Programmation fonctionnelle en Big Data : Map-Reduce

Master 2 I2L, 2020/2021

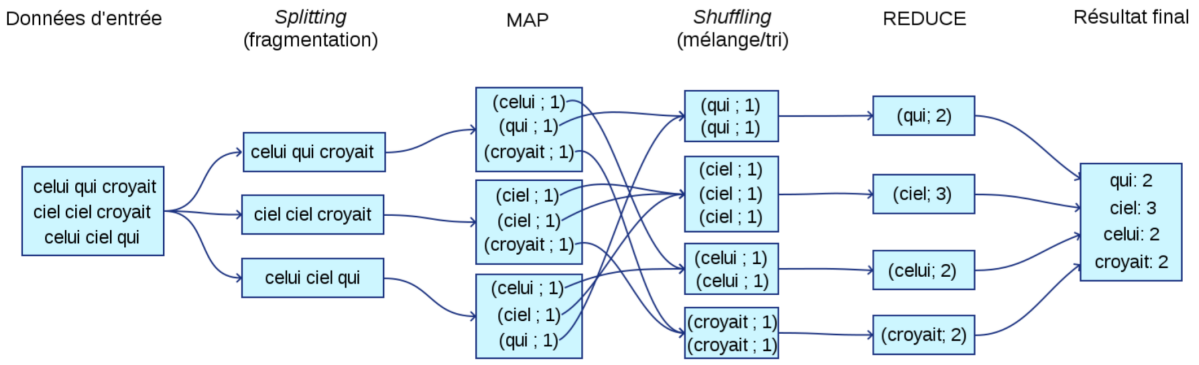
Le but est d'apprendre les algorithmes pour le traitement de données dans le "big data" qui sont basés sur des principes de programmation fonctionnelle. En particulier, cette activité permet de découvrir le principe "map-reduce".

Principe Map-Reduce en big data

Le principe de Map-Reduce (MR) est très bien présenté dans le document de Benjamin Renaut, Tokidev SAS : `https://cours.tokidev.fr/bigdata/` "introduction, map/reduce"

L'algorithme MR permet de traiter en parallèle une quantité importante de données en divisant les données en blocs, puis en appliquant une fonction sur chaque bloc de données à l'aide de différentes machines tout en réduisant les interactions entre machines à une étape simple et précise. Par l'utilisation d'un système de fichiers distribué (HDFS), l'idée de MR est de déplacer le code de traitement plutôt que de déplacer les données. L'algorithme est profondément inspirée par la programmation fonctionnelle puisque les étapes de l'algorithme map-reduce sont issues de la programmation fonctionnelle.

Les 4 étapes de l'algorithme MR sont illustrées à l'aide du schéma suivant extrait des slides cités plus haut :



Les 4 étapes algorithmiques sont :

- split : est une fonction qui divise l'ensemble des données en blocs de données,
- map : est une fonction classique en programmation fonctionnelle qui applique une fonction sur chaque bloc de données. Le résultat de la fonction appliquée à chaque bloc est un ensemble de couples (clé, valeur),
- shuffle : est l'opération qui consiste à regrouper les couples (clé ; valeur) qui ont des clés identiques,
- reduce : est la fonction classique de programmation fonctionnelle (aussi appelée fold) qui réduit un ensemble de données en une donnée finale.

L'étape principale d'interaction entre les données, shuffle, est toujours la mÃªme. Elle est donc définie dans l'algorithme MR. Par contre, pour utiliser un algorithme MR, l'utilisateur doit fournir les éléments et fonctions suivantes :

— le couple (clé, valeur)
— la fonction split : donnée → [donnée, . . . , donnée],
— la fonction map : donnée → [{clé, valeur}, . . . , {clé, valeur}],
— la fonction reduce : [valeur, . . . , valeur] → valeur.

Dans la suite, nous allons coder l'algorithme MR en haskell, mais dans une version séquentielle. Nous allons également développer l'algorithme classique qui consiste à compter la fréquence d'apparition des mots dans un texte.

1 The big exemple : Fréquence d'apparition des mots

Nous allons développer en mÃªme temps l'algorithme MR en haskell et résoudre le dénombrement des mots d'un texte à l'aide de l'algorithme MR qui est l'équivalent du "hello world" pour hadoop.

1.a. Découvrir le problème qui consiste à compter les mots dans un texte à partir des slides précédents (3-5, slide 25 et suivants).

1.b. Nous allons construire l'algorithme MR par étape en introduisant successivement les fonctions split, map, shuffle, reduce. Ecrire la première étape minimale suivante :
```
>  map_reduce_1 the_split the_map the_reduce dat = the_split dat
```

1.c. Les données sont des chaines de caractères représentées en Haskell par des listes de char. Ecrire le programme principal qui permet d'exécuter `map_reduce_1` sur la donnée suivante : `dat = "la la la la\nvoila le temps\ndes cerises\ndes bonnes cerises\ndes oiseaux sont la aussi"`.

1.d. Définir la fonction `cut :: Char -> [Char] -> [ [Char] ]` qui permet de découper un texte selon la séparateur `sep`. Par exemple, `cut '\n' data` doit donner la liste suivante : `["la la la la","voila le temps","des cerises","des bonnes cerises","des oiseaux sont la aussi"]`. Définir la fonction `wc_split :: [Char] -> [ [Char] ]` à partir de la fonction `cut` qui découpe un texte en liste de lignes.

1.e. Définir une fonction `wc_map :: [Char] -> [ ([Char], Int) ]` qui résout le problème initial. Par exemple, `wc_map "la la la la"` pour résultat `[("la",1),("la",1),("la",1),("la",1)]`.

1.f. Définir la fonction `map_reduce_2 the_split the_map the_reduce dat` qui introduit l'étape map. Par exemple, appliqué à `dat` nous devons obtenir : `[("la",1),("la",1),("la",1),("la",1),("voila",1),("le",1),("temps",1),("des",1),("cerises",1),("des",1),("bonnes",1),("cerises",1),("des",1),("oiseaux",1),("sont",1),("la",1),("aussi",1)]`

1.g. Définir la fonction `shuffle :: Ord a => [ (a,b) ] -> [ (a,[b]) ]` (et la troisième version de `map_reduce_3`) qui doit regrouper ensemble les couples qui ont la mÃªme clé. Par exemple, dans cette étape nous devons obtenir : `[("aussi",[1]),("bonnes",[1]),("cerises",[1,1]),("des",[1,1,1]),("la",[1,1,1,1,1]),("le",[1]),("oiseaux",[1]),("sont",[1]),("temps",[1]),("voila",[1])]`. Vous pouvez utiliser la fonction prédéfinie `sortBy` du module `Data.List`.

1.h. Définir la fonction `wc_reduce :: [ Int ] -> Int` qui réduit la liste associée à chaque clé à son nombre d'élément. Par exemple, `wc_reduce [1, 1, 1]` a pour valeur 3.

1.i. Définir enfin la fonction `map_reduce`.

2 Graphe social

En suivant l'autre exemple donné par Benjamin Renaut, nous cherchons à calculer la liste des amis communs à deux personnes dans un réseau social. Les relations sociales sont codées de la manière suivante :

```
A>B C D
B>A C D E
C>A B D E
D>A B C E
E>B C D
```

Dans cet exemple, A est en relation avec B, C et D.

Nous cherchons à obtenir le résultat suivant :

```
"A-B": "C, D"
"A-C": "B, D"
"A-D": "B, C"
"B-C": "A, D, E"
"B-D": "A, C, E"
"B-E": "C, D"
"C-D": "A, B, E"
"C-E": "B, D"
"D-E": "B, C"
```

2.a. Développer un algorithme de map-reduce pour résoudre ce problème.