Indexation et recherche de contenu utilisant MapReduce de Google

Documentation technique

Elodie Corbel, Kévin M'Ghari,
Mickaël Olivier, Clarisse Renou

Encadrant : Alexandru Costan

1 Introduction

Indexation et recherche de contenu utilisant MapReduce de Google, voici l'intitulé de notre sujet. En fait, il s'agit simplement de la réalisation d'un moteur d'indexation avec son propre système d'indexation.

Cette documentation technique a pour but de vous présenter l'architecture de notre projet réalisé lors de notre troisième année à l'INSA de Rennes.

Pour se faire, nous allons premièrement vous présenter l'architecture globale. Puis, un listing des dossiers afin que vous vous visualisez mieux où se trouvent les différents composants de notre projet et comment l'utiliser. Enfin, nous vous présenterons en détails, le fonctionnnement de notre moteur de recherche.

Si vous avez besoin d'informations quant à l'utilisation et à l'installation de notre projet, veuillez vous reporter à la documentation utilisateur (disponible dans l'archive de notre projet).

2 Architecture globale

Depuis le début de projet, nous avons choisi de décomposer notre projet en trois composantes (*voir figure* **??**) :

- l'interface graphique, pour la communication avec l'utilisateur afin qu'il puisse entrer les mots qui veut rechercher.
- le moteur de recherche, pour la recherche dans l'index et les calculs
- l'indexation, réalisée par Hadoop, afin d'indexer le contenu

Voici le détail des différentes entrées-sorties :

2.1 Interface graphique

On a en entrée la requête entrée par l'utilisateur composée de un ou plusieurs mots.

En résultat, le moteur de recherche va nous donner les fichiers dans lesquels se trouve ce/ces mot/mots avec les lignes dans lesquelles il apparaît.

On utilise pour cela une JApplet, il faut donc pour l'ouvrir l'intégrer dans une page html. La version de Java requise dépend de celle installée avec votre navigateur. Il est cependant conseillé d'utiliser Java 1.6 pour compiler et exécuter l'applet.

FIGURE 1 – Architecture globale du projet

2.2 Moteur de recherche

Il se compose en 2 parties : une partie qui se charge de faire interface avec l'utilisateur et l'autre de construire une HashMap à partir de l'index pour avoir l'index en mémoire vive.

La partie qui fait interface avec l'utilisateur prend en entrée l'expression entrée par l'utilisateur et en sortie rend les lignes et les fichiers dans lesquels se trouve l'expression entrée.

La partie qui place l'index en mémoire vive, prend en entrée le fichier donné par l'indexation, le parcourt et donne en sortie une HashMap [1].

Le moteur de recherche est lié à l'applet et se trouve donc dans le même dossier.

2.3 Indexation

L'indexation est faite à partir du framework d'Apache : Hadoop.

Elle prend en entrée des fichiers texte, nous avons décidé que ce serait des livres libres de droit disponibles sur internet mais ce pourrait être n'importe quels autres fichiers contenant du texte.

En sortie, nous avons l'index construit par Hadoop, il s'agit d'un fichier texte contenant pour chaque ligne un mot du texte le fichier dans lequel il apparaît et les numéros de ligne des occurences de ce mot dans le fichier donné.

La version d'Hadoop que nous avons utilisé pour le projet est la 1.0.3. Vous trouverez un tutoriel de comment installer Hadoop à l'adresse suivante : `http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/`

1. voir partie n°.. pour le détail de la HashMap

3 Listing contenu dossiers

Pour une meilleure compréhension de notre projet, nous avons décidé de vous présenter le contenu des dossiers de l'archive :

```
total 28
drwxr-xr-x 6 hduser hadoop 4096 mai   11 11:03 hadoopMR
drwxr-xr-x 4 hduser hadoop 4096 mai   11 11:03 JavaDoc
drwxr-xr-x 4 hduser hadoop 4096 mai   11 11:03 MapReduce
drwxr-xr-x 5 hduser hadoop 4096 mai   11 11:03 Rapport LateX
drwxr-xr-x 5 hduser hadoop 4096 mai   11 11:03 SearchEngine
```

