Faculté des Sciences
Recherche Documentaire

Master 1 Informatique

Recherche Documentaire

Racine Mattieu - Jamet Jason - Grandjean Guillaume

Table des matières

1 Introduction

1.1 Cahier des charges

Le projet consistait à développer un moteur de recherche. En d'autres termes, il fallait pouvoir effectuer des recherches correspondant à un corpus de questions fourni à partir du corpus de documents donné. De plus nous devions concevoir ou utiliser une procédure de stemming [1]. Pour réaliser ces différentes choses, une interface conviviale devait être conçue, afin de permettre à un utilisateur de fournir sa requête et d'accèder facilement aux résultats. Enfin une stop list (stop words) devait être défini (contenant les mots les plus courants) et les requêtes données par l'utilisateur peuvent avoir différentes formes.

1.2 Outils utilisés

Pour ce projet, nous avons choisis d'utiliser le PHP. Il nous semblait évident que l'indexation, ainsi que les recherches seraient moins optimisées, cependant, nous pouvions proposer une belle interface en un minimum de temps.
Comme nous l'expliquerons plus tard, grâce à différents moyens et une certaine persévérance, nous avons réussi à optimiser au maximum l'indexation des documents.

Pour la gestion des données, et l'insertion de ces dernières dans l'interface, nous avons utilisé `Mysql`. Afin d'installer le serveur MySQL, le langage PHP5, mais aussi divers module, vous pouvez lancer la commande suivante dans un terminal :
`sudo apt-get install apache2 php5 mysql-server libapache2-mod-php5 php5-mysql`

Que se soient pour réaliser les différents éléments du sujet, ou afin d'optimiser l'indexation, plusieurs librairies ou extensions ont dû être installées :

Afin de pouvoir utiliser un Stemmer, l'extension Pecl devait être installée : `sudo apt-get install pecl`. C'est cette dernière qui nous permet d'installer la librairie Stem : `pecl install stem`.
Plusieurs choix de langage sont proposés lors de l'installation tel que l'anglais que nous avons utilisé. Après l'installation, il suffit de rajouter les quelques lignes suivantes :

`À copier dans le fichier php.ini (/etc/php5/apache2/php.ini)`

```
;;;;;;;;;;;;;;;;;;;;;
; Dynamic Extensions ;
;;;;;;;;;;;;;;;;;;;;;

extension=stem.so
extension=quickhash.so
```

Comme il le sera expliqué plus tard, c'est l'extension `quickhash.so` qui permettra une Optimisation de l'indexation. Son installation se fait par la commande suivante dans un terminal : `pecl install quickhash` tout en rajoutant quickhash.so dans php.ini.

De plus il faudra modifier le fichier `my.cnf` qui se trouve dans le dossier `/etc/mysql/` :

```
[mysqld]
tmp_table_size=256M
max_heap_table_size=256M
```

et enfin il faudra redémarrer le serveur avec la commande : `sudo service mysql restart`.

1. C'est un procédé de transformation des flexions en leur radical ou racine

En ce qui concerne la correction orthographique, qui est proposée à chaque recherche, elle est géréé par la bibliothèque `Pspell`. Son installation se fait grâce aux commandes suivantes (en ligne de commande) :

```
sudo apt-get install libpspell-dev
sudo apt-get install php5-pspell
sudo apt-get install aspell-en
```

Son utilisation se fait comme suit :

```
private function is_correct_word($word,$pspell_link)
    {
        if (pspell_check($pspell_link, $word))
            return true;
    ...
    }

private function verifcation_words($words){

$pspell_link = pspell_new("en");
...
@$sugg = pspell_suggest($pspell_link, $word)[0];
...
$NewRequest .= $sugg;
}
```

2 Fonctionnalités implémentées

2.1 Indexation

2.1.1 Tableaux en Mémoires

Lors de notre première approche, afin de concevoir une indexation correcte, nous avons décidé de créer un tableau associatif en mémoire dont chaque case contenait le mot et un autre tableau associatif contenant le nom et les positions du mot dans le document.

Cependant, nous nous sommes vite aperçu que la structure n'était pas optimisée pour la situation (au bout d'une heure l'indexation n'avait toujours pas atteint 1000 documents sur 3102). Pour pallier à ce problème nous avons donc pensé à une base de données.

2.1.2 Base de Données

On a donc utilisé une base de données dans le but de résoudre le problème de mémoire. En effet en `"MySql"` les tables peuvent être en `"Memory"` ce qui permet d'enregistrer les données en mémoire vive et non sur l'espace disque. Cela a pour effet d'augmenter la rapidité de traitement des requêtes `"SQL"` (Indexation réduite à 15 minutes).

Lors de notre première ébauche les fichiers contenant les documents sont `"parsés"` et les titres de ceux-ci sont insérés dans la base de données, chaque nouveau mot est ajouté dans un tableau et inséré dans un string qui représente la requête `"SQL"` a inséré à la fin de l'indexation dans la base de données.

A l'instar des nouveaux mots lus, la position de ceux-ci est insérée dans un string qui représente la requête `"SQL"` à exécuter dès la fin de l'indexation afin de les insérer dans la base de données. En effet le fait d'insérer les éléments en une seule requête au lieu d'ajouter élément par élément est plus optimisé.

2.1.3 Table de Hachage

Cependant, pour les 500 premiers documents cette méthode semblait rapide mais nous nous sommes vite aperçu que le fait d'ajouter un mot dans un tableau afin de vérifier que celui-ci avait déjà été traité n'était pas optimisé.

En effet, plus il y a d'élément dans le tableau plus le parcours est long. Nous avons décidé de chercher à optimiser ce mécanisme car nous savions que le `"PHP"` était un langage puissant et pouvait faire mieux que cela.

Nous avons donc après de nombreuses recherches sur internet trouvé la classe `"QuickHashStringInt"` qui gère les données non plus dans un tableau mais avec une table de hachage. Les résultats attendus ne se sont pas fait attendre. En effet cette dernière nous a permis de réduire le temps d'indexation à 8 secondes.

Toujours en quête d'optimisation, nous avons modifié quelques fonctions de `"PHP"` afin d'utiliser uniquement celle écrite en `"PERL"` (Plus rapide) ce qui nous a permis d'obtenir un temps de 6 secondes pour l'indexation.

2.1.4 Nouvelle Indexation

Les clients peuvent ajouter des nouveaux documents à indexer en les plaçant dans le répertoire prédéfini et relancer une indexation qui prendra en compte ces derniers.

Les tables sont vidées à chaque nouvelle indexation.

2.2 Traitement des requêtes

Il existe 3 types de requêtes :

— #new_index# : permet d'indexer les documents.

— #rapport# : permet d'afficher le rapport.

— Recherche de la requête dans les documents.

Lorsqu'un utilisateur lance une recherche (après qu'une première indexation ait été faite), nous enlevons d'abord les mots de la requête qui sont contenus dans la liste des StopWords, et de même nous enlevons les mots "or", "and", "not" (appelés KeyWords) en début et fin de chaîne car ils n'ont pas lieu d'y être. Nous effectuons enfin la requête sans les divers mots enlevés.

Cette requête se définit en plusieurs étapes. Tout d'abord, les KeyWords restant sont regroupés dans un tableau tandis que les derniers mots sont contenus dans un autre tableau.

`Exemple`

And his dictatorship and makes democracy
array(2) [0]=> array(2) [0]=> array(2) [0]=> string(3) "his" [1]=> string(12) "dictatorship" [1]=> array(2) [0]=> string(5) "makes" [1]=> string(9) "democracy" [1]=> array(1) [0]=> string(3) "and"
AP891216-0001&word=dictatorship,make,democraci&stopWord=his

C'est sur les mots contenus dans ce dernier tableau que l'on effectue le Stemmer, afin de rechercher tous les mots qui ressemblent à ceux recherchés.

`Exemple`

```
1 echo stem_english('judges'); //Returns the stem, "judg"
```

2.3 Pertinence des Documents

2.3.1 Calcul du taux de pertinence

Dans le but d'obtenir une liste de documents triée de façon cohérente, nous nous sommes basés sur la méthode "TF-IDF". Cette méthode prend en compte la fréquence des termes dans un document, la taille du document, et le nombre d'occurence des mots dans tous les documents. (somme pour chaque mot de (NombreOccurenceMotDansDocument / NombreMotsDansDocument) * log (NombreDocumentTotal / NombreDocumentOuMotsApparait)). Grâce à ce calcul, nous pouvons déterminer un taux de pertinance pour chaque document. Plus ce taux sera élevé, plus le document sera bien classé dans la liste des résultats.

Afin d'obtenir les documents les plus pertinents possible, nous avons décidé de prendre en compte les positions des différents mots trouvés, leurs ordres, et la présence de chacun d'entre eux.

2.3.2 Parcours de l'index

Suite à une phase d'étude sur la manière la plus performante de récupérer une liste de documents cohérente, nous sommes parvenus à deux solutions. La première consiste à n'utiliser que des requêtes SQL, afin de récupérer une liste de documents, déjà triée par taux de pertinence (calculs effectués dans la requête). Elle a pour avantage de pouvoir limiter le nombre de résultat récupéré. L'inconvénient est la complexité de la requête (beaucoup d'imbrications), et de ce fait, l'exécution de celle-ci est lente (jusqu'à 3 secondes pour charger 10 résultats).

```
SELECT name, (SUM(log) * COUNT(log) * COUNT(log) * COUNT(log)) as pagerank
FROM
   (SELECT *,
    (LOG(
           document / occurence_total
         ) * (occurence_dans_document / nombre_mot_doc)
        ) as log
    FROM
        (SELECT word.word, document.name,
         (SELECT count(*)
          FROM position
          WHERE position.id_word =
                word.id and position.id_document =
                document.id) as occurence_dans_document,
         (SELECT count(*) as nb_document
          FROM document) as document,
         (SELECT count(distinct id_document) as nb_occurence
          FROM position WHERE position.id_word = word.id) as occurence_total,
         (SELECT count(*)
          FROM position WHERE id_document = document.id) as nombre_mot_doc
        FROM document, position, word
        WHERE word IN ($qMarks)
        AND id_document = document.id
        AND id_word = word.id
        GROUP BY id_document, id_word) as newtable) as newtable2
        GROUP BY name
        ORDER BY `pagerank`  DESC
        LIMIT ".$limit_start.", ".$pagination."
```

La seconde solution utilise également des requêtes SQL, mais ici, seul la liste non triée de document est récupérée. Ce tri ce fait par la suite grâce à du PHP. La requête SQL sera donc beaucoup plus simple et exécutée de façon presque instantanée, les calculs effectués en PHP sont également très rapides (récupération et traitement des résultats en moins d'une seconde).

Requête :

```
SELECT word, document.name, position
FROM document, position, word
WHERE word  REGEXP ?
AND id_document = document.id
AND id_word = word.id
GROUP BY document.name, word, position
ORDER BY word.word ASC
```

Traitement :

```
foreach($idf as $word => &$arrayFileArrayPos) {
 $numberOfFile = count($arrayFileArrayPos);
 foreach($arrayFileArrayPos as $file => &$arrayPos) {
   $response = $bdd->prepare(" SELECT count(*)
                               FROM position
                               WHERE id_document =     (   SELECT id
                                                           FROