Travaux pratiques Spark

Jonathan Lejeune

Objectifs

Ce document a pour objectif de vous familiariser avec la programmation Spark en s'inspirant en partie d'exercices de traitement de données vus en TD Map-Reduce.

Prérequis

Vous devez avoir installé la plate-forme Spark et avoir configuré votre espace de travail correctement pour coder et compiler vos programmes scala avec l'API de Spark. Vos programmes seront validés avec JUnit en mode local (i.e. `setMaster("local[*]")`). Vous aurez également besoin de la librairie scala DataGenerator qui vous a été fournie dans les ressources globales de l'UE (vous pouvez soit inclure les sources dans votre environnement ou alors produire un jar et ajouter ce jar à votre environnement de travail). Cette librairie permet de générer des fichiers de données pour les tests. Nous utiliserons le système de fichiers local pour le stockage des fichiers d'entrée et de sortie (il est donc inutile de démarrer le HDFS).

Informations complémentaires

- La sortie de logs de l'application spark affichée sur le terminal peut être très verbeuse et peut noyer éventuellement vos affichage de debug. Il est possible de limiter ces logs en ajoutant dans votre code :

  ```
  Logger.getLogger("org.apache.spark").setLevel(Level.OFF)
  ```

  Attention de bien importer le `Logger` et le `Level` du package `org.apache.log4j`
- Les solutions aux exercices ne doivent pas se baser sur les appels aux actions `collect` ou `take` des RDD. Ces actions sont réservées à vos phases de debug et doivent être supprimées par la suite

Exercice 1 – StereoPrix

StereoPrix est une entreprise de grande distribution et souhaite faire des statistiques sur les ventes. Ces données sont stockées dans des fichiers textes. Chaque ligne d'un fichier correspond à la vente d'un produit et on peut y trouver des informations comme :

- la date et l'heure de vente
- le nom du magasin où le produit a été vendu
- le prix de vente
- la dénomination du produit
- la catégorie du produit (ex : fruits et légumes, électroménager, jouet, ...)

Le package de travail de cet exercice est `datacloud.spark.core.stereoprix`. Les fichiers d'entrée sont des fichiers textes et chaque ligne respecte le format suivant :

```
JJ_MM_AAAA_hh_mm magasin prix produit categorie
```

Dans cet exercice, chaque fonction à programmer est complètement indépendante d'une autre. Elle doivent instancier un **SparkContext** et réaliser le traitement demandé. Comme nous sommes en mode local et qu'il ne peut exister qu'un seul `SparkContext` actif dans une JVM, il faudra penser à fermer celui-ci à la fin de la fonction grâce à la méthode `stop`.

Question 1

Créer un `object` scala `Stats`.

Question 2

Dans l'objet `Stats`, écrire une fonction `chiffreAffaire` qui pour l'url des fichiers d'entrée (=une chaîne de caractère) et une année (= un `Int`) données calcule *le chiffre d'affaire de l'entreprise pour cette année*. Le résultat de cette fonction sera un entier.

Question 3

Tester avec la classe Junit suivante fournie dans les ressources de TP :

`datacloud.spark.core.stereoprix.test.ChiffreAffaireTest`

Question 4

Dans l'objet `Stats`, écrire une fonction `chiffreAffaireParCategorie` qui pour l'url des fichiers d'entrée et l'url des fichiers de sortie, calcule et stocke vers l'url de sortie *le chiffre d'affaire généré pour chaque catégorie*. Le format de sortie attendu est de type texte et les lignes respecteront le format suivant :

```
categorie:chiffreAffaire
```

Question 5

Tester avec la classe Junit suivante fournie dans les ressources de TP :

`datacloud.spark.core.stereoprix.test.ChiffreAffaireParCategorieTest`

Question 6

Dans l'objet `Stats`, écrire une fonction `produitLePlusVenduParCategorie` qui pour une url de fichiers d'entrée et une url de sortie, calcule et stocke vers l'url de sortie *le produits le plus vendu par catégorie*. Le format de sortie attendu est de type texte et les lignes respecteront le format suivant :

```
categorie:produitLePlusVendu
```

Question 7

Tester avec la classe Junit suivante fournie dans les ressources de TP :

