1. Introduction

Cette semaine, nous allons découvrir Spark, un outil de calcul distribué très efficace.

Pour commencer, téléchargez l'archive `TP4.zip` et décompressez-la. Elle crée un dossier TP4 contenant déjà quelques fichiers. Mettez-vous dedans. Durant le TP, vous allez produire différents scripts pySpark (Spark pour Python). Chaque script correspond à une question. Ils seront nommés de manière à rappeler la question, ex: `tp4_2_1.py` pour la question 2.1.

Attention, Spark n'aime pas les espaces dans les chemins et noms. Ex: `.../Big Data/...` Il faudrait renommer en `.../Big_Data/...`

2. Tutoriel sur `arbres.csv`

Pour tous les exercices, il est recommandé de consulter la documentation des RDD :

http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD

Il y a un grand nombre de méthodes. Il faut seulement étudier celles qui sont utilisées ici.

2.1. Nombre d'arbres

On va commencer avec `tp4_2_1.py`. Ce programme affiche le nombre de lignes du fichier `hdfs:/share/paris/arbres.csv`. C'est pas exactement le nombre d'arbres car il y a la ligne de titre, mais peu importe pour ce premier exemple.

On crée un RDD à partir du fichier, puis on appelle la méthode `count()`. Un RDD est une sorte de liste, et si elle est assez grande, elle est distribuée sur plusieurs machines pour un traitement parallèle.

Vous éditerez ce script avec `idle` ou `geany` sur votre poste local. Et pour le lancer, il faudra aller dans la fenêtre `ssh hadoop`. Il y a deux façons de lancer l'exécution du script :

- `spark-submit --master spark://master:7077 tp4_2_1.py`
  Ça lance l'exécution sur le cluster Spark. C'est le plus rapide pour du Big Data. Le cluster de l'IUT compte 4 esclaves appelés *spark workers* qui vont exécuter le programme en parallèle, pourvu qu'il y ait assez de travail pour chacun. Par contre, c'est pas très rentable si le fichier est minuscule.
  Chaque esclave possède 4 cœurs (*cores*), cela fait 16 cœurs utilisables pour un travail. Il y a une file d'attente des travaux que vous lancez. Tous les cœurs sont attribués au premier travail lancé ; les autres doivent attendre qu'il se finisse. Une autre stratégie consisterait à attribuer seulement une partie des cœurs à chaque travail.
  Lorsque le travail sera fini, vous pourrez aller voir l'historique sur la page Spark sur hadoop ou sur Spark sur master si vous avez reconfiguré le proxy comme dans le TP1 et le TP3.
- `spark-submit tp4_2_1.py`
  Ça lance l'exécution en local, sur master. Dans l'absolu, c'est le mode le plus lent car les données à traiter sont transférées sur cette machine au lieu de rester dans HDFS. Cependant pour de tout petits fichiers, c'est lui le plus rapide. Dans ce mode, il n'y a aucun historique.

NB: si vous souhaitez toujours lancer dans le cluster, alors transformez la 10^{e} ligne du script en :

```
conf = SparkConf().setAppName(appName).setMaster("spark://master:7077")
```

De cette façon, en tapant `spark-submit tp4_2_1.py`, ça sera lancé dans le cluster et non en local.

C'est déjà fait sur certains des prochains scripts.

2.2. Hauteur moyenne des arbres

Dans la suite de ce TP, nous allons lire des fichiers CSV, extraire et convertir des champs. C'est assez illisible de faire ces extractions en plein dans les fonctions de calcul Spark. On a donc fait comme dans les TP2 et TP3, on a défini une classe utilitaire.

Savez-vous écrire une classe en Python ? Par rapport à Java, c'est assez bizarre. L'indentation délimite le contenu de la classe. L'entête est bizarre, le constructeur a un nom bizarre. Il faut mettre `self` en tant que premier paramètre dans toutes les méthodes, mais ce paramètre est comme invisible lors de l'appel. La création d'une instance est bizarre, il n'y a pas de `new`. Voici un exemple :

```
class MaClasse(object):

    # constructeur
    def __init__(self, param1, param2):
        self.membre1 = param1
        self.membre2 = param2

    # affichage (equivalent de toString())
    def __str__(self):
        return "MaClasse['%s', %s]" % (self.membre1, self.membre2)

    # accesseur
    def getMembre1(self):
        return self.membre1

instance = MaClasse("pi", 3.1415);
print instance
print instance.getMembre1()
```

Maintenant, on étudie `tp4_2_2.py`. Il calcule et affiche la hauteur moyenne des arbres. Il importe la classe `Arbre`. On a simplifié les noms des *getters* parce qu'ils sont les seules méthodes.

Pour le lancer, il faut rajouter l'option `--py-files arbre.py` :

- `spark-submit --master spark://master:7077 --py-files arbre.py tp4_2_2.py`
- `spark-submit --py-files arbre.py tp4_2_2.py`

Cela entraîne la création d'un fichier `arbre.pyc` contenant le code Python compilé. C'est comme un `.class` pour Java.

