24 NOVEMBRE 2016

PROJET SHAVADOOP

INF 727 : SYSTEMES REPARTIS POUR LE BIG DATA

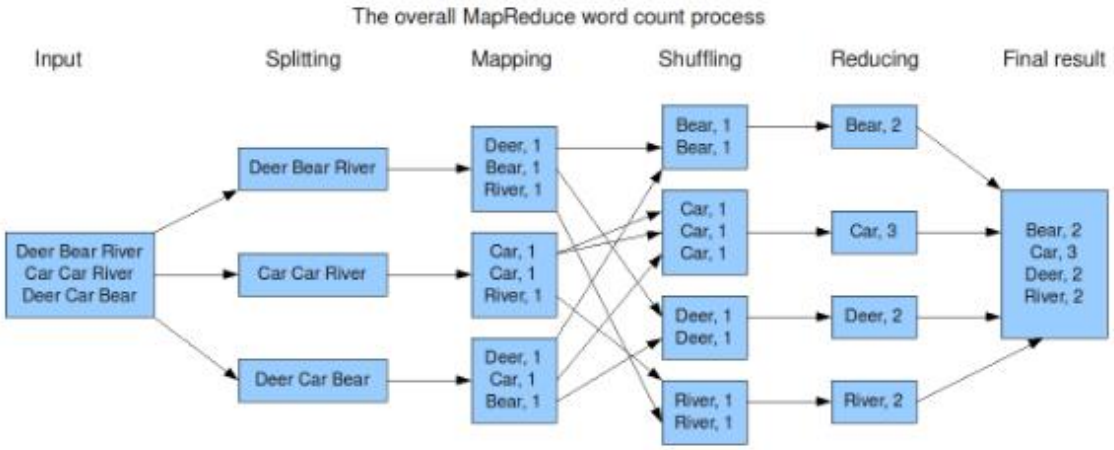
ALEXANDRE BREHELIN

Table des matières

I. Introduction

Pour rappel, notre objectif est de construire un programme qui imite le fonctionnement de l'algorithme du framwork Map-Reduce. Notre souhait est d'implémenter un algorithme qui calcule un « Word Count » sur plusieurs machines, et de manière parallélisée. Pour cela nous aurons à disposition l'ensemble des machines disponibles sur le site de Télécom Paris-Tech, sur lesquelles nous pouvons nous connecter sans restriction à l'aide de la commande SSH.

Notre implémentation personnalisée de l'algorithme Map-Reduce se fera selon le modèle ci-dessous :



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

Cette note est accompagnée de trois fichiers, d'une part le fichier zip qui contient le code source de notre programme, et d'autre part, les deux fichiers jar servant à l'utilisation du programme.

II. Comment lancer le projet

Pour lancer notre programme, nous aurons besoin des deux fichiers jar joints avec cette note (master_shavadoop.jar & salve_shavadoop.jar). Le premier cité représente le code associé au Master de notre algorithme et le second au programme que nous exécuterons sur chaque machine. Lors du lancement de notre « Word Count » nous aurons seulement besoin du premier cité. Une fois les deux jar récupérés, et sauvegardés dans le même chemin, nous pouvons tenter d'exécuter le programme. Pour cela, il suffit de se rendre dans le terminal, de se placer dans le dossier où se situe nos jar et d'exécuter la commande suivante :

```
> java –jar master_shavadoop.jar PATH INPUT.txt
```

Le « PATH » doit être un chemin tel que « /cal/homes/brehelin/Desktop/Sx_Folder/ », et le fichier INPUT.txt doit être le nom du fichier texte sur lequel on souhaite effectuer notre comptage tel que « forestier_mayotte.txt ».

Le « PATH » représente le répertoire de référence tout au long du processus de traitement, il est donc nécessaire que celui-ci contienne tous les éléments dont a besoin notre programme. Il est impératif que notre dossier contienne:

- Le fichier source texte sur lequel nous voulons lancer notre programme
- Un fichier texte qui référence l'adresse des machines sur lesquelles nous souhaitons paralléliser notre programme sous le nom « ip_adress.txt » (joint avec cette note).
- Le fichier texte « pronom_list.txt » qui référence tous les pronoms et autres mots à bannir de notre « Word Count »

Après lancement du projet, on retrouvera tous nos fichiers de sortie dans ce dossier de référence.