Le contenu listé au-dessus est la racine du projet où se trouve les différentes composantes. Le dossier `hadoopMR` est le dossier global où l'on met les exécutables et on lance le projet. Dans le répertoire `JavaDoc` se trouve toute la documentation que nous avons générée pour notre projet, elle y est au format PDF et html. Le dossier `MapReduce` contient tout ce qui concerne le coe source de la partie indexation du projet. Le dossier `Rapport LateX` contient ce présent rapport, le rapport final du projet et la documentation utilisateur. Et enfin, le dossier `SearchEngine` est le dossier dans lequel se trouve le code source du moteur de recherche.

```
./hadoopMR:
total 4392
drwxr-xr-x hduser hadoop   avril 30 15:38 inputFiles
drwxr-xr-x hduser hadoop   mai   11 11:02 inputFilesSplit
drwxr-xr-x hduser hadoop   mai    9 21:48 outputFiles
drwxr-xr-x hduser hadoop   mai    9 21:50 Page Web
-rw-r--r-- hduser hadoop   mai    9 20:40 hadoopIndex.jar
-rwxr-xr-x hduser hadoop   mai    9 21:25 scripthadoop.sh
-rwxr-xr-x hduser hadoop   mai    9 21:25 splitScript.sh
```

Comme vous pouvez le voir, l'exécutable `hadoopIndex.jar` sert à construire l'index. Il est lancé à partir du script `scripthadoop.sh` seul script à lancer pour exécuter l'indexation du contenu. Contenu qui, par ailleurs, doit être placé dans le dossier `inputFiles`.

```
./hadoopMR/Page Web:
total 6172
-rw-r--r-- 1 hduser hadoop 2920064 mai    9 21:50 applet.jar
-rw-r--r-- 1 hduser hadoop    1693 avril 30 15:38 blanc.jpeg
-rw-r--r-- 1 hduser hadoop 3329319 avril 30 15:38 color.jpg
-rw-r--r-- 1 hduser hadoop   16464 avril 30 15:38 hadoop.jpg
-rw-r--r-- 1 hduser hadoop   37705 avril 30 15:38 orange.jpg
-rw-r--r-- 1 hduser hadoop     354 avril 30 15:38 page.html
```

Le fichier `page.html` est la page web qu'il faut ouvrir pour lancer l'applet `applet.jar`.

```
./Rapport LateX:
total 12
drwxr-xr-x 2 hduser hadoop 4096 mai    9 15:47 Documentation technique
drwxr-xr-x 2 hduser hadoop 4096 mai   11 11:03 Documentation Utilisateur
drwxr-xr-x 2 hduser hadoop 4096 mai   11 11:03 Rapport Final
```

```
./MapReduce:
total 28
drwxr-xr-x 5 hduser hadoop 4096 mai    9 15:31 index

./MapReduce/index/src:
total 16
-rw-r--r-- 1 hduser hadoop 1106 mai    9 20:41 IndexDriver.java
-rw-r--r-- 1 hduser hadoop 2743 mai    8 16:37 IndexMapper.java
-rw-r--r-- 1 hduser hadoop 1133 mai    8 16:37 IndexReducer.java
-rw-r--r-- 1 hduser hadoop 1054 mai    7 18:20 NotSplit.java

./SearchEngine:
total 88
drwxr-xr-x 8 hduser hadoop  4096 mai    9 21:48 bin
drwxr-xr-x 8 hduser hadoop  4096 mai   11 11:03 src

./SearchEngine/src:
total 24
drwxr-xr-x 2 hduser hadoop 4096 mai   11 11:03 grammar
drwxr-xr-x 2 hduser hadoop 4096 mai   11 11:03 index
drwxr-xr-x 2 hduser hadoop 4096 mai   11 11:03 path
drwxr-xr-x 2 hduser hadoop 4096 mai   11 11:03 reader
drwxr-xr-x 2 hduser hadoop 4096 mai   11 11:03 search
drwxr-xr-x 2 hduser hadoop 4096 mai   11 11:03 window

./SearchEngine/src/grammar:
total 4
-rw-r--r-- 1 hduser hadoop 3776 mai   11 11:03 Index2.java

./SearchEngine/src/index:
total 8
-rw-r--r-- 1 hduser hadoop 2645 mai   11 11:03 IndexBuilder.java
-rw-r--r-- 1 hduser hadoop 1265 mai   11 11:03 Informations.java

./SearchEngine/src/path:
total 4
-rw-r--r-- 1 hduser hadoop 1696 mai   11 11:03 Paths.java

./SearchEngine/src/reader:
total 12
-rw-r--r-- 1 hduser hadoop 6430 mai   11 11:03 FileRead.java
-rw-r--r-- 1 hduser hadoop 1526 mai   11 11:03 SortLineNumbers.java

./SearchEngine/src/search:
total 24
-rw-r--r-- 1 hduser hadoop  816 mai   11 11:03 FoundInfos.java
-rw-r--r-- 1 hduser hadoop 4557 mai   11 11:03 Search.java
-rw-r--r-- 1 hduser hadoop 9973 mai   11 11:03 Seeker.java

./SearchEngine/src/window:
```

```
total 16
-rw-r--r-- 1 hduser hadoop 2593 mai   11 11:03 Fenetre.java
-rw-r--r-- 1 hduser hadoop 1938 mai   11 11:03 Logger.java
```

