24 NOVEMBRE 2016

RAPPORT DE PROJET SHAVADOOP

DEVELOPPEMENT EN JAVA D'UN WORD COUNT

AL ANI MOHAMED
BREHELIN ALEXANDRE

À l'attention de Rémi SHARROCK, professeur référent

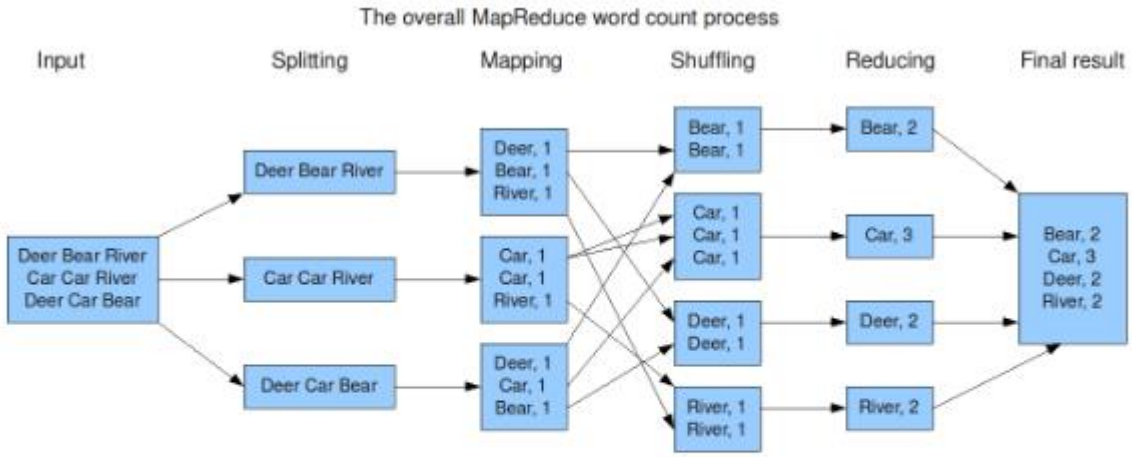
Table des matières

I. Introduction

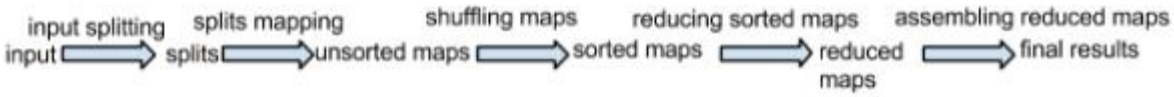
Dans le cadre du cours de systèmes répartis, nous avons effectué un programme dont l'objectif est, en nous basant sur le modèle de Hadoop et du Framework MapReduce, de construire « from scratch » un programme en Java permettant de compter les mots dans un fichier grâce à plusieurs machines à notre disponibilité et de manière parallèle.
Ainsi, nous diviserons notre fichier et effectuerons les calculs nécessaires sur plusieurs machines connectées au réseau informatique de l'école Télécom Paristech.

Nous avons à notre disposition les pc de l'école qui sont connectés en réseau et sur lesquels nous pouvons nous connecter en SSH sans nécessité de login/mot de passe.

Nous suivrons ce modèle :



Définitions (à partir du schéma avec les blocs bleus):



II. Fonctionnement du programme

Le programme peut être découpé en 6 étapes :

- Phase de **Démarrage**
- Phase de **File Splitting**
- Phase de **Mapping**
- Phase de **Shuffling**
- Phase de **Reducing**
- Phase d'Assembling

a) Phase de Démarrage :

Dans un premier temps, nous chargeons notre liste de machine que nous avons stockées en format txt dans un répertoire. Nous testons une par une les machines en nous connectant en ssh via le processBuilder et en testant une commande bash « echo OK» et nous vérifions que nous avons bien l'output « OK » de la machine. Nous gardons les machines connectées dans une ArrayList ainsi que dans un fichier txt que nous stockons afin d'éviter de faire ce test plusieurs fois.