III. Comment fonctionne le projet

Une fois le projet lancé, celui-ci déclenche un processus de traitement de notre fichier texte selon 6 étapes que nous appellerons :

- Démarrage
- Splitting
- Mapping
- Shuffle + Reducing
- Assembling
- Sorting

a) Démarrage

La première étape, que nous appelons « démarrage » consiste à tester la connexion à chacune des machines inscrites dans notre fichier « ip_adress.txt ». Lors du lancement du programme chacune des machines qui réussit sa tentative de connexion s'affiche précédée d'un « OK ». Une fois toutes les machines testées, cette étape renvoie un fichier texte dans notre chemin de référence sous le nom « connect_machine.txt ». Celui-ci contient la liste de toutes les machines qui ont réussi à se connecter. Dans le même temps, notre programme conserve la liste des machines connectées, ainsi que le nombre total de machine connectées. Cette opération est réalisée par la fonction « read_write » dans la classe « master_shavadoop », et consiste à lancer des micro-batch à partir de la classe ProcessBuilder.

b) Splitting

La seconde étape de notre projet consiste à découper notre fichier texte d'entrée en plusieurs morceaux. L'objectif du découpage de notre texte est de créer un nombre de morceaux importants afin de donner le traitement de chacun de ces morceaux à une machine, et donc de paralléliser le traitement des morceaux de texte. La première approche pour exécuter cette étape est de découper notre fichier texte par ligne, et donc de créer un nombre de fichiers à traiter égal au nombre de lignes. Par ailleurs, sur des fichiers de taille importante on peut optimiser ce découpage par un découpage initial par mots, car découper par ligne alloue de manière très inéquitable la charge de travail entre les machines.

On décide après le test sur le fichier « forestier_mayotte.txt », d'opter pour un découpage du fichier texte par mots. On découpe donc le fichier en n mot, puis on divise le nombre total de mots par le nombre de machines. On obtient alors le nombre de mots que l'on doit renseigner dans chaque morceau à traiter afin d'avoir autant de fichiers à traiter que de machines. Evidemment, s'il existe un reste à cette division, on crée un nombre supplémentaire de fichiers selon que le reste soit pair, impair ou autre. Ce découpage permet donc d'avoir environ autant de fichiers à traiter pour chaque machine, et ceux-ci de taille égale. Par ailleurs, si le nombre de mots est inférieur au nombre de machines, on décide de créer autant de fichiers qu'il y a de mots. Dans ce cas-là, notre solution est moins optimale qu'un découpage par ligne. Malgré tout, il est rare que l'on possède plus de machines que de mots dans un fichier texte.

Dans cette étape, on s'occupe également de normaliser chacun des mots, lors du découpage en n mot afin de renvoyer des mots dits propres dans nos morceaux. Les morceaux créés sont envoyés dans le chemin de référence sous la forme Sx où x représente le numéro du morceau.

c) Mapping

Dans notre processus, l'étape de mapping consiste, pour chaque morceau Sx, à lancer un traitement sur une machine. Le dît traitement, extrait tous les mots du fichier Sx, et crée un fichier SMx qui liste chacun des mots avec le chiffre 1. Lors de cette étape, pour chaque Sx, on lance le jar salve_shavadoop avec le mode « SXUMX » pour réaliser l'opération. Cette opération génère deux dictionnaires, le dict_UMX qui référence chacun des Sx et par quelle machine il a été traité, et le dict_W qui référence pour chaque mot dans quels RMx il apparait.

Lors du lancement du projet dans le terminal, l'étape de mapping renvoie plusieurs informations à l'utilisateur. D'une part le traitement de chaque Sx par le jar salve_shavadoop print les mots qui ont été traités (de manière unique), et d'autre part, les dict_W et dict_UMX sont affichés en sortie dans

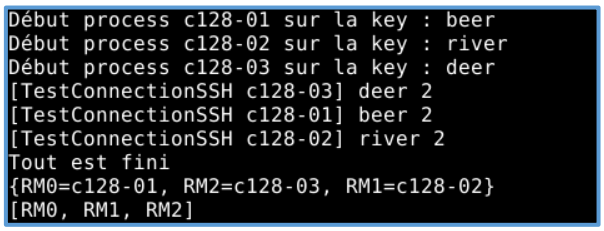
le terminal à la fin de la réalisation du mapping. Par ailleurs, lors de l'étape de mapping par le jar, on réalise une étape supplémentaire dans la normalisation des mots car on ne conserve pas dans les fichiers RMx les mots appartenant à notre fichier «pronom_list.txt » et ceux qui sont composés de chiffres.

