TP2 - A la découverte du Framework MapReduce

1 Introduction et objectifs du TP

Le TP précédent était consacré à l'installation d'un cluster Hadoop. Nous allons maintenant exploiter ce cluster pour effectuer les premiers traitements à l'aide du patron de programmation MapReduce. Mis au point par Google, ce patron permet de traiter de façon parallèle de gros volumes de données. Ce patron, largement populaire chez les acteurs du Big Data, repose sur deux primitives, *Map* et *Reduce*, inspirées par la programmation fonctionnelle. Au cours de ce TP, nous utiliserons l'implémentation du patron MapReduce fournie par le framework Hadoop et mettrons en place les processus MapReduce en Java. La Figure 1 illustre les 5 étapes d'un process (appelé communément *job*) MapReduce :

1. Un job MapReduce prend un jeu de données en entrée initialement stocké sur une partition HDFS. Ces données peuvent être de différentes natures, e.g., documents textuels, n-uplets, données numériques, images. La première étape consiste à *découper* ce jeu de données de sorte à produire un ensemble de paires de type $\langle Clé, Valeur \rangle$. Chacune de ces paires sera ensuite envoyée à un nœud qui exécutera la seconde étape, la fonction Map.
2. Dans chaque nœud où elle s'exécute, la fonction Map reçoit en entrée une paire de type $\langle Clé, Valeur \rangle$, effectue un traitement sur la valeur et retourne un ensemble de paires de type $\langle CléInter, ValeurInter \rangle$. Chaque nœud émet ainsi un ensemble de paires. Cette étape est à la charge du programmeur.
3. Les paires de type $\langle CléInter, ValeurInter \rangle$ émises par tous les nœuds Map lors de la phase précédente sont triées dans cette troisième étape. Sur le principe des tables de hachages, les paires sont regroupées selon les valeurs de $CléInter$. Le résultat de cette étape est ainsi la géné-

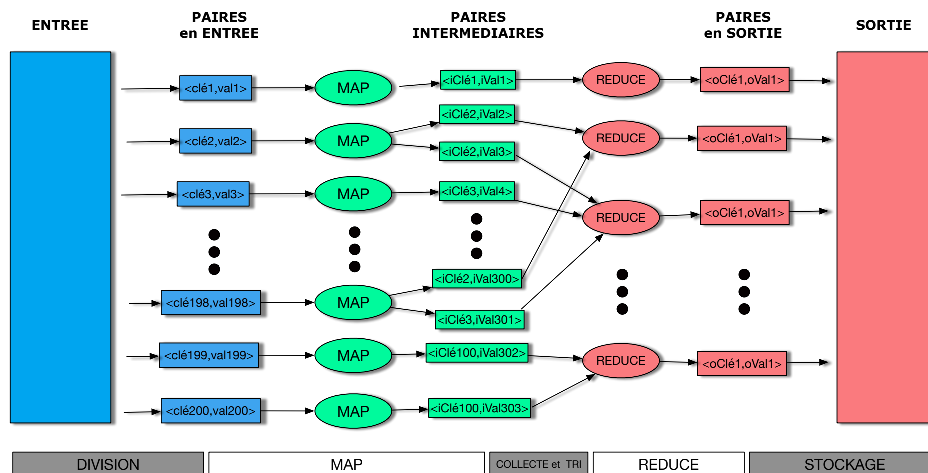

FIGURE 1 – Les étapes d'un processus MAPREDUCE . Dans la partie haute de cette figure, les rectangles représentent des données et les ovales représentent des nœuds du cluster. Concernant la partie basse, les étapes du processus qui présentent un fond gris sont des étapes automatiquement gérées par le framework. A contrario, les étapes sur fond blanc sont à la charge du développeur.

ration de nouvelles paires, de type $\langle Cl\acute{e}Inter, ValeurInter[]\rangle$. Cette étape est automatique.

4. Chaque noœud où s'exécutera une fonction *Reduce* reçoit ainsi une paire de type $\langle Cl\acute{e}Inter, ValeurInter[]\rangle$. La fonction *Reduce* effectue alors un traitement sur cette paire et retourne une nouvelle paire de type $\langle Cl\acute{e}Sortie, ValeurSortie\rangle$. Cette étape est à la charge du programmeur.
5. Enfin, ces paires de type $\langle Cl\acute{e}Sortie, ValeurSortie\rangle$ sont agrégées et stockées sur le cluster. Dans le cadre de l'implémentation de MapReduce par Hadoop, le résultat du processus sera stocké sur le système de fichier HDFS.