Ainsi, nous avons une classe « ConnectedMachines » qui contient une méthode qui prend en entrée le fichier des machines, le parcours ligne par ligne, vérifie pour chaque machine la connexion SSH et renvoie en sortie une ArrayList de String : les machines connectées au réseau.
Nous parallélisons le test sur notre machine pour tester chacune des machines grâce à des Threads et ainsi gagner en performance.
Pour tester la connexion SSH, nous faisons appel à la classe « TestConnectionSSH » avec un TIMEOUT de 3 secondes pour éviter les temps d'attente trop longs qui peuvent ralentir notre programme. Cette classe contient une méthode run() qui contient le ProcessBuilder qui permet de lancer des commandes process en micro Batch. Nous lançons notre commande « echo OK » et vérifions grâce à la classe « LecteurFlux » le retour de la machine.

b) Phase de File Splitting

Pour le file Splitting, nous utilisons la classe « FileSplit » qui en plus de diviser le fichier d'entrée, va "nettoyer" chacun des mots présents dans notre fichier.
Ainsi, une méthode va prendre le fichier en entrée, le lire ligne par ligne, va nettoyer les chaînes de caractères (normalise le mot en retirant les caractères spéciaux, retire les espaces, les accents…) et diviser le fichier. Nous divisons le fichier en fonction du nombre de machines à notre disposition et du nombre de mots dans le fichier. Nous effectuons la division suivante pour savoir combien de mots chaque fichier aura : nombre_total_de_mots / nb_machines. Nous créons Sn fichiers avec n le nombre de machines à notre disposition.
Ainsi nous avons 1 fichier S et 1 fichier UM par machine.

c) Phase de Mapping

La phase de mapping consiste à créer des fichiers UMX qui contiennent chaque mot différent de chaque fichier SX. Pour lancer cette phase, nous faisons appel à une classe « SLAVESHAVADOOP ».
Nous lions alors, dans notre main, chaque machine à un fichier S /UM qui sera créé et nous lançons via la classe « LaunchSlave » le jar « SLAVESHAVADOOP » sur une machine correspondante. Nous créons un Thread que nous démarrons via la commande start(). Le calcul des UM sera alors calculé en parallèle sur les machines distantes.
La classe « LaunchSlave », qui hérite de la classe Thread, contient une méthode run() qui lance une méthode launch(). Cette méthode, récursive si nous rencontrons une erreur, lance un process builder contenant une machine ainsi qu'une commande. Dans ce cas, la commande sera le lancement du jar du Slave. Nous récupérons l'output dans un attribut de la classe.

La classe SLAVESHAVADOOP contient une méthode main qui prend plusieurs arguments en entrée. En premier, nous avons le chemin vers lequel celle-ci va créer les fichiers, le mode d'exécution modeSXUMX pour le mapping et le mode UMXSMX pour le shuffling, reducing que nous verrons plus tard.
Pour le mode SXUMX, nous lisons le fichier passé en argument, nous le parcourons et « printons » les mots lorsque ceux-ci sont uniques. Nous écrivons aussi chaque mot dans un fichier Umx.
Après la phase de Mapping, nous récupérons l'output de nos threads dans une Hashmap (mot → UM)

d) Phase de Shuffling & Reducing

Dans notre classe main, nous itérons sur les clés de notre Hashmap. Pour chaque mot, nous lançons un nouveau thread launchSlave qui va créer les fichiers SMX et RMX. Nous aurons un fichier SM et RM par mot. Cette fois ci, c'est le mode UMXSMX que nous étudions.
Nous lisons, pour chaque ligne du fichier UMX le mot, que nous comparons à notre mot passé en argument. Si égalité il y a, nous écrivons dans notre fichier SMX le mot, suivi de « 1 » puis nous incrémentons un itérateur. À la fin de notre boucle, nous écrivons dans notre fichier RM le mot ainsi que le nombre d'occurrences.

e) Assembling

Dans cette dernière étape, nous créons un fichier output contenant chaque mot avec le nombre d'occurrence grâce à l'output de notre thread. Nous trions avant d'écrire dans notre fichier en vérifiant si le mot est dans une liste de « stop words » : une liste de mots qui n'ont aucun intérêt à être relevés (de, a, à, le, la ...)

III. Comment lancer le programme

Dans le zip du projet, vous trouverez un dossier « systèmes_répartis ». Ce dossier contient le fichier des machines, un fichier de « stop words », les fichiers exemples du cours et les deux fichiers jar à exécuter « MASTERSHAVADOOP » et « SLAVESHAVADOOP ».

Placez ce fichier dans un répertoire de votre choix.

Placez-vous dans ce répertoire avec un terminal et lancez le jar master en indiquant en premier argument le fichier txt sur lequel vous voulez calculer le nombre de mots.

Exemple, en ligne de commande :

java -jar MASTERSHAVADOOP.jar forestier_mayotte.txt

IV. Exemples de résultats obtenus

Nous avons exécuté notre code sur les exemples du TP. Nous avions 3 fichiers : forestier_mayotte.txt, deontologie_police_nationale.txt et domaine_public_fluvial.txt.

Voici les résultats obtenus pour forestier_mayotte :

```
biens 8
code 5
forestier 5
forestiers 4
partie 4
communes 4
dispositions 4
agroforestiers 4
legislative 3
mayotte 3
gestion 3
proprietaires 2
attachees 2
souscrivent 2
preliminaire 2
benefice 2
presentant 2
accorde 2
livre 2
publiques 2
regis 2
bonne 2
prioritairement 2
aides 2
titre 2
article 2
table 2
garanties 2
volontairement 1
constitue 1
demembrer 1
forestiere 1
matieres 1
unite 1
propriete 1
sommaire 1
engagement 1
agroforestiere 1
```

```
 Tout est fini ! Voici les temps d'exécution

**---------------**----------------**
Temps de Démarrage : 6
Temps de Splitting : 0
Temps de Mapping : 6
Temps de Shuffling + Reducing : 6
Temps de Assembling : 0
Temps total : 19 secondes
nombre total de mots : 38
**---------------**----------------**
```

Les résultats pour Police_nationale :

```
police 30
article 18
nationale 15
titre 12
fonctionnaires 10
autorite 10
fonctionnaire 8
ordre 8
devoirs 7
code 7
commandement 7
ordres 5
faire 5
deontologie 5
cas 4
public 4
personne 4
subordonne 4
donne 4
droits 4
execution 3
respectifs 3
lorsqu 3
fonctions 3
ier 3
responsabilite 3
autorites 3
generaux 3
preliminaire 3
regles 3
```

```
**---------------**----------------**
Temps de Démarrage : 6
Temps de Splitting : 0
Temps de Mapping : 10
Temps de Shuffling + Reducing : 15
Temps de Assembling : 0
Temps total : 32 secondes
nombre total de mots : 279
**---------------**----------------**
```

Les résultats pour domaine_public_fluvial :

```
article 160
bateau 105
immatriculation 74
lieu 68
tribunal 64
titre 45
navigation 42
domicile 42
bureau 41
code 40
creanciers 39
proprietaire 37
juge 36
inscription 36
saisie 34
delai 34
nom 34
commerce 34
date 32
prix 32
certificat 32
inscriptions 32
```

```
 Tout est fini ! Voici les temps d'exécution

**---------------**----------------**
Temps de Démarrage : 6
Temps de Splitting : 0
Temps de Mapping : 16
Temps de Shuffling + Reducing : 51
Temps de Assembling : 0
Temps total : 74 secondes
nombre total de mots : 970
**---------------**----------------**
```

V. Problèmes, résolutions, optimisations

- Mapping : FileSplitting pas optimal. Au départ, nous créions un fichier S par ligne. Ce qui voulait dire que nous avions plusieurs fichiers S vides et d'autres non uniformément répartis. Traiter des fichiers vides voulait dire que nous construisions un Thread pour rien et cela, en plus d'être inutile, ralentissait l'exécution. Notre première idée était de lancer un process par machine en divisant la taille de notre fichier original par notre nombre de machines. Or, travailler avec des Bytes était compliqué parce que cela coupait des mots en 2. Il aurait fallu vérifier la présence d'espace à la fin et au début de chaque fichier et concaténer les chaines entre elles. Nous avons donc décidé de compter le nombre de mots dans notre fichier et de le diviser par le nombre de machines que nous avons pour avoir des fichiers S ayant tous le même nombre de mots.

- Shuffling : Pour chaque mot, nous lançons un process. Cela n'est pas optimal car cela « bombarde » les machines qui ne peuvent traiter plus de tâches que leurs nombre de cœur. Pour pallier à ce problème, il faudrait lancer 1*nombre_de_coeur thread par machine. Il faudrait ainsi concaténer plusieurs lancements de jar dans launchSlave.
Aussi, beaucoup de process nous renvoyaient des erreurs « ssh-exchange » ou « timeout ». Pour pallier à cela, nous avons construit une fonction récursive qui relance le slave en cas d'erreur. Malheureusement, certains process continuent de mourir
- Shuffling & Reducing : L'étape de création des SMx n'a aucun intérêt dans notre programme puisque l'on calcule au sein de la même procédure les RMx. Dans le cas de gros fichiers, il serait probablement plus optimal de séparer les deux taches, et pourquoi pas proposer des traitements de plusieurs SMx à la fois ce qui éviterai de créer autant de RMx que de nombres de mots.

- Normalisation du texte : Dans les fichiers de sortie, nous obtenons beaucoup de mots parasites, des caractères spéciaux ou des mots « bizarres ». Nous avons donc décidé de gérer cela au niveau de notre fileSplitting où nous normalisons le texte.

- Fichiers parasites : Lorsque nous n'utilisons plus certains fichiers créés « RM », « SM »… Nous les supprimons avec une commande UNIX, en lançant sur les slaves.