Vous pouvez voir qu'on reprend des éléments du TP1 :

- La séparation d'un RDD en champs à l'aide de la classe `Arbre`.
- Récupération du champ qui nous intéresse ou `None` si ça échoue (absent ou ligne de titre).
- Suppression des valeurs `None` à l'aide d'un filtre. Le paramètre de `filter` est soit `None` pour simplement enlever ces valeurs du RDD, soit une fonction booléenne appelée sur chaque élément — l'élément est enlevé si elle retourne `False`.
- Addition des valeurs avec deux variantes : à l'aide d'un `reduce` assez simple, ou avec la méthode `sum`. Vous voudrez bien les essayer toutes les deux.
- Division avec le nombre d'éléments pour obtenir la moyenne.
- À la fin, il y a une variante qui calcule directement la moyenne des nombres d'un RDD avec la méthode `mean`.

À noter que les affichages de résultats doivent se faire avec des variables, parce qu'une écriture comme la suivante ne fonctionne pas bien ; le message et le résultat sont dissociés.

```
print "hauteur moyenne des arbres =", hauteurs_ok.mean()
```

2.3. Genre du plus grand arbre

On passe à `tp4_2_3.py` qui affiche le genre du plus grand arbre.

Le principe est de construire des paires (clé, valeur), avec ici la hauteur des arbres comme clé et leur genre en tant que valeur. Ensuite, on classe les paires par ordre de clé décroissante grâce à `sortByKey` et on garde seulement la première paire.

On voit ici une technique extrêmement importante, la manipulation de paires (clé, valeur). C'est comme avec MapReduce sur Yarn. L'algorithme repose sur le choix des clés et valeurs à associer. Il y a de nombreuses méthodes, pour information (elles ne sont pas à apprendre, juste savoir qu'il y a du choix) :

- `aggregateByKey`, `combineByKey`, `countByKey`, `foldByKey`, `groupByKey`, `reduceByKey`, `sortByKey`, `subtractByKey` pour travailler avec les clés,
- `countByValue`, `mapValues` pour gérer les valeurs,
- une fonction pour extraire uniquement les clés : `keys`,
- deux fonctions pour accéder aux valeurs : `lookup` et `values`.

Il y a une variante pour faire le tri, `sortBy`. On lui fournit une lambda qui extrait la valeur à utiliser pour le tri. Ici, on extrait la hauteur des paires.

La fin du script montre une autre méthode, bien plus efficace qu'un tri, utilisant `max`. Le paramètre à lui fournir est une fonction ou lambda qui retourne la valeur dont on recherche le maximum (comme pour `sortBy`). Il y a aussi une méthode `min`.

2.4. Nombre d'arbres de chaque genre

Pour finir, `tp4_2_4.py` affiche le nombre d'arbres de chaque genre.

Le principe est de construire une paire (genre, 1) par arbre du fichier, puis de cumuler les valeurs genre par genre avec `reduceByKey`.

Sauriez-vous le faire avec la méthode `countByKey` ?

3. Stations météo

Vous allez travailler avec le fichier `hdfs:/share/noaa/isd-history.txt`. Pour mémoire, voici ses champs :

offset	taille	exemple	signification
0	6	`225730`	USAF = Air Force Datsav3 station number
7	5	`99999`	WBAN = NCDC WBAN number
13	29	`LESUKONSKOE`	Station name
43	2	`US`	FIPS country ID (pays)
48	2	`KS`	ST = State for US stations
51	4	`LFRO`	CALL = ICAO call sign (code aéroport)
57	7	`+64.900`	LAT = Latitude in decimal degrees
65	8	`+045.767`	LON = Longitude in decimal degrees
74	7	`+0071.0`	ELEV = Elevation in meters (altitude)
82	8	`19590101`	BEGIN = Beginning Period Of Record (YYYYMMDD)
91	8	`20140206`	END = Ending Period Of Record (YYYYMMDD)

Le fichier `station.py` extrait les valeurs nécessaires. Remarque : le constructeur n'a pas besoin de séparer en champs car il n'y a pas de séparateur.

N'oubliez pas de changer le paramètre `--py-files` lors du lancement des programmes.

Ensuite, vous allez rencontrer un problème avec les 22 premières lignes. Ce sont des titres à ignorer dans les traitements. Il faut savoir que les lignes correctes commencent par un chiffre. Voici donc comment lire les données :

```
brut = sc.textFile("hdfs:/share/noaa/isd-history.txt")
brut = brut.filter(lambda ligne: len(ligne)>0 and ligne[0] in '0123456789')
```

3.1. Répartition des stations par hémisphère

En se basant sur la latitude, il faudrait compter le nombre de stations dans l'hémisphère nord et dans l'hémisphère sud. Pour cela,

- Il faut définir une fonction (isolée, pas une *lambda*) qui retourne `"nord"` ou `"sud"` selon la latitude. Vous devez ignorer les stations dont la latitude est inconnue.
- Il faut appeler cette fonction sur chaque station afin de construire des paires `("nord", 1)` ou `("sud", 1)`
- Il faut additionner les valeurs des paires ayant la même clé.

