IP PARIS

SLR207: Project

Implementation of Hadoop MapReduce from scratch in Java.

Author:
FACUNDO RAIME, Frank Enrique

Email:
frank.facundo@telecom-paris.fr

April 22, 2020

1 Introduction

MapReduce est un patron d'architecture créée par Google, dans laquelle on exécute des tâches en parallèle dans une architecture de software distribué. Le but est de faire un traitement dans des très grands fichiers, ceux-ci peuvent être de l'ordre des téraoctets par exemple.

Dans ce projet le but sera d'utiliser le patron d'architecture MapReduce pour réaliser un comptage des mots dans un grand fichier, cette architecture est décrite dans la Figure 1.

D'abord on montrera quelques résultats des algorithmes de comptage en séquentielles ensuite on étudiera quelques outils des réseaux que l'on utilisera ensuite dans notre projet.

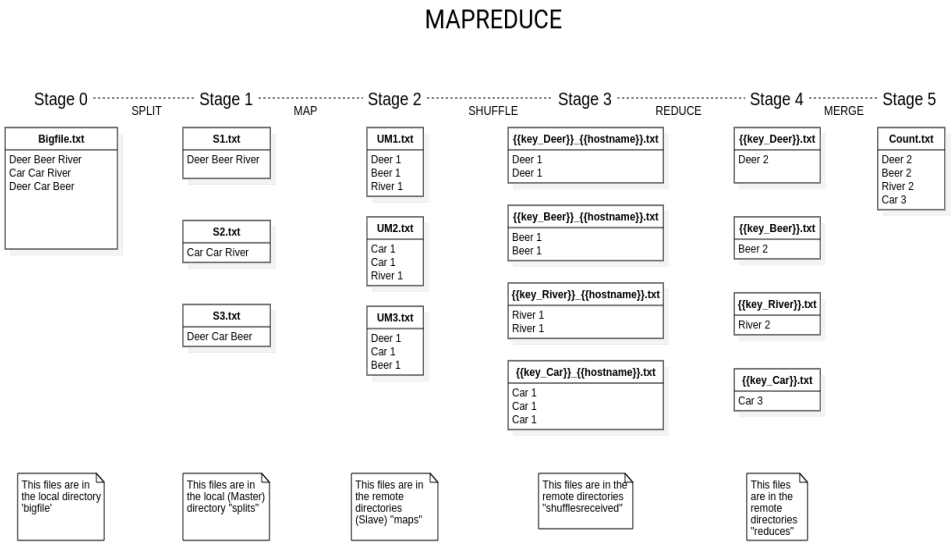

Figure 1: Architecture of MapReduce

2 Comptage séquentiel

En ce qui concerne le programme qui conte la quantité de chaque mot d'un grand fichier de manière séquentiel et ensuite fait un tri selon la quantité d'occurrences des mots. Le code se trouve dans le projet Java "Hadoop_MapReduce" dans le Java package "hadoop".

Pour faire des tests de rendement, on a pris les fichiers du "Codes en vigueur de la République Française", ceux-ci sont disponibles sur le dépôt Github suivante:

- https://github.com/legifrance/Les-codes-en-vigueur

Les tests ont étés faits dans un ordinateur du type "Lenovo YOGA 900-13ISK2" avec un processor "Intel(R) Core(TM) i7-6560U CPU @ 2.20GHz" et avec un SSD Samsung "NVMe SSD Controller SM981/PM981"

Donc les résultats sont les suivantes :

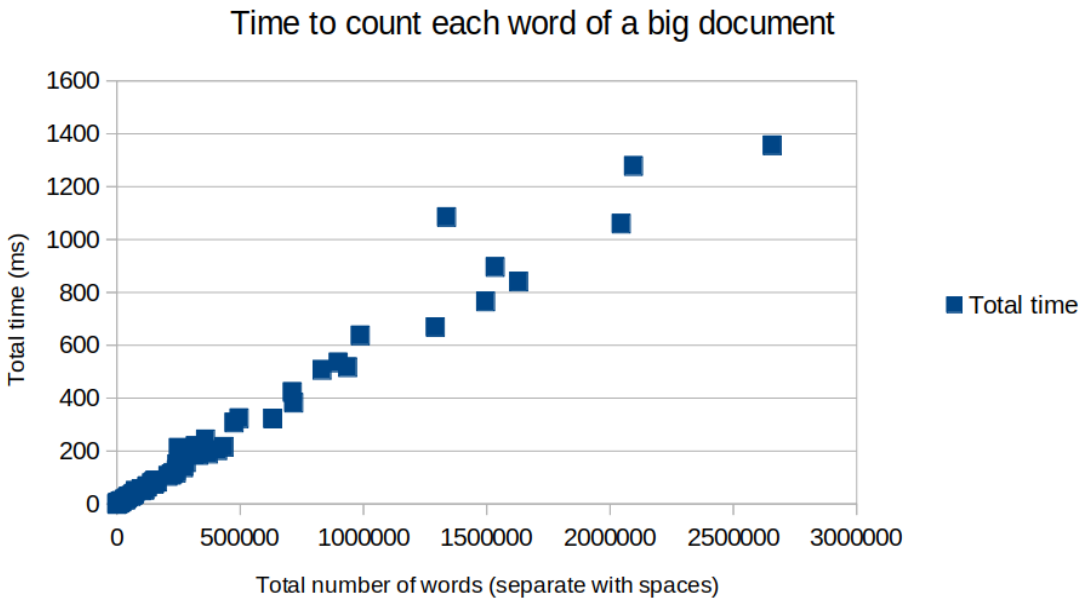

Figure 2: Le temps du comptage d'un grand fichier.

Une remarque importante est que ces fichiers ont une taille qui va de 1.4kB jusqu'à 18.1 MB.

Ensuite on a travaillé avec un plus gros fichier de nom "CC-MAIN-20170322212949-00140-ip-10-233-31-227.ec2.internal.warc.wet", ce fichier est un extrait de toutes les pages internet transformées au format texte brut (format WET), qui a une taille de 398.8MB, notre algorithme nous a donné comme résultat un temps total de traitement de 51436 millisecondes avec un nombre total de mots égal à 48082756 mots.

3 Commandes Linux sur le sujet de reseaux

Pour pouvoir travailler dans et avec plusieurs ordinateurs on n'utilisera que des connexions SSH. Pour cela on citera les commandes plus outils.

Obtenir le nom court de votre ordinateur :

- hostname -s

Obtenir le nom long de votre ordinateur :

- hostname -f

Obtenir des informations réseaux avec une interface graphique ou d'autres commandes :

- network-admin
- cat /etc/hostname

Obtenir les adresses IP de votre ordinateur :

- ifconfig
- (IPv4) curl ifconfig.me
- (IPv6) curl ifconfig.co

Obtenir les adresses IP à partir d'un nom d'ordinateur :

- nslookup <NOM>
- dig

Obtenir le nom d'un ordinateur à partir de l'adresse IP :

- nslookup <IP>
- dig

Vérifier la connexion aux machines de Télécom Paris

- ping ssh1
- ping ssh1.enst.fr
- ping 137.194.47.131

Si l'on veut se connecter depuis l'extérieur il faut configurer une connexion VPN, le plus simple est d'utiliser OpenVPN.

Commandes pour calculer en ligne de commande :

- expr 2 + 3
- bc «< 2 + 3
- echo $((2 + 3))
- python -c "print(2 + 3)"

Commandes pour calculer en ligne de commande à distance :

- ssh ffacundo@ssh.enst.fr expr 1 + 1

Copier une clé publique vers une autre machine :

- ssh-copy-id username@remote_host

Exécution d'un programme Java en local :

- java -jar slave.jar

Exécution d'un programme Java à distance :

- ssh ffacundo@tp-tests.enst.fr java -jar /tmp/ffacundo/slave.jar

4 Architecture Master-Slave

Pour cette architecture nous avons un Master qui va diriger les Slaves qui seront d'autres machines, faisant de ce système une architecture répartie.

D'un côté on aura le Master qui exécute des commandes en ligne de commande grâce à la Class "processBuilder" de Java. Ces commandes utiliseront la commande "ssh" du système Linux pour pouvoir se communiquer avec les machines Slaves. D'un autre côté les Slaves exécutent des tâches en local dans un premier moment, mais ensuite ils échangeront des fichiers, ce processus d'échange sera détaillé par la suite.

4.1 Deploy of slaves

Le programme Deploy est dans le package "deploy", ce Java package a 4 classes.

- Deploy.java
- Implementation.java
- Command.java
- IO.java

La classe "Deploy.java" est la classe main, elle exécute simplement le programme et n'a que 30 lignes de code.
La classe "Implementation.java" a les méthodes:

- void getFunctional_PCs()
- void deployFile(String pathfile)
- **plusieurs getters()**

Ces méthodes exécutent un processus par machine disponible. Pour aller plus vite, et profiter le fait que l'on travaille sur un système reparti, on crée un thread pour chaque commande, une fois toute la liste de threads est prête. On commence l'exécution de chaque thread et finalement, on attend que tous les processus terminent pour passer à la tâche suivante.