(Ci-dessus un exemple de sortie de l'étape Mapping, le mot « Car » n'apparait pas car il s'agit d'un mot du fichier « pronom_list.txt »)

d) Shuffle + Reducing

L'étape suivante consiste pour chacune des clés du dictionnaire Dict_W, à parcourir ses valeurs, soit les UMx et à créer un fichier SMx qui contient toutes les occurrences du mot, soit la clé. Ce fichier SMx n'a pas de réel intérêt dans notre implémentation. En effet, lors du parcours des UMx, on calcule la somme des occurrences, et on crée un fichier RMx avec le mot (la clé) et son nombre d'occurrences totales. Pour réaliser ce traitement on boucle sur les clés de Dict_W et sur les machines afin de réaliser le traitement de manière parallélisé. Sur chacune des machines est lancé le jar salve_shavadoop avec le mode « UMXSMX ». Ce lancement est réalisé à partir de la classe « LaunchSlaveShavadoop » qui nous permet de lancer nos threads sur chaque machine, et de récupérer des informations à la fin de celles-ci. Cette étape permet de créer également deux dictionnaires, RM_MACHINE qui donne pour chaque RMx quelle machine a effectué le traitement, et le dictionnaire RM_NAME qui renvoie la liste de tous les RMx créés. Sous cette forme, le nombre de RMx doit être égal au nombre de mots.

Lors de cette étape, à chaque lancement de thread est affiché le nom de la clé et la machine qui la traite en sortie, puis le résultat du nombre total d'occurrences à la fin du thread. A la fin de tous les threads on affiche « Tout est fini », ainsi que les deux dictionnaires que nous avons créé lors du traitement.



(Ci-dessus la sortie de l'étape Shuffling+Reducing sur l'exemple classique)

e) Assembling

Une fois tous nos fichiers RMx créés, il est nécessaire de créer un seul et unique fichier de sortie. Ce fichier de sortie appelé « output.txt » sera enregistré dans le répertoire de référence. L'étape d'Assembling utilise la fonction « mergefile » dans la class master_shavadoop, qui consiste à parcourir chacun des fichiers RMx et à écrire leurs contenus dans un fichier commun. Cette étape s'exécute uniquement dans le Master, et ne renvoie pas d'affichage particulier dans le terminal. Par ailleurs, l'écriture sur le fichier commun a lieu à chaque lecture de RMx, et non à la fin de la lecture de tous les RMx.

f) Sorting

L'étape de sorting prend en entrée le fichier que l'on extrait de l'étape précédente, assembling, et renvoie en sortie dans notre répertoire de référence le fichier « output_sort.txt ». On utilise la fonction « sorted_file » de la class master_shavadoop pour réaliser cette opération. Notre fonction parcourt tout le fichier « output.txt » pour classer chacun des mots selon son occurrence.

IV. Résultats

Cette section présente les meilleurs résultats que l'on a pu obtenir sur 3 fichiers différents au cours de nos différents tests. Nous évaluons le résultat par la qualité du classement comparativement à la recherche d'occurrence des mots dans des logiciels de traitements de texte, et non par le temps d'exécution qui est corrélé aux machines fonctionnelles (varie selon l'exécution). Les 3 textes utilisés pour nos tests sont : forestier_mayotte, déontologie_police_nationale et domaine_public_fluviale. Chacun des tests a lieu sur un groupe de près de 60 machines inscrites dans le fichier « ip_adress.txt » (elles peuvent se connecter ou non).

Résultats Forestier_mayotte :

Mots	Occurrences
biens	8
forestier	6
code	6
partie	5
gestion	4
aux	4
dispositions	4
agroforestiers	4
communes	4
mayotte	4
…	…

Cf : Répertoire Github, output_forestier_moyotte_sorting.txt

Etapes	Temps d'exécution en milliseconde
Démarrage	17 474
Splitting	104
Mapping	8 635
Shuffle + Reducing	17 191
Assembling	4 228
Sorting	7
Temps total	**47 639**

Résultats Déontologie_police_nationale :

Mots	Occurrences
police	38
nationale	20
article	20
titre	13
autorite	11
fonctionnaires	10
code	9
aux	9
fonctionnaire	9
commandement	8
…	…

Cf : Répertoire Github, output_deontologie_sorting.txt

Etapes	Temps d'exécution en milliseconde
Démarrage	62 689
Splitting	208
Mapping	13 658
Shuffle + Reducing	25 850
Assembling	31 441
Sorting	18
Temps total	**133 864**

Résultats Domaine_public_fluviale

Mots	Occurrences
article	172
bateau	101
immatriculation	82
tribunal	69
aux	65
lieu	64
bureau	49
code	48
navigation	45
bateaux	43
…	…

Cf : Répertoire Github, output_domaine_fluvial_sorting.txt

Etapes	Temps d'exécution en milliseconde
Démarrage	21 725
Splitting	383
Mapping	17 886
Shuffle + Reducing	38 823
Assembling	74 757
Sorting	32
Temps total	**153 606**

V. Erreurs & Pistes d'optimisations

- Connection des machines : Le test de connexion à l'ensemble des machines est loin d'être optimal. En effet, d'une part ces tentatives ne sont pas réparties et d'autre part, nos tentatives ne sont pas soumises à une restriction de temps. Il y a donc de fortes latences lorsqu'une machine ne répond pas car on doit patienter jusqu'au message d'erreur.

- Splitting : Notre méthode de découpage par mot n'est certainement pas la plus optimale car elle nous oblige à parcourir tout le fichier, et donc à réaliser une pré-étape de mapping de notre texte. De plus cette méthode n'est pas optimale pour les petits fichiers, et crée un délai de traitement relativement long pour ceux-ci. Par ailleurs, la répartition des taches de manière équitable entre les machines semble la bonne approche, il faudrait donc splitter notre fichier en Bytes de taille relativement égale le tout en s'assurant de ne pas couper de mots lors du split.

-**Mapping :** Lors de cette étape, il serait préférable de lancer au sein d'un même thread tous les traitements qui concernent une machine. En effet, lancer plus de 4 threads (donc plus de 4 connexions SSH) sur des machines 4 cores semble générer des erreurs à cause d'une surcharge de requêtes.

-Shuffling + Reducing : Cette étape génère régulièrement, et de manière aléatoire, des erreurs dans l'exécution de notre processus. De même, le nombre de machines utilisées dans le traitement semble influer sur la réussite du traitement. Deux types d'erreurs surviennent sur ce traitement, des erreurs de « Time Out », et « SSH Excchange -Remove Connection Host ». Ces erreurs concernent des erreurs de connexion machine, et entrainent la perte des mots qui ont été concernés par le traitement lancé sur ces machines. Il serait donc plus judicieux de ne pas lancer plus de threads que de core présents sur les machines, et de limiter les connexions SSH. Il serait aussi préférable de réfléchir à un traitement plus « groupé » des mots puisque lors de cette étape on crée autant de fichiers qu'il y'a de mots, et on génère donc autant de connexions SSH.

-Normalisation du texte : On utilise un fichier « pronom_list.text » qui renseigne l'ensemble des mots que l'on ne veut pas compter, celui-ci permet de ne pas conserver ces mots. De plus, on normalise les mots de manière à supprimer les caractères spéciaux ou chiffres dans les mots afin de ne pas conserver des mots dit « bizarres ».

-**Assembling :** Notre étape d'assembling semble prendre régulièrement beaucoup de temps lors des divers lancements. Cela peut venir d'une part du grand nombre de fichiers RMx, ou de l'absence de certains d'entre eux dû aux erreurs précédemment citées. Par ailleurs, il est fort probable qu'il existe une façon plus optimale de traiter notre assembling. On pourrait notamment penser à garder en cache les informations des RMx, et ensuite écrire toutes les informations en un seul processus.