3.2. Plus grande période de mesures

On voudrait savoir quelle est l'identifiant USAF et le nom de la station qui a le plus grand écart d'années entre le début de ses mesures et la fin. Par exemple, la station 225730 LESUKONSKOE va de 1959 à 2014, ça fait 55 années mais ce n'est pas la plus longue période.

Pour cela,

- L'année se trouve au début des champs BEGIN et END. Il suffit de la convertir en entier.
- Il faut d'abord garder les stations qui ont une année de début et une année de fin.
- Il faut construire des paires (fin-début, usaf+" "+nom)
- Il faut classer ces paires dans l'ordre des années d'écart décroissantes, puis garder la première.

Il y a aussi les méthodes `min` et `max` auxquelles on passe une fonction ou *lambda* qui permet de spécifier sur quoi on recherche l'extremum. L'exemple de la doc n'est pas clair (usage de la fonction `str` pour comparer les valeurs d'une liste). Voici par exemple comment trouver la station la plus haute :

```
stations = ...
plus_haute = stations.max(lambda s: s.altitude())
```

Bien comprendre que la lambda sert à sélectionner la valeur à maximiser, mais c'est le n-uplet entier qui est retourné.

On peut aussi appliquer cette technique à une liste de paires (clé, valeur). Par exemple, retourner la paire ayant la plus petite valeur :

```
paires = ...
petite_paire = paires.min(lambda (cle,val): val)
```

3.3. Pays ayant le plus de stations

On veut savoir quel est le pays qui a le plus de stations. Le pays est un code sur deux lettres, ex: `FR`. Il faut ignorer les pays incorrects.

A un moment, vous pourrez avoir besoin d'une astuce qui consiste à échanger les clés et les valeurs dans les paires à traiter. Il suffit d'une `lambda (c,v): (v,c)`. Ou alors utiliser `sortBy` avec une lambda qui extrait la valeur.

3.4. Nombre de pays ayant des stations

On veut savoir combien de pays possèdent des stations météo. Pour cela, il faut faire la liste des pays, mais en un seul exemplaire chacun, puis les compter.

Vous pourrez employer la méthode `distinct` ou trouver un moyen avec des paires (clé, valeur).

Le nombre de pays est 248, voir cette page.

4. Horodateurs de Paris

Pour finir, on va travailler avec les relevés des parcmètres. Ce sont des fichiers CSV dans le dossier `hdfs:/share/paris/horodateurs-transactions`.

La structure de ces fichiers avait été donnée dans le TP3 :

champ	exemple	signification
0	8331703	numéro (identifiant) du parcmètre
1	11/08/2014 08:58:51	date de la transaction
2	Rotatif	type d'usager

champ	exemple	signification
3	CB	moyen de paiement
4	2,40	montant en € avec une virgule dans le prix
5	2,00	nombre d'heures payées (fractionnaire, avec une virgule)
6	11/08/2014 08:58:51	début du stationnement
7	11/08/2014 11:00:00	fin du stationnement

Le fichier `horodateur.py` contient la classe `Horodateur` déjà faite. Vous n'avez plus qu'à programmer les scripts de traitement.

Les temps de calcul vont devenir nettement plus importants. Le fichier est très gros. N'hésitez pas à écrire des programmes sans aucune erreur, qui font exactement ce qu'ils doivent dès le début, afin de ne pas perdre du temps.

Si c'est trop lent, ne demandez pas le calcul sur toutes les données, mais seulement l'un des fichiers, par exemple : 

`hdfs:/share/paris/horodateurs-transactions/Transactions_SGCH_20140811_20140820.csv`

4.1. Horodateur qui a rapporté le plus

Il faut afficher le numéro de l'horodateur qui a rapporté le plus d'argent sur la totalité des données.

4.2. Prix moyen par heure de stationnement

Il faut calculer le prix moyen par heure de stationnement à Paris.

4.3. Plus petite durée de stationnement

Afficher la plus courte durée de stationnement ainsi que le montant payé.

4.4. Répartition des paiements par mois

On voudrait savoir quelle somme est récoltée chaque mois par les horodateurs. Il faut parvenir à extraire le mois du champ 1. C'est une chaîne, il suffit d'extraire le mois avec la notation des tranches de Python `[de:à]`.

Il faudrait que les mois apparaissent dans l'ordre croissant.

5. Travail à rendre

Remontez au dessus du dossier `TP4` et compressez-le en `.tar.gz`. Ne vous trompez pas entre le `TP4.zip` qui vous a été fourni et lui. Votre archive doit contenir tous les scripts Python que vous avez faits et modifiés. Déposez-la sur Moodle dans la zone de dépôt prévue.