Tout le contenu que vous pouvez voir au-dessus constitue le code source de notre projet. Son organisation et son fonctionnement vous sera expliqué dans la partie suivante.

4 Architecture Détaillée

4.1 MapReduce

Toute la partie qui met en oeuvre le travail fournit par notre algorithme MapReduce sur un serveur distant d'hadoop est contitué de quatre classes en tout.

4.1.1 IndexDriver

Il s'agit là de la classe qui contient le main de notre algorithme. Ainsi, on y déclare tout d'abord un job auquel on associe la classe courante. Puis on lui passe le chemin des fichiers d'entrées et des fichiers de sortie en vérifiant que le nombre d'arguments lors de l'appel du main est bien de 2, en renvoyant une erreur dans le cas échéant.
Il reste ensuite à définir la classe de mappage (IndexMapper.class), de reducer (IndexReducer.class), ainsi que le format des valeurs que l'on écrira en sortie(Text.class, NoSplit.class) de manière à ce que le job tourne de manière autonome. Enfin, on attend que le job se termine en testant son état afin de sortir de la méthode main.

4.1.2 IndexMapper

La classe de mappage qui permet de découper les fichiers d'entrée. En entrée on a pour clé l'offset de la ligne par rapport au début du fichier et en valeur la ligne, en sortie on a pour clé intermédiaire le mot, le nom du fichier et en valeur le numéro de la ligne dans lequel il se trouve. C'est pour cette raison que la classe hérite de Mapper<LongWritable, Text, Text, Text>, les deux premiers paramètres représentant la clé et la valeur en entrée et respectivement en sortie pour les deux derniers. On crée un paramètre qui permet de compter les lignes en l'initialisant à 0. Ensuite, on réécrit une multitude de classe pour mapper, sahcant que l'on aura un mapper par fichier pour faciliter le découpage (qui sera en fait assuré de manière indépendante) :

Deux classes annexes sont présentes : supprimerPonctuation, qui supprime toute la ponctuation d'un String à l'aide d'un StringBuffer et motAIgnorer, qui teste si un mot passé en paramètre appartient à une liste de mot que l'on ne souhaite pas indexer, à savoir "et", "ou", "où", "de", "des", "d", "le", "les","l","la","je","il","au","aux","du","un","une","a","à","or","ni","que","si","y", "m","mon","ma","mes","me","ne", "nous","on","sa","ses","se","qui","s","t","ta", "tes","te","il","là","qu","sans","sur".

