commande (pour utiliser le langage Scala, lancer : « *spark-shell* »). Les *shells* ouvrent automatiquement un objet *SparkContext*, appelé par défaut *sc*, que vous pouvez directement utiliser par des commandes telles que « *sc.version* » ou « *sc.appName* ». Vous pouvez alors directement utiliser les commandes vues en cours. Pour quitter le mode console : « *quit()* ».
- **Mode script :** Pour cela vous devez programmer un script python et l'exécuter grâce à l'outils « *spark-submit* ». Voici un exemple de commande :

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

[3] http://www.lemondeinformatique.fr/actualites/lire-hadoop-vs-spark-apache-5-choses-a-savoir-63271.html.

[4] Notons toutefois l'existence de SparkQL (http://spark.apache.org/sql/), entièrement compatible avec Hive, qui comble partiellement cette lacune.

Cette commande stipule que vous allez exécuter *Spark* en mode Local (*i.e.* votre processeur) sur le fichier « *PySpark_Pi.py* » avec la valeur 10 comme argument (ce mode sera le mode privilégié pour la séance de manière à éviter des étranglements au niveau du serveur). Exécutez cette commande pour obtenir une approximation de *pi*. Dans le fichier « PYSPARK_P*i.py* », vous remarquerez :

- La présence de la librairie *Spark* pour Python : « *from pyspark import SparkContext* ».
- Dans le programme principal, la création et le lancement d'un contexte *Spark* dénommé « *sc* », et de son arrêt par « *sc.stop()* ».
- L'appel d'abord à la méthode « *sc.parallelize(…)* » qui crée un RDD pour réaliser les traitements parallèles, puis l'utilisation successive des méthodes Python « *map(…)* » et « *reduce (…)* ». Celles-ci s'appliquent sur l'objet renvoyé par la méthode précédente (chaînage).

Travail à réaliser

1. **Python en programmation fonctionnelle :** Il s'agit de programmer un algorithme qui calcule le nombre de secondes à partir d'une heure donnée dans le format suivant : « hh:mm:ss ». Ainsi « 8:19:22 » donnera 29962 secondes. Développer une première version itérative « python structurée », puis une version « python fonctionnelle » de l'algorithme (avec « *map(…)* » et « *reduce(…)* »).
2. **PySpark :** Ré-exécutez certains exemples du cours pour vous les approprier. Vous trouverez l'ensemble des fichiers dans le sous-répertoire « *exemple_cours* ». Pour les fichiers commençant par « *PySpark* » (et nécessitant donc *Spark*), adaptez la commande suivante « *spark-submit --master local[2] PySpark_exemple3.py* ».

3. PySpark « wordcount » improved !

Revenez dans le répertoire principal du TP à l'aide de la commande « *cd ..* ».
Suite au précédent TP, le livre « *dracula* » doit se trouver dans le répertoire « *livres* » de votre espace sur HDFS (à vérifier grâce à « *hdfs dfs -ls livres* »).
Lancez alors le Job Spark à l'aide de la commande suivante :

>> spark-submit --master local[2] PySpark_wc1bis.py livres/dracula

Observez (« *hdfs dfs -ls sortie* ») que le résultat est stocké dans deux fichiers de sortie : « *part-00000* » et « *part-00001* » (la présence du fichier vide « *_SUCCESS* » signifie que le job s'est terminé normalement), et que les résultats ne sont pas triés par ordre alphabétique. La raison de la présence des deux fichiers provient du mode de calcul distribué (ici nous avons 2 partitions). Pour fusionner les 2 fichiers en 1 seul, vous pouvez réduire le nombre de partitions à 1 en remplaçant la ligne

wordCounts.saveAsTextFile("sortie")

par la ligne

wordCounts.coalesce(1).saveAsTextFile("sortie")

Attention : cette commande doit être utilisée avec parcimonie, c'est à dire lorsque le nombre de données à traiter est faible. Supprimer cette commande pour la suite.
Si vous souhaitez obtenir un seul fichier, il est préférable de lancer la commande Linux suivante, a posteriori, qui agit sur les fichiers stockés sous hdfs : « *hdfs dfs -cat sortie/* | hdfs dfs -put - sortie/output.txt* » qui fusionne tous les fichiers présents dans le répertoire « *sortie* » dans un fichier « *output.txt* » lui-même sous hdfs dans le répertoire « *sortie* ».
mo

Modifications à réaliser

À partir de « *PySpark_wc1bis.py* », essayer d'implémenter les modifications suivantes. Je vous rappelle qu'entre 2 exécutions, il est nécessaire détruire le répertoire « *sortie* » de votre espace de stockage hdfs, sous peine de voir apparaître un message d'erreur, pas forcément facile à interpréter !

1. PySpark_wc2.py : S'assurer que les sorties soient triées par ordre alphabétique. Vérifiez que l'ordre alphabétique sur les deux fichiers est bien consécutif et observez le nombre de lignes de chaque fichier grâce à la commande suivante : « *hdfs dfs -cat sortie/part-00000 | wc -l* » (« *wc* » est une commande Linux qui permet d'avoir des infos sur un fichier, « *-l* » signifiant le nombre de lignes).

2. PySpark_wc3.py : Ne garder que les mots qui apparaissent dans le texte au moins *X* fois, la valeur de *X* étant fixée par un argument supplémentaire lors de l'appel à « *spark-submit* ». Par exemple :
   >> spark-submit --master local[2] PySpark_wc3.py livres/Dracula 1000
   Pour récupérer la valeur entière passée en argument, utiliser la commande suivante : « *threshold = int(sys.argv[2])* » (la variable *threshold* aura la valeur 1000 dans cet exemple). L'argument « *sys.argv[1]* » représente le nom du fichier, et « *sys.argv[0]* » le nom du programme. N'oubliez pas « *import sys* » au début du fichier Python.

3. PySpark_wc4.py : En repartant de la version précédente, proposer un algorithme pour compter le nombre total de lettres de tous les mots restants après filtrage.

4. En fonction du temps qu'il vous reste, écrivez un algorithme pour
   a. Calculer la moyenne du nombre de caractères de chaque ligne (PySpark_wc5.py).
   b. Corriger les problèmes liés à la ponctuation, *cf* TP Hadoop Python (PySpark_wc6.py).

4. IL EST OU LE BEL ARBRE ?

On considère ici le fichier de données *Opendata* de type CSV provenant de http://opendata.paris.fr décrivant des arbres remarquables à Paris (ce fichier est déjà présent dans le répertoire *Spark*) :
https://opendata.paris.fr/explore/dataset/arbresremarquablesparis/information/
Lancez « *more arbresremarquablesparis.csv* ». Chaque ligne décrit un arbre : position GPS, arrondissement, genre, espèce, famille, année de plantation, hauteur, circonférence, etc. Le séparateur entre les champs est ';'. La première ligne contient les titres des colonnes.

Écrire un programme *Spark* permettant d'afficher les coordonnées GPS de l'arbre le plus grand (on affichera aussi sa taille). **Trucs :**

- Certains arbres n'ont pas de taille renseignée : on devra donc filtrer le RDD en supprimant les lignes dont la hauteur est égale à ''. Le même filtrage permettra de ne pas considérer la 1ière ligne qui contient les en-têtes des colonnes.
- Vous allez récupérer la hauteur comme une chaîne de caractères. Pour la transformer en nombre réel : « *float(hauteur)* »
- Pour vérifier le résultat, vous pouvez entrer les coordonnées GPS sur le site :
  http://www.coordonnees-gps.fr/
  et vérifiez que l'adresse est bien la bonne !

5. LIBRAIRIE *SPARK STREAMING* : TRAITEMENT DE FLUX CONTINUS

Spark streaming[5] est une extension de la librairie principale de *Spark*, qui permet de traiter des flux continus de données. Elle est tolérante aux erreurs et permet de réaliser des algorithmes complexes grâce à des fonctions de haut niveau comme *map*, *reduce*, *join*, *window*. Il est possible d'appliquer les algorithmes de *Spark* en *Machine Learning* (*MLlib*) ou en *Graph Processing* (*GraphX*[6]) sur les flux.

[5] ***Spark streaming*** *programming guide*: https://spark.apache.org/docs/1.5.0/streaming-programming-guide.html
[6] ***GraphX*** *programming Guide*: https://spark.apache.org/docs/1.5.0/graphx-programming-guide.html

Spark Streaming reçoit les flux de données et les divise en paquets qui sont traités par le *Spark Engine* pour générer le résultat sous forme de paquets.

La librairie fournit des objets appelés *DStream*, pour *Discretized Stream*, qui représente un flux continu de données. Les *DStream* peuvent soit être créés par des flux de données entrants (HDFS), soit par des sources provenant de *Kafka*[7], *Flume*[8] ou *Kinesis*[9], ou encore par des opérations sur des *DStreams*. En interne, un *DStream* est représenté par une séquence de RDDs.

Un exemple : *wordcount* en streaming

Comme exemple, nous allons mettre en œuvre un « *wordcount* » adapté au streaming. Pour cela, vous aurez besoin de 2 terminaux. Pour les deux terminaux, loggez-vous sur votre compte sur le cluster. Alignez vos deux terminaux sur votre écran pour les voir simultanément.

- Dans le Terminal 1, lancez le serveur de données : « *nc -lk 9999* » (nc pour « *netcat* »).
- Dans le Terminal 2, entrez dans le répertoire « *~/tp-hadoop-python/Spark* » et lancez :
  spark-submit --master local[2] SparkStreaming_wc.py localhost 9999

Revenez au Terminal 1 et tapez des mots. Pour entrer une série de mots d'un coup, vous pouvez insérer un espace entre chaque mot, validez par la touche « entrée ». Vous devriez voir l'effet du « *wordcount* » sur les mots sous forme de tuple (clé, valeur) dans le Terminal 1.
Les paquets de données correspondent à une durée d'1 seconde. Pour allonger le temps associé aux paquets de données, il suffit de modifier la ligne « *ssc = StreamingContext(sc, 1)* », en remplaçant la valeur 1 par la valeur souhaitée en secondes.

Dans cet exemple, la variable « *lines* » est un *DStream* entrant, alimenté par le serveur *netcat*. On distingue deux types de sources :

- Les « *basic sources* », exemples : *file systems*, *socket connections*, et *Akka actors*.
- Les « *advanced sources* », exemples : sources depuis *Kafka*, *Flume*, *Kinesis*, *Twitter*.

L'exemple précédent illustre une connexion à une source basique à travers un serveur :
lines = ssc.socketTextStream(sys.argv[1], int(sys.argv[2])).

6. REMARQUE : MRJOB ET PYSPARK

Il est possible de lancer un job *Spark* dans un environnement mrjob, pour bénéficier instantanément des fonctionnalités de *mrjob* (les fichiers, les paramètres, la connexion à EMR d'Amazon…), *cf* Figure ci-dessous :
https://mrjob.readthedocs.io/en/latest/guides/spark.html

On peut même mixer des jobs *mrjob* avec des jobs *Spark* :

```
def steps():
  return [
    MRStep(mapper=self.preprocessing_mapper),
    SparkStep(spark=self.spark),
  ]
```

[7] **Apache Kafka**: *A fast, scalable, fault-tolerant messaging system* (https://kafka.apache.org/).

[8] **Apache Flume**: *Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data* (https://flume.apache.org).

[9] **Amazon Kinesis** : données diffuses en temps réel dans le cloud AWS (https://aws.amazon.com/fr/kinesis/).

```python
import re
from operator import add

from mrjob.job import MRJob

WORD_RE = re.compile(r"[\w']+")


class MRSparkWordcount(MRJob):

    def spark(self, input_path, output_path):
        # Spark may not be available where script is launched
        from pyspark import SparkContext

        sc = SparkContext(appName='mrjob Spark wordcount script')

        lines = sc.textFile(input_path)

        counts = (
            lines.flatMap(lambda line: WORD_RE.findall(line))
            .map(lambda word: (word, 1))
            .reduceByKey(add))

        counts.saveAsTextFile(output_path)

        sc.stop()


if __name__ == '__main__':
    MRSparkWordcount.run()
```