Voici le code implémenté.

```
Scanner sc = new Scanner(new File("listPCs.txt"));

while (sc.hasNext())
{
        String str = sc.nextLine();
        Command command = new Command();
        command.setCommandSSH(str, "␣hostname");
        threads.add(command);
}
sc.close();

for (Thread thread : threads) {
        thread.start();
}
for (Thread thread : threads) {
        thread.join();
}
```

4.2 Nettoyage of slaves

Le programme Clean est dans le package "clean", ce Java package a 4 classes.

- Clean.java
- Implementation.java
- Command.java
- IO.java

La classe Clean.java est la classe main, comme la classe main de deploy elle exécute simplement le programme et n'a que 20 lignes de code.
La classe Implementation.java a les méthodes:

- void getFunctional_PCs()
- void clean()
- **plusieurs getters()**

Ces méthodes exécutent un processus par machine disponible et pour gagner du temps on applique la même méthode détaillée précédemment.

5 MapReduce

Pour cette partie de l'implementation du MapReduce, on détaillera les programmes du Master et du Slave.

5.1 Master

Le programme Master est dans le package master, ce programme est le plus grand en termes de lignes de code. Ce package a 5 classes :

- Master.java
- Deploy.java
- Command.java
- IO.java
- Split.java

La classe master est la classe main, celui-ci vérifie d'abord les machines de l'école qui sont disponibles, pour ensuite effacer tous les possibles fichiers qui y sont et qui peuvent provoquer un mauvais fonctionnement du programme. Par la suite, on déploie aux machines de l'école un fichier "Functional_PCs.txt" qui a la liste de tous les machines disponibles et fonctionnent correctement, ceci est nécessaire pour la phase de shuffle, on déploie le programme Slave qui a comme nom "slave.jar". C'est à partir de ce moment que l'on commence à exécuter notre algorithme de MapReduce avec toutes les phases déjà presentées dans la Figure 1, donc pour résumer notre programme Master exécute dans l'ordre suivante les méthodes:

Prétraitement:

- deploy.getFunctional_PCs();
- deploy.clean();
- deploy.deployFunctionalPCs();
- deploy.deployFile(slave.jar);

MapReduce:

- Split.split(filename, numberOfSplits);
- deploy.deploySplits();
- deploy.map();
- deploy.shuffle();
- deploy.reduce();

Pendant les étapes de map, shuffle et reduce, le programme Master lance les multiples processus légers de manière parallèle, la méthode utilisée est la même que celui que l'on a décrit pour le programme Clean et Deploy, i.e. créer pour chaque processus un thread, les lancer tous ensemble et attendre que tous finissent ses tâches.

5.2 Slave

En ce qui concerne le programme Slave on a 5 classes aussi, ceux-ci sont :

- Slave.java
- Reduce.java
- Shuffle.java
- SendFile.java

Comme dans tous les autres programmes, on a comme classe main la classe qui a le même nom que le package, dans l'occurrence Slave.java.

Le programme Slave a 3 modes de fonctionnement:

- Mode 0 : Count
- Mode 1 : Shuffle
- Mode 2 : Reduce

Ces fonctionnements sont réglés avec les arguments au moment de l'appel du Slave, par exemple:

- Mode 0 : java -jar slave.jar 0 /tmp/ffacundo/splits/S0.txt
- Mode 1 : java -jar slave.jar 1 /tmp/ffacundo/maps/UM1.txt
- Mode 2 : java -jar slave.jar 2

6 Résultats et Conclusion

Après implémentation du projet, j'ai effectué un test avec le fichier forestier.txt du dépôt les codes en vigueur, le séparant 30 en splits de taille 50kB, et exécutant l'algorithme sur 33 slaves.

Les tests ont été peu convaincants car l'exécution a duré près de 2 minutes et l'algorithme séquentiel a pris moins d'une seconde pour faire le comptage. Ceci est du pour plusieurs raisons, d'abord le temps d'envoi des splits vers les machines de Télécom Paris (slaves) est de plusieurs secondes, cela est dû au fait que les fichiers sont envoyés vers internet car le Master s'exécutait sur ma machine à Gif-sur-Yvette. Ensuite le shuffle est l'une des phases le plus couteux en termes de temps car plusieurs machines envoient et reçoivent plusieurs files en même temps. Tous ces processus bien sûr peuvent être optimisés mais malheureusement pour problèmes de temps je n'ai pas pu les corriger.

On conclut qu'il faut rester en séquentielle tant que le fichier à traiter ne soit plus gros que quelques gigaoctets (Google utilise MapReduce pour traiter des fichiers de taille de l'ordre de Téraoctets) et utiliser le MapReduce dans un réseau local très rapide.

Bibliographie

[1] https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

[2] Tom White. *Hadoop-The Definite Guide*. (Anglais)