`datacloud.spark.core.stereoprix.test.ProduitLePlusVenduParCategorieTest`

Exercice 2 – Hit Parade de Last.fm

Last.fm est un site web de radio en ligne et de musique communautaire offrant différents services à ses utilisateurs comme par exemple l'écoute ou le téléchargement gratuit de musiques. Il existe plus de 25 millions d'utilisateurs qui utilisent Last.fm tous les mois générant ainsi beaucoup de données à traiter. L'analyse de données la plus courante se fait sur les informations que les utilisateurs transmettent au site lorsqu'ils écoutent une musique. Grâce à ces informations, il possible de produire entre autres des hit-parades. Un titre peut être écouté de deux manières différentes par un utilisateur :

- soit en local sur son propre ordinateur et les informations d'écoute sont envoyées directement au serveur de Last.fm
- soit via une web radio sur le site même. Dans ce cas, l'utilisateur a la possibilité de passer le titre sans l'écouter.

Le système logue pour chaque couple utilisateur-titre :

- le nombre de fois où l'utilisateur a écouté le titre en local
- le nombre de fois où l'utilisateur a écouté le titre en ligne
- le nombre de fois où l'utilisateur a passé le titre sans l'écouter entièrement (=skip).

Le tableau ci-dessous donne un exemple des données maintenues par le système

UserId	**TrackId**	**LocalListening**	**RadioListening**	**Skip**
U111115	T222	0	1	0
U111113	T225	3	0	0
U111117	T223	0	1	1
U111115	T225	2	0	0
U111120	T221	0	0	1

Les données sont stockées dans des fichiers de logs textuels et chaque ligne respecte le format suivant :

```
UserID TrackID LocalListening RadioListening Skip
```

L'objectif de l'exercice est de produire un hit parade à partir des fichiers de logs. Le critère de classement se basera sur deux critères :

- **Critère 1** : le nombre total de personnes différentes qui l'ont écouté au moins une fois (en local ou en radio)
- **Critère 2** : le nombre total de fois où il a été écouté (en local et en radio) moins le nombre total de fois où il a été passé sans écoute.

Le critère 1 est le critère de choix le plus prioritaire. Si deux titres on des valeurs différentes, le titre ayant la plus grande valeur aura une meilleure place dans l'ordre du hit parade. En cas d'égalité du critère 1, la même politique s'applique sur le critère 2. Enfin en cas d'égalité du critère 2, on triera les titres par ordre lexicographique

Dans l'exemple, on doit donc obtenir :

TrackId	**#listener**	**score**
T225	2	5
T222	1	1
T223	1	0
T221	0	-1

Question 1

Dans le package `datacloud.spark.core.lastfm`, créer un object scala `HitParade`

Question 2

Afin de typer les tracks et les users dans les données que l'on va manipuler, créer dans cet object, deux case classes `TrackId` et `UserId` qui prennent chacune un attribut `id` de type String.

Question 3

Dans l'ensemble des données d'entrée il est possible que deux lignes aient le même couple (UserId,TrackId). Les doublons sont donc possibles. Dans l'object `HitParade` écrire une fonction `loadAndMergeDuplicates` qui prend un context spark (déjà initialisé) et l'url des données d'entrée produit un `RDD[((UserId,TrackId),(Int,Int,Int))]`. Le RDD résultant contiendra l'ensemble des données d'entrée après avoir fusionner les données des éventuels doublons (UserId,TrackId). Les trois entiers de la partie de droite d'un élément correspondent respectivement au nombre d'écoutes en local, le nombre d'écoute en radio et le nombre de skip par l'utilisateur.

Question 4

Dans l'object `HitParade` écrire une fonction `hitparade` qui prend en paramètre un `RDD[((UserId,TrackId),(Int,Int,Int))]` (qui résultera de `loadAndMergeDuplicates`) et produit un `RDD[TrackId]` classé dans l'ordre décrit précédemment (élément 0 = le gagnant du hit parade, élément 1 = seconde place, etc.).