En outre, la classe principale map permet d'effectuer le traitement de mappage sur une ligne à partir de son offset et du texte que l'on y lit. On passe aussi le contexte hadoop comme paramètre. On ignore la casse en mettant le texte en

minuscules, on supprime la ponctuation avant de créer un StringTokenizer mots à l'aide du traitement effectué. On récupère en outre le nom du fichier actuel dans le contexte. Ensuite, on effectue une boucle sur toute la ligne en utilisant la fonction hasMoreTokens() de la classe StringTokenizer.
Dans cette boucle, on incrémente l'offset de la ligne et on récupère le mot sous forme de Text. S'il ne s'agit pas d'un mot dans la liste de mots à ignorer qui a été préétablie, alors on écrit dans le contexte le mot, le nom du fichier puis le numéro de la ligne, le tout étant séparé par des espaces.

4.1.3 IndexReducer

La classe de type reducer suit à peu prêt la même architecture que celle associée au mappage, ainsi elle hérite de Reducer<Text,Text,Text,Text>. La seule fonction ici est reduc, qui aggrégège les résultats du mapper afin d'avoir en résultat la liste des numéros de lignes associés à une entrée d'un mot dans un fichier texte. Son rôle dans le détail est de contruire un StringBuilder avec pour séparateur un simple espace, que l'on remplie pour chaque valeur passée en paramètre par un simple cast de celle-ci de Text en String, le tout suivi d'un espace. Pour finir, en utilisant la clé passée comme paramètre, on écrit ce StringBuilder sous forme de Text dans le le contexte.

4.1.4 NotSplit

Enfin, la classe NotSplit permet de gèrer les fichiers en entrée. Ici, elle sert à indiquer que l'on ne veut pas qu'hadoop découpe les fichiers comme ce traitement est fait de manière indépendante. Pour qu'un fichier ne soit pas découpé par hadoop afin d'obtenir les numéros de ligne, il nous suffit donc d'invoquer cette classe qui hérite en fait de FileInputFormat<K, V> et d'override deux méthodes. D'une part, on s'arrange pour que isSplittable renvoie toujours false. D'autre part, on s'arrange pour que createRecordReader retourne uniquement un nouvel objet qui est en fait un LineRecordReader casté en RecordReader<K,V>.

4.2 Indexation

On a notre index dans la mémoire morte grâce à mapreduce, il nous faut désormais le passer dans la mémoire vive de java, par exemple sous dorme de hashmap. Cette partie décrit la fois la construction de cet index.

4.2.1 Index

Cette classe va parcourir le fichier Index construit grâce à MapReduce et appeler les méthodes adéquatse pour construire la hashmap. Elle est statique afin d'avoir accès à la création d'index partout. scanLine est un Scanner qui va s'occuper mot par mot de la ligne pour construire la hashmap, builder est IndexBuilder chargé de construire la hashmap. Enfin, on lui associe un paramètre recherche de type S earch, qui permet de faire la recherche sur l'index que l'on spécifiera du côté utilisateur. Les méthodes de cette classe sont les suivantes :

La méthode scanFile scanne ligne par ligne le fichier index pour la transmettre au builder. Il s'agit d'une simple boucle qui vérifie si l'on est arrivé à la fin du fichier et appelle la méthode builder pour chaque ligne.
La méthode builder vérifie que la ligne suive la grammaire appropriée de notre index

et ajoute au builde dans le cas ou rien ne la contredir le mot, puis le fichier et la liste de lignes que l'on lit dans la ligne.
La méthode build enfin, initalise l'IndexBuilder et le BufferedReader pour lancer l'analyse du scanFile, le tout après avoir vérifié que le fichier d'index est présent, et après l'avoir ouvert. Enfin, elle récupère la recherche que l'on passe du côté utilisateur et lance dessus la méthode statique toDo qui effectue la dite recherche.

