JULIEN MAKSOUD – MS BGD 2020-2021

Projet MapReduce – INF727

<u>Ce que j'ai pu faire :</u> Arriver à la fin de l'étape 12, à savoir un MapReduce pour les fichiers input (41 secondes) et pour une cinquantaine de versets de la Bible (158 secondes). Mais n'ai pas réussi à faire passer ce code à l'échelle pour un fichier de 10mb. Les limitations et une série d'améliorations possibles seront citées plus bas.

<u>Ce que j'aurais voulu faire :</u> Implémenter lesdites améliorations et savoir à partir de combien de mots (ou quelle taille en data) il devient « rentable » de passer par un MapReduce plutôt que de faire un séquentiel. Et également essayer de retrouver la loi d'Amdahl en retrouvant le facteur 20 avec un grand nombre de machines.

TEMPS RELEVÉS EN SÉQUENTIEL

<u>WORDCOUNT EN SÉQUENTIEL</u>				
Tache	**Inputs (deer deer car river)**	**Code Forestier (2ko)**	**Code du commerce (10 mo)**	**Fichier de 330mo**
Open & Read File	1.2 msec	2.96msec	430 msec	19130 msec
Wordcount	0 msec	0 msec	869 msec	18270ms
Tri par ordre alphabétique	0 msec	0.99 msec	267 msec	16530 msec
<u>Total</u>	**1.2 msec**	**4 msec**	**1670 msec**	**51000 msec**

FONCTION DE LECTURE + COMPTAGE DES MOTS

Entrée [62]:

```
%%time

def print_words(filename):

    start_time = time.time()
    text =  open(filename,"r", encoding='utf-8').read().replace("\n"," ").split()
    count = {}
    print("Read & Preprocess file: %s seconds " % (time.time() - start_time))

    start_time = time.time()
    for w in text:
        if w not in count:
            count[w] = 1
        else:
            count[w] += 1
#    print("It took --- %s seconds --- to count word occurences" % (time.time() - start_time))
    print("Count word occurences: %s seconds " % (time.time() - start_time))
    start_time = time.time()
    count_sorted = sorted(count.items(), key = lambda x : (-x[1],x[0]), reverse= False)
#    print("It took --- %s seconds --- to sort word by occurences" % (time.time() - start_time))
    print("Sort word by occurences and alphabet: %s seconds " % (time.time() - start_time))
    return count_sorted

```

```
Wall time: 0 ns
```

Entrée [63]:

```
print_words(r"C:\Users\julie\Documents\MasterBGD_Python\INF727_TP_CalculDistribue\input.txt")
```

```
Read & Preprocess file: 0.001256704330444336 seconds
Count word occurences: 0.0 seconds
Sort word by occurences and alphabet: 0.0 seconds
```

Out[63]:

```
[('Car', 3), ('Beer', 2), ('Deer', 2), ('River', 2)]
```

<u>Fig 1 : Rappel du code en séquentiel</u>

WORDCOUNT EN PARALLELE			
Tâche en cours	**Temps (ms) Fichier Input**	**Temps (ms) 3 x 200 mots**	**Temps (ms) File 10Mo**
Identification des nœuds existants – séquentiel	**49.9**	**15.8**	**47.2**
Temps nécessaire à pinger 3 machines – séquentiel	**4392**	**3899**	**4142**
Deploy (création d'arborescence et copie des Slave.py sur les nœuds de calculs) – process en parallèle / nécessite un ssh	**5847**	**5005**	**5988**
Envoi des splits aux machines distantes – process en parallèle. ***Process retardé par : un scp qui implique un ssh***	**564**	**358**	**2400**
Execution des Maps – process en parallèle ***Process retardé par :*** - ***Le fait que le Master lance la phase Map rend nécessaire le ssh pour se connecter aux nœuds de calcul – Le Map des splits se fait ensuite en parallèle sur les différents nœuds.*** - ***Génération et écriture du fichier*** - ***Nécessite d'attendre la fin du Map sur tous les nœuds***	**5189**	**5672**	**5452**
Envoi de la liste des machines aux slaves – process parallèle ***Process retardé par un ssh lié au scp qui est un protocole sécurisé nécessitant une authentification.***	**581.6**	**385**	**629**
Préparation et éxécution du shuffle – process en parallèle. ***Process retardé par :*** - ***Génération de nombreux fichiers (autant que de mots uniques)*** - ***Calcul de modulo de hash pour savoir où les envoyer*** - ***SCP (copie sécurisée impliquant un ssh) pour envoyer les fichiers sur d'autres machines.*** - ***L'attente que toutes les machines aient fini le shuffle***	**7304**	**132934**	***INFINI*** ***Timeout mis à 2h40***
Réduce sur le slave – process en parallèle ***Process ralenti par :*** - ***Le fait que le master contrôle le lancement du process*** - ***Chaque machine doit ouvrir une série de fichiers et les lire afin d'alimenter un dictionnaire. Dans la phase séquentielle, ça prend à chaque fois du temps.*** - ***L'attente que tous les reduce soient finis***	**8009**	**4769**	**N/A**
Copie des résultats de Reduce vers le master pour Reduce « final » – process en parallèle pour la copie puis séquentiel pour le reduce final	**8165**	**4938**	**N/A**
TOTAL (en secondes)	**41 secs**	**158 secs**	**INFINI**

LIMITATIONS

Espace disque occupé : L'espace disque occupé est colossal comparé à la taille des inputs. J'ai obtenu un facteur 15 entre la taille des fichiers inputs et la taille totale occupée sur le disque, ce qui est intenable en production.

Le shuffling est de loin l'étape la plus longue : Il y a donc un problème de temps de copie et de communication : La création et l'envoi de très nombreux fichiers (au moins un par mot) ainsi que les transferts entre différents slaves sont rédhibitoires si la quantité de données à traiter devient significative.

Temps d'ouverture et écriture de fichiers : Il y a un temps incompressible d'ouverture et de prétraitement. On voit de suite qu'il vaut mieux avoir un gros fichier que beaucoup de petits fichiers.

Passage à l'échelle : En essayant avec trois fichiers de 10 Mo, j'ai dépassé le timeout de 2h40 que je m'étais fixé. **Le système est donc trop archaïque et ne passe pas à l'échelle.**

Pas de gestion des pannes : Si une machine tombe en panne en plein milieu d'un calcul, il faut pouvoir récupérer son dernier point de synchronisation afin d'éviter de recommencer le calcul de zéro.

Les points de synchronisations : En pratique (par exemple dans Hadoop), les données à traiter sont divisés en fichiers de 64mb. On peut donc s'attendre à une synchronisation « naturelle » des process. Si un split est nécessaire, on doit s'assurer pour le wordcount que ledit split ne se fait pas au milieu d'un mot (par exemple chercher le dernier espace avant la fin d'une ligne et couper à cet endroit)

Elasticité : Dans un cas réel, les données à traiter arrivent continuellement. Il faut un système élastique qui soit capable de solliciter automatiquement des nœuds de calculs supplémentaires pour faire face à une charge de travail accrue.

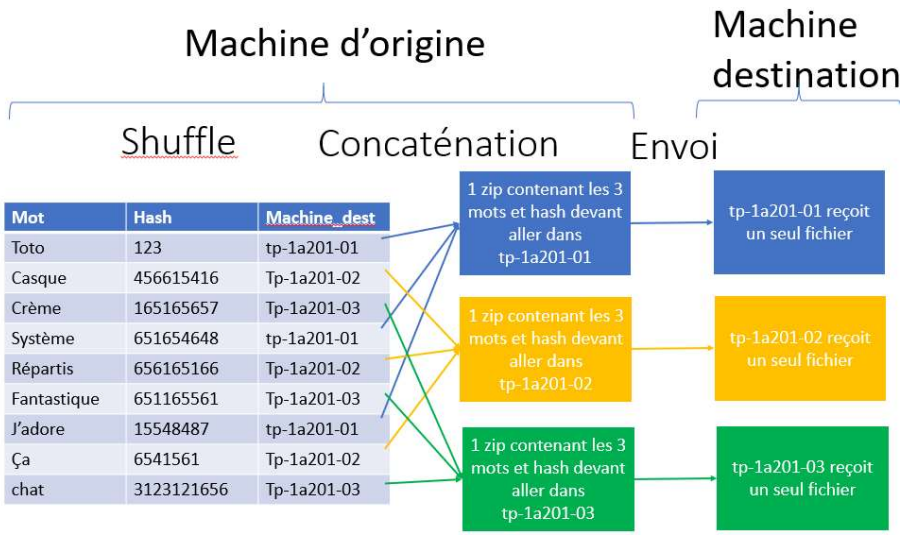
QUELQUES OPTIMISATIONS POSSIBLES

Il faut diminuer le « coût fixe » lié aux ouvertures / écritures / fermetures / communications. Pour cela :

- **Utiliser beaucoup plus de mémoire cache que d'écriture sur disque.**
- **Réduire au strict minimum les communications entre machines**
- **Limiter le nombre de fichiers à échanger. Envoyer un fichier de 1Gb est plus rapide qu'envoyer 1000000 fichiers de 1ko chacun**

Pour ce faire, on peut envisager de :

- **Diminuer le nombre des fichiers à transférer par exemple avec un ZIP :** J'ai essayé (sans succès !!) de zipper tous les fichiers qui doivent être shufflés vers une même machine (avec 3 machines, je génère donc 3 zips, ce qui ne me fait que 6 envois en tout). Si on a n machines, on limite le nombre de ***scp*** à n*(n-1) quel que soit le nombre de mots. Les fichiers envoyés d'une machine à elle-même peuvent être juste déplacés. Une alternative est de concaténer tous les fichiers dans un dataframe contenant hash + mot + count et stocker ce dataframe dans un seul fichier qui est ensuite envoyé.



- **Diminuer la taille des fichiers. Cela peut être fait en faisant un premier reduce à l'étape du Shuffle**. Vu que chaque mot a un hash unique associé, il faut incrémenter le compteur dans le fichier <hash>-<machine> afin de gagner du temps par la suite. **Spark** le fait déjà avec la fonction ***reducebykey***.
- **Supprimer les données d'un point de synchronisation une fois le point suivant atteint :** Typiquement, une fois le shuffle fait, on peut supprime le fichier map et stocker les splits sur une solution de stockage à long terme (si on en a plus besoin)

- **Utiliser des splits de même taille :** Le but étant que les process qui tournent en parallèle se finissent en même temps, il faut répartir la charge de travail de manière équitable. Sur Hadoop, les fichiers sont en général splittés en 64mb.
- **Nombre de machines à allouer / « Elasticité » du système :**
    - En batch processing, on connaît la quantité de donnée à traiter. On peut donc « réquisitionner » un certain nombre de machines disponibles que l'on sait nécessaires au calcul. Et ajouter une marge (par exemple, si on a besoin de 20 machines, en réquisitionner 23).
    - En stream processing, il faut un « manager » qui connaisse à tout instant la charge de travail sur les épaules des différents nœuds/workers.
- **Voir s'il est possible de se passer des mots pendant une partie du workflow :** Si on peut retrouver le mot à partir de la clef, alors on devrait pouvoir se passer de l'écriture du mot dans les shuffles, et juste écrire le nombre d'occurrences dans le nom du fichier pour économiser des ouvertures et des écritures.
- **Faire un premier reduce pendant le shuffle :** Afin de gagner du temps, il faudrait indexer les hash calculés et incrémenter le count associé si le shuffle voit un mot déjà indexé. Le slave peut également faire le reduce de son shufflereceived sans attendre l'ordre du Master. Puis envoyer son « reduce partiel » au Master afin que le Master fasse le « reduce final ».