Nous voyons ainsi que l'utilisateur ne s'occupe pas de la parallélisation de la tâche mais se consacre uniquement au traitement des données à travers les fonctions Map et Reduce, i.e., le reste étant géré par le framework. Néanmoins, la conception des fonctions *Map* et *Reduce* peut s'avérer non triviale et demander une profonde réflexion de la part du programmeur.

2 Ecriture et exécution du premier job MapReduce : *WordCount*

2.1 Fonctionnement

MapReduce possède son *HelloWorld*, i.e., un premier programme qui permet de comprendre la syntaxe et les mécanismes généraux de fonctionnement d'un langage ou d'un patron de programmation. Dans le contexte MapReduce ce programme s'intitule *WordCount* et permet, comme son nom l'indique, de compter le nombre d'occurrences des mots présents dans un document ou une collection de documents. La Figure 2 reprend les codes de la Figure 1 en illustrant les étapes d'un processus MapReduce à l'aide de la tâche *WordCount*.

Le processus est assez intuitif à comprendre. Le rôle de la fonction *Map* est de découper la chaîne de caractères donnée en entrée pour analyser chaque mot. Ainsi, pour chaque mot, dénoté w, la fonction *Map* retourne $\langle w, 1\rangle$. Les mots w_i servent alors de clés pour l'étape de tri. Chaque fonction *Reduce* recevra alors en entrée une paire dont la clé sera un mot w et la valeur sera un tableau ; dénoté T_w, composé de 1. La taille de ce tableau sera le nombre d'occurrences du mot w dans les documents en entrée. La fonction *Reduce* est alors extrêmement simple et ne se contente que de renvoyer le mot w et la taille de T_w.

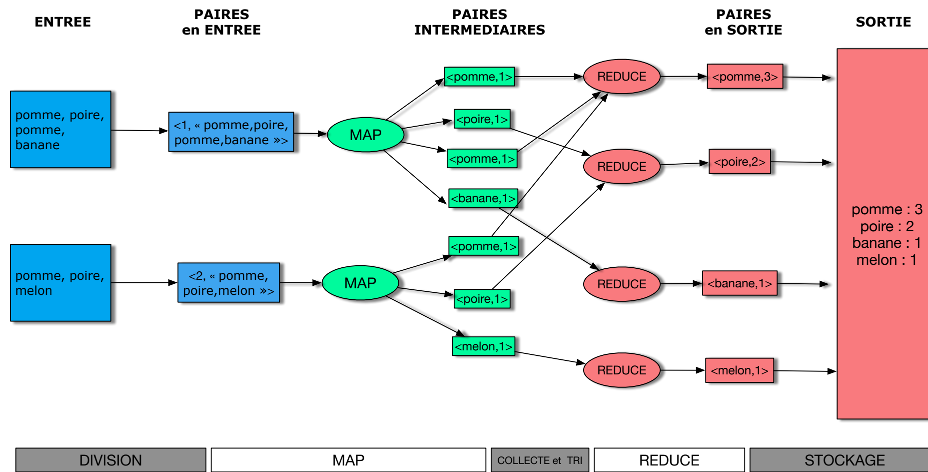

FIGURE 2 – Illustration des étapes d'un processus MAPREDUCE à l'aide de la tâche *WordCount*.

2.2 Code Java

Vous trouverez à l'adresse `www.irit.fr/~Yoann.Pitarch/Docs/SID/BigData/wc1.zip` une archive nécessaire à l'exécution de ce TP. Cette archive contient le code source Java, excessivement commenté, permettant d'exécuter cette tâche ainsi que les fichiers d'entrée du processus. Prenez le temps de le parcourir avec attention car tous les jobs MAPREDUCE que vous développerez respecteront cette structure.

2.3 Instructions

1. Lancez le cluster que vous avez créé lors du TP précédent ;
2. Téléchargez l'archive sur le nœud principal et décompressez là dans le répertoire de votre choix (par exemple dans `/home/hduser/TP/wc1`) ;
3. Copiez les données sur la partition HDFS et vérifiez leur présence ;
4. Compilez le code source Java et créez l'archive jar à l'aide des commandes :
   — `hadoop com.sun.tools.javac.Main WordCount1.java`
   — `jar cf wc1.jar WordCount*.class`
5. Lancez le job avec la commande
   `hadoop jar wc1.jar WordCount1  <INPUT_DIR> <OUTPUT_DIR>` ;

6. Vérifier que le job a correctement fait son travail en regardant le résultat produit.
7. L'exemple précédent est relativement inefficace en terme de communication entre les nœuds. En effet, chaque fonction *Map* émettra autant de paires qu'il y aura de mots dans le document qu'elle traite. Dans un environnement distribué, il est important de minimiser les coûts de communication et donc de réfléchir à des solutions qui minimisent le nombre de paires émises par la fonction *Map*. Adaptez le code précédent pour optimiser le processus de comptage de mots.

3 Ajout d'une blacklist dans le comptage d'occurrences

Aussi simple soit-elle, l'opération de comptage de mots trouve de multiples applications, notamment en Recherche d'Information où elle est au cœur des techniques d'indexation et de certains modèles pour représenter l'importance d'un mot dans un document. Dans un tel contexte, certains mots peuvent être vides de sens et les compter n'est pas pertinent. Nous allons donc voir comment passer un fichier en paramètre du programme MapReduce. Ce fichier comprendra une liste de mots (un mot par ligne) qui ne seront pas pris en compte lors de la phase de comptage. Si dans un contexte non distribué, inclure une telle liste de mots vides peut paraître triviale, cela se révèle plus difficile dans le cas présent. En effet, ce fichier doit être recopié sur chaque nœud exécutant une tâche *Map* ou *Reduce* (en fonction de quelle étape a besoin du fichier). Hadoop possède un mécanisme de gestion d'un cache distribué qui permet de rendre disponible, à tous les nœuds du cluster, des fichiers (texte, archives ou autre) en lecture seule dont l'application aurait besoin.

Les fichiers en cache peuvent être spécifiés dans le code Java via l'API en utilisant les fonctions `Job.addCacheFile(URI)`, `Job.addCacheArchive(URI)`, `Job.setCacheFiles(URI[])` et `Job.setCacheArchives(URI[])` où URI est de la forme `hdfs://host:port/absolute-path#fileName`. Vous trouverez à l'adresse `www.irit.fr/~Yoann.Pitarch/Docs/SID/BigData/WordCount2.java`, le code commenté du programme Java qui permet de prendre en compte une liste de mots vides.

Le code introduit trois nouveaux concepts :

1. La fonction `setup` qui permet d'effectuer quelques traitements sur chaque *Mapper* avant que la fonction *Map* ne s'exécute.
2. La notion de cache distribué comme discutée précédemment.
3. La notion de configuration qui permet de définir des paramètres qui seront connus par tous les nœuds du cluster.

3.1 Instructions

1. En reprenant ce qui a été dit lors du précédent exercice, compilez et lancez ce programme `WordCount2`. Testez son comportement.
2. Nous nous proposons d'ajouter une nouvelle fonctionnalité au programme à travers l'ajout d'une option : permettre de ne pas considérer la casse lors du comptage des mots. Modifiez le programme précédent pour ajouter cette fonctionnalité.

4 Quand les jobs s'enchaînent !

Jusqu'à présent, les programmes MapReduce que nous avons vus ne contenaient qu'un seul job. Néanmoins, il est des situations plus complexes (et réalistes) où une application MapReduce consiste en l'enchaînement de plusieurs jobs. Autrement dit, les sorties d'un job MapReduce deviennent les entrées d'un nouveau job MapReduce. Nous allons illustrer cela en programmant le calcul du TF-IDF. Dans la mesure où vous avez suivi (avec la plus grande attention) un (si vous êtes en M1) ou deux (si vous êtes en M2) modules de Recherche d'Information, cette mesure ne sera pas détaillée dans ce sujet. Schématiquement, le calcul du TF-IDF s'effectue en 3 temps, i.e., 3 jobs MapReduce :

1. Job 1 (`WordFrequenceInDocs`) : il s'agit de calculer pour chaque mot, son nombre d'occurrences par document. Ce job est très proche de celui que vous venez de programmer ;
2. Job 2 (`WordCountsForDocs`) : il s'agit de calculer le nombre de mots par documents ;
3. Job 3 (`WordsInCorpusTFIDF`) : il faut combiner ces informations pour calculer le TF-IDF pour chaque terme.

4.1 Instructions

1. Téléchargez l'archive à l'adresse `www.irit.fr/~Yoann.Pitarch/Docs/SID/BigData/tfidf.zip` et décompressez là. Vous trouverez à l'inté-

rieur les classes Java et les scripts nécessaires au calcul du TF-IDF.

2. Les fonctions *Map* et *Reduce* des 3 jobs MAPREDUCE ont volontairement été effacés. Seules les spécifications demeurent. Vous devez donc écrire ces fonctions.
3. Une fois l'écriture des fonctions terminée, compilez le code et créez l'archive jar avec la commande `ant` (installez `ant` via la commande `sudo yum install ant` si cette commande n'existe pas)
4. Lancez le script `run_tfidf.sh <INPUT_DIR> <OUTPUT_DIR>`
5. Vérifiez les sorties.