Question 5

Tester avec la classe Junit suivante fournie dans les ressources de TP :

`datacloud.spark.core.lastfm.test.HitParadeTest`

Exercice 3 – Calcul matriciel

Dans cet exercice notre but final est d'écrire une classe scala utilisant les RDD pour faire des opérations sur des matrices d'entiers. L'avantage de spark est que l'on peut tirer parti de ses capacités de parallélisation afin de pouvoir passer à l'échelle lorsque la taille des matrices considérées sont grandes. Vous aurez besoin de la classe `VectorInt` qui vous avez programmée lors du TP sur scala. En se basant sur la classe `VectorInt`, nous allons coder une classe qui représentera une matrice répartie sur un cluster à l'aide de Spark. Les questions qui suivent vous aideront à enrichir progressivement les services qu'offre cette classe.

Question 1

Déclarez une classe `MatrixIntAsRDD` qui aura un seul attribut immuable `lines` de type `RDD[VectorInt]` qui sera renseigné à l'instanciation de la classe. On représente ainsi une matrice comme étant un ensemble de vecteurs, où chaque vecteur représente une ligne de la matrice. Vous y ajouterez la méthode suivante qui permet d'avoir une représentation String de la matrice :

```
override def toString={
    val sb = new StringBuilder()
    lines.collect().foreach(line=> sb.append(line+"\n"))
    sb.toString()
  }
```

Question 2

Dans un objet compagnon de la classe `MatrixIntAsRDD` (un `object` portant le même nom et déclaré dans le même fichier), écrire une méthode de conversion implicite d'un `RDD[VectorInt]` vers une `MatrixIntAsRDD`. Pour que cette conversions soit connu dans la classe, il est impératif de déclarer l'objet compagnon avant la classe.

Les matrices que nous allons manipuler sont donc des ensembles de vecteurs ordonnés par indice de ligne. Nous souhaitons construire une instance de `MatrixIntAsRDD` à partir d'un fichier texte. Dans le fichier texte, les éléments d'une ligne sont séparés par un espace. Nous devons assurer que toutes les matrices ayant le même nombre de lignes soient partitionnées exactement de la même manière. En résumé deux matrices ayant le même nombre de lignes doivent :

- avoir le même nombre de partitions
- que chaque partition correspondante entre les deux matrices, doivent avoir le même nombre d'éléments.
- être triées par ordre de vecteur croissant, c'est à dire que le n-ième élément dans le RDD doit être la n-ième ligne de la matrice

Question 3

Dans l'objet compagnon, écrire une fonction `makeFromFile` qui permet de créer une nouvelle instance de `MatrixIntAsRDD` à partir du fichier texte. Cette fonction prend trois arguments :

- l'URL du fichier texte d'entrée de la matrice
- le nombre de partitions sur lequel le RDD résultant sera distribué
- le contexte spark à utiliser (supposé déjà instancié et correctement initialisé)

Pour assurer une répartition déterministe des éléments sur l'ensemble des partitions, vous pourrez utiliser la transformation `sortBy`. Cette transformation répartit de manière déterministe sur un nombre de partitions donnés les éléments d'un RDD en fonction d'un ordre sur une clé. Nous supposerons que l'utilisateur de la fonction `makeFromFile` donnera le même nombre de partitions pour chaque fichier de matrice ayant le même nombre de lignes.

Attention : depuis la version 2.4 de Spark le nombre de partitions produites par `sortBy` est borné par le nombre d'éléments présents dans le RDD. Il faut donc s'assurer que le nombre de partition passé en paramètre soit inférieur au nombre d'éléments du RDD (ce qui est le cas si on considère des matrice de taille importante). Pour vous aider, voici l'enchaînement des transformations attendues dans cette fonction :

- 1er rdd (`RDD[String]`) : création à partir du fichier (`textfile`)
- 2ème rdd (`RDD[Array[Int]]`) : on transforme chaque ligne du 1er rdd en array de Int (`map`)
- 3ème rdd (`RDD[VectorInt]`) : on transforme chaque élément du 2ème rdd en VectorInt (`map`)

A partir de maintenant nous souhaitons nous assurer que la répartition des éléments soit la même quelque soit le fichier d'entrée :

- 4ème rdd (`RDD[(VectorInt,Long)]`) : lier chaque élément du 3eme rdd avec son index (`zipWithIndex`)
- 5ème rdd (`RDD[(VectorInt,Long)]`) : trier le 4ème rdd en fonction de l'index croissant (`sortBy`)

- 6ème rdd (`RDD[VectorInt]`) : enlever la partie droite de chaque élément pour ne garder que le vecteur (`map`)
- enfin construire et retourner une MatrixIntAsRDD à partir du 6ème rdd.

Question 4

Tester avec la classe Junit suivante fournie dans les ressources de TP :

`datacloud.spark.core.matrix.test.MakingTest`

Question 5

Écrire dans la classe `MatrixIntAsRDD` , deux méthodes `nbLines` et `nbColumns` qui renvoient respectivement le nombre de lignes et le nombre de colonnes de la matrice. Y ajouter également la méthode `get(i:Int,j:Int):Int` qui renvoie l'élément à l'indice i, j où $0 \leq i < nb\_lines$ et $0 \leq j < nb\_columns$.

Question 6

Tester avec la classe Junit suivante fournie dans les ressources de TP :

`datacloud.spark.core.matrix.test.GetterTest`

Question 7

Écrire dans la classe `MatrixIntAsRDD` la méthode `equals(a:Any):Boolean` qui teste si deux matrices sont égales.

Question 8

Tester avec la classe Junit suivante fournie dans les ressources de TP :

`datacloud.spark.core.matrix.test.EqualsTest`

Question 9

Écrire dans la classe `MatrixIntAsRDD` la méthode `+(other: MatrixIntAsRDD): MatrixIntAsRDD` qui additionne deux matrices.

Question 10

Tester avec la classe Junit suivante fournie dans les ressources de TP :

`datacloud.spark.core.matrix.test.PlusTest`

Question 11

Écrire dans la classe `MatrixIntAsRDD` la méthode `transpose():MatrixIntAsRDD` qui calcule la transposée d'une matrice, c'est à dire que la ligne i devient la colonne i et la colonne j devient la ligne j.

Question 12

Tester avec la classe Junit suivante fournie dans les ressources de TP :

`datacloud.spark.core.matrix.test.TransposeTest`

Question 13

La multiplication de deux matrices A par B nécessite que le nombre de colonnes de A soit égale au nombre de lignes de B. Ainsi si la matrice C est égale à la multiplication de A par B, alors l'élément (i, j) de C correspond à la multiplication la ligne i de A avec la colonne j de B. Exemple :

$$\begin{bmatrix} 1 & 1 & -1 \\ 4 & 0 & 2 \\ 1 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 2 & 1 \\ 3 & -2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 5 & -4 \\ 8 & -2 \\ 2 & -1 \end{bmatrix}$$

On remarque ainsi que $C_{0,0} = ligne0_A * colonne0_B = 1 * 2 + 1 * 3 + (-1) * 0$.
Cependant, ce calcul n'est pas applicable sur un RDD[VectorInt] car les données sont distribuées. La solution que nous proposons est donc la suivante :

- **Étape 1 :** Coupler la colonne i de la matrice A avec la ligne i de la matrice B
- **Étape 2 :** Appliquer le produit dyadique sur chaque couple
- **Étape 3 :** Faire la somme des produits dyadique

Sur notre exemple ceci donnerait

Étape 1	**Étape 2**	**Étape 3**
$\begin{bmatrix}1\\4\\1\end{bmatrix}, \begin{bmatrix}2,-1\end{bmatrix}$	$\begin{bmatrix}2,-1\\8,-4\\2,-1\end{bmatrix}$	
$\begin{bmatrix}1\\0\\0\end{bmatrix}, \begin{bmatrix}3,-2\end{bmatrix}$	$\begin{bmatrix}3,-2\\0,0\\0,0\end{bmatrix}$	$\begin{bmatrix}5&-4\\8&-2\\2&-1\end{bmatrix}$
$\begin{bmatrix}-1\\2\\0\end{bmatrix}, \begin{bmatrix}0,1\end{bmatrix}$	$\begin{bmatrix}0,-1\\0,2\\0,0\end{bmatrix}$	

En suivant cet algorithme, ajouter la méthode `*` à la classe `MatrixIntAsRDD` pour multiplier des matrices.

Question 14

Tester avec la classe Junit suivante fournie dans les ressources de TP :

`datacloud.spark.core.matrix.test.FoisTest`