EN CAS DE PANNE

- **En cas de panne machine, il faut que le split et le slave soient envoyés sur un autre nœud de calcul.** Le master devrait « pinger » les slaves toutes les 15 secondes. Et en cas de timeout, envoyer le split et le slave sur un autre nœud afin de continuer le calcul. **Idéalement mettre la donnée à traiter en cache en mémoire afin d'être très rapidement disponible sur de nouveaux slaves (c'est ce que fait *Spark*).**
- Il faudrait pouvoir **relancer directement le process sur un autre slave sans avoir à recalculer les étapes précédentes**. C'est ce que fait *Spark* à l'aide de points de sauvegarde fixes lors du traitement des données. Il serait ici capable de récupérer directement les shuffles sans refaire le map
- Des moyens importants sont déployés par les entreprises pour aller vers **l'autonomic computing**, qui serait capable d'autoguérir en réparant automatiquement les pannes, d'autoprotection en cas d'attaque et optimisation des flux de données (auto-optimisation)

CONCLUSIONS

On peut faire l'analogie avec le fait de prendre l'avion plutôt que le train : Ca n'a pas de sens sur un Paris-Londres. Mais ça en a sur un Paris-Madrid. Si la distance est assez longue, la vitesse d'un avion devient « rentable » quand bien même on perd beaucoup de temps à se rendre à l'aéroport et faire face à toutes les pertes de temps que ça implique.

On a besoin de garder une certaine synchronisation entre les slaves. Même si les reduce partiels sont finis plus vite sur une machine que sur une autre, le bottleneck reste la machine la plus lente (sans compter le réseau)

Au final, comme attendu, quand il s'agit de traiter quelques mots, le MapReduce est bien plus rapide en séquentiel. Par contre, la parallélisation devient rentable quand le nombre de mots augmente de manière significative.

Quand les données arrivent d'un coup, on peut dimensionner l'équipement nécessaire au traitement. Mais quand le flux de données est continu et varie (trafic sur un site internet par exemple), alors on a besoin **d'élasticité** avec un nombre de machines qui fluctue au cours du temps. Il faut donc un process automatique qui sait gérer le besoin disponible et y répondre. Cela afin de garder de bonnes performances face à un afflux massif de données à traiter. Et également savoir « réduire la voilure » et de regrouper/éliminer les nœuds inutiles afin de réduire la facture énergétique. Les providers de service cloud (tel Amazon EC2) ont maintenant cette capacité de proposer de la puissance de calcul « on-demand ».

ANNEXE : DEROULE DU WORKFLOW

Le programme contient un deploy, un master et un slave. Ces fichiers sont de base stockés sur mon user. Le lancement du master peut se faire depuis n'importe où. Au lancement du master, les étapes suivantes sont effectuées ;

1. Lecture d'un fichier contenant la liste de tous les nœuds disponibles **– process en séquentiel**
2. Recherche de machines qui répondent au ping **– process en séquentiel / necessite des ssh**
3. Execution du Deploy qui créé toute l'arborescence nécessaire sur les nœuds distribués et y copie le slave.py **– process en parallèle / nécessite un ssh**
4. Envoi des splits (fichiers textes) vers les machines sélectionnées **– process en parallèle**
5. Le Master lance l'éxécution du Map sur les slaves. Les slaves lisent les fichiers texte et génèrent les fichiers UM.txt **– process en parallèle sur les différents nœuds. Nécessite d'attendre la fin du Map sur tous les nœuds**
6. Le Master envoie la liste des nœuds à chacun des nœuds **– process en parallèle**
7. Préparation et éxécution du shuffle : Dans chaque nœud, calcul des hash pour chaque mot contenu dans les fichiers Map (UM.txt) Puis création d'un fichier contenant autant de fois le mot qu'il n'existe dans les splits. Cette étape génère beaucoup d'entropie. Elle créé beaucoup de fichiers. Puis va calculer où les envoyer puis enfin fera des scp pour les envoyer. **– process en parallèle / créé et envoie de nombreux fichiers ce qui est chronophage**
8. Les slaves reçoivent les fichiers (un par mot) qu'ils doivent ouvrir afin de lister les mots contenus et le nombre de fois qu'ils apparaissent.
9. Le Master doit compiler tous les reduce générés dans les différent slaves (chaque slave contient un fichier reduce par clef de hachage) et sortir un wordcount.