4.2.2 IndexBuilder

Cette classe s'occupe de construire l'index avec la création d'une hashmap. Ses paramètres sont index, la Map<Integer, HashMap<String, Informations» que nous avons utilisé pour stocker nos informations, currentWord, le mot courant, currentFile, le fichier courant, lines, l'ArrayList<Long> définissant la liste de lignes auxquelles on trouve le mot currentWord dans le fichier currentFile. Pour finir, currentID, un identifiant unique pour chaque combinaison (mot,nomFichier) généré automatiquement et nous permettant de trier la HashMap tout en jouant le rôle de hashcode.
Outre le constructeur, on trouve dans cette classe les méthodes addWord, addFile, addLine qui récupèrent respectivement currentWord, currentFile et lines. Enfin, la fonction buildset, lancée pour chaque ligne, ajoute dans l'index une entrée composée d'une part de currentID, et d'autre part d'une HashMap<String, Informations> elle même composée de currentWord, puis de la classe Informations dans laquelle on placera currentFile et lines.

4.2.3 Informations

Son seul intérêt est de permettre de mapper deux objets en même temps au lieu d'un (il s'agit d'une classe à deux paramètres : un String pour le fichier et une ArrayList<Long> pour les lignes). Seuls un constructeur, deux getters, deux setters (pour chaque paramètre) sont présents.

4.2.4 Paths

De même, le seul intérêt de cette classe est de stocker sous forme de static String les différents chemins utilisés par le moteur de recherche.

4.2.5 FileRead

Il s'agit de la classe qui récupère les lignes du fichier pour un mot donné, et permet d'afficher ce qu'obtiendra l'utilisateur après la recherche en mettant les mots intéressants en gras. Elle a comme paramètre le chemin d'accès aux fichiers découpés, le nombre de lignes par fichier découpés, que nous avons fixé à 100 dans nos essais. En outre, on retrouve d'une part fileName, lines qui sont le nom du fichier découpé et ses lignes, et d'autre part wordToSearch, qui est justement le mot recherché.
Les méthodes associées sont, en plus d'un constructeur :

getLinesText, qui pour chaque instance de lines va concaténer dans un StringBuilder les lignes auxquelles on trouve le mot wordToSearch.
getContextLine, qui prend comme paramètre une ligne pour l'analyser. De manière générale, la méthode concatène chaque mot de la ligne dans un StringBuilder en faisant de la mise en forme.

getFilePart, qui récupère le chemin où sont stockés les fichiers découpés et crée le découpage de manière automatique.
formatStringStrong, qui permet de mettre en forme le résultat en vue de l'affichage à l'aide des méthodes vues précédemment.

4.2.6 SortLineNumbers

Il s'agit juste d'une classe qui applique l'algorithme de tri rapide pour trier les numéros de ligne par ordre croissant. Elle a donc pour paramètre une List<Long> nommée listLignes. Ses méthodes sont le constructeur, getLinesSorted qui appelle juste quicksort, l'algorithme de tri générique, entre la première et la dernière ligne du fichier. Notons que quicksort, présente dans la classe, se sert de la fonction exchange, qui permet d'échanger deux instances dans listLignes.

4.3 Recherche

Cette partie décrit l'intégralité du moteur de recherche.

4.3.1 Search

Cette classe récupère l'expression entrée dans le moteur de recherche. Elle a comme paramètres toSeek, la List<String> des mots à chercher, expression, l'expression entrée dans le moteur qu'il faut analyser sous forme de String, et enfin seeker, qui est une instance de la classe Seeker décrite plus loin. En plus du constructer, des getters et des setters, les fonctions associées sont :

supprNonIndexe, qui permet de supprimer les mots passés dans le moteur de recherche qui sont trop courants pour vraiment intéresser l'utilisateur. Il s'agit des mêmes mots que ceux que l'on a pas indexé.

toDo, la fonction qui effectue tout le traitement de recherche. Elle permet tout d'abord d'analyser l'expression passée en la splittant et en appelant supprNonIndexe. Ensuite, pour chaque mot de l'expression, dans une boucle, la fonction lance les prédicats de recherche associés à AND, OR et NOT dans le cas où ils sont rencontrés. Dans le cas d'un AND ou d'un OR on effectuera ensuite un saut de deux mots, ceux associés à l'expression. Dans le cas du NOT, ce saut concerne un seul mot. Sinon, on recherche juste le mot passé en paramètre. Après ce traitement de base, le seeker est composés de deux liste : une liste de fichiers intéressants, et une liste de fichiers rebuts, que l'on souhaite cacher à l'utilisateur. Il nous suffit de faire une différence ensembliste et enfin, d'effectuer l'affichage pour chaque élément de la nouvelle liste ainsi obtenue.

4.3.2 Seeker

Il s'agit de la classe qui recherche dans la hashmap pour un mot donné. Ses paramètres sont un StringBuilder qui contiendra le message à afficher à l'écran de l'utilisateur, une List<FoundInfos> info qui contient les fichiers intéressants, tet une List<FoundInfos> intox qui contiendra les fichiers que l'ont veut éliminer des résultats de la recherche. En plus du constructeu, des getters et settersr, voici les différentes méthodes associées :

addMessage, qui concatène l'objet passé en paramètre au StringBuilder message

seek, qui permet de faire le parcourt de l'index pour un mot simple. La fonction vérifie si le mot est présent dans un des fichiers txt et si tel est le cas elle récupère son contexte dans info.

seekAnd permet de faire le parcourt de l'index pour deux mots et renvoie leurs contextes concernant des fichiers identiques. Il suffit pour celà de faire deux boucles imbriquées sur les listes de fichier associées à chacun des deux mots, de vérifier les cas où l'on tombe sur deux fichiers identiques pour dans ce cas récupérer le contexte associé.

seekOr permet de faire le parcourt de l'index pour deux mots et renvoie leurs contextes concernant des fichiers différents. Il suffit pour celà de faire deux boucles imbriquées sur les listes de fichier associées à chacun des deux mots, de vérifier les cas où l'on tombe sur deux fichiers différents pour dans ce cas récupérer le contexte associé.

seekNot effectue le même travail que seek mais renvoie le résultat de ses recherches dans intox et non dans info.

predicatInvalid permet de donner des informations sur le formalisme du prédicat utilisé en cas d'echec (voir partie moteur de recherche).

getMessage permet de retourner message sous forme de StringBuilder concaténé à la balise </html>

isPresent vérifie si le mot est présent dans la hashmap, en parcourant chaque instance de l'index

getFichiers rend une List<String> de fichiers qui est celle associee au mappage d'un mot

getNbOccurences rend le nombre d'occurences totales d'un mot dans l'index

getResult rend les numéros de lignes pour un mot, c'est-à-dire une Map dans laquelle on a pour clé le fichier dans lequel le mot se trouve et pour valeur la liste du numéro des lignesx. On construit cette map par parcourt de l'index.

getLinesText permet d'obtenir le texte associé aux lignes d'un mot dans un fichier, tout en mettant ledit mot en gras. C'est cette fonction associée au mappage de getResult qui nous permet de faire le tri dans la classe Search.

4.3.3 FoundInfos

Cette classe associe à un mot la liste de ses fichiers, elle sert juste de structure pour contenir un mot et la List<String> de ses fichiers.

4.4 Affichage et GUI

Cette artie décrit l'interface utilisateur, qui est une applet, ainsi que le logger qui permet de tracer l'exécution du programme.

4.4.1 Fenetre

Notre fenêtre est constituée d'un JTextField entree, d'un JTextPane resultat. Elle a aussi comme paramètres un Search toEvaluate et un String whereSearch que l'on associe à un chemin de la classe Path. Cette classe est uniquement constituée de la fonction init, qui tout d'abord crée le logger tout en lui donnant son contexte. Ensuite, elle décrit la fenêtre en termes de JFrame, auquel on ajoute un JButton, notre JTextField entree et ntre JTextPane resultat (sous forme de JScrollPane).

Après recentrage de la fenêtre, on crée un actionListener sur search qui vérifie les actions performées sur l'entree et récupère son texte lorsque l'on appuie sur le JButton. L'event déclenche aussi la construction de l'index, construction qui elle-même déclenche le toDo. Ensuite, l'actionListener met dans le JScrollPane resultat et le logger le message associé au seeker.

4.4.2 Logger

Cette classe en place un log sous forme de fichier texte de ce que souhaite afficher le programmeur, en l'occurence le traitement de notre requête. Elle a pour paramètres le nom du fichier de log, son chemin et un BufferedWriter pour écrire dans ce fichier. Les méthodes associées sont :

CreateLogger, qui permet d'initialiser le logger en indiquant le chemin de création du fichierpathLog et son nom logFileName.
addInLog, la méthode principale qui permet d'ajouter à la suite du fichier la phrase de log que l'on veut, ainsi que la date d'exécution.
formatDate, qui met la date e au format dd/MM/yyyy kk :mm :ss

5 Indexation

Une fois le traitement sur le texte pur effectué via MapReduce, on obtient donc un fichier texte qui contient ligne par ligne les entrées suivantes, séparées par des espaces : mot fichier suite de numéros de lignes associés.
Il faut donc reconstituer à partir de ce fichier beaucoup moins lourd que le texte initial un index dans la mémoire vive. Pour celà, nous vons choisi d'utiliser une Map<Integer, HashMap<String, Informations ».
Il s'agit en fait d'une Hashmap d'Hashmap contenant en clé principale un integer (pseudo hashcode permettant de parcourir simplement la structure) auquel on associe une autre HashMap, dont la clé (secondaire) est un String qui décrit le mot auquel est associée l'entrée. Enfin, on associe à ce String une classe Information, qui possède comme attributs un autre String, qui est un fichier dans lequel on trouve ce mot, et une ArrayList<Long> qui contient les numéros de lignes où l'on trouve le mot dans le fichier décrit.

5.1 Moteur de recherche

Le moteur de recherche va parcourir la hashmap reconstituée à partir du fichier texte produit par MapReduce afin d'en tirer des informations intéressantes. Tout d'abord, nous avons pensé à une recherche simple, d'une suite de mots. Il s'agissait là de découper l'expression passée par l'utilisateur afin d'accomplir une recherche mot par mot. Sur ce découpage, nous avons choisi d'éliminer certains mots courants de la lange française, à savoir les articles, les mots de liaisons et les pronoms.
Ensuite, la classe chargée de traiter l'expression, nommée search, passe la main à la classe seeker qui va pour chaque mot vérifier s'il s'agit d'une clé de la hashmap constituant l'index puis, si tel est le cas, renvoyer les entrées correspondantes (fichiers, numéros de lignes, nombre d'occurences). A partir de ces informations, on renvoie un affichage de chaque instance de ligne où le mot clé est trouvé, en donnant les numéros de ligne associés. L'ordre de l'affichage se base sur un critère de relevance qui concerne le nombre d'occurences de chaque mot clé pour chaque fichier. On réaffichera des parties d'un même fichier pour chaque mot, ce qui nous permet d'obtenir le contexte dans lequel ce mot a été trouvé.
Pour finir, l'idée nous est venue d'introduire quelque prédicats de base, à savoir AND, OR et NOT. Ils permettent respectivement de rechercher un fichier contenant deux mots, de rechercher des fichiers ne comprenant pas les deux mots passés en argument en même temps, et d'éliminer un mot de la recherche. Ils sont cummulables, et leur grammaire est de la forme suivante :

AND mot mot
OR mot mot
NOT mot

Pour mettre en place le AND comme le OR, il a fallut parcourir le résultat associés aux deux mots et éliminer les résultats ne correspondant pas - à savoir les fichiers ne contenant pas les deux mots dans le cas du AND, et inversement dans le cas du OR, ceux contenant les deux mots.
Pour le NOT, un cas plus particulier, il a fallu créer une deuxième liste dans la classe Seeker qui contienne les informations à éliminer. Après avoir traitéer tous les mots, dans la classe Search, il nous suffit alors d'enlever les instances de cette liste à la liste de recherche principale. Dans le cas où aucun NOT ne serait utilisé, cette liste est vide donc la méthode marche dans tous les cas.
Les seuls problèmes que nous avons rencontré dans cet partie sont purement des problèmes d'algorithme, d'organisation des structures de données et de lecture du texte passé en amont par l'utilisateur.

6 Conclusion

LaTeX c'est facile pour produire des documents standard et nickel ! Et BibTeX pour les références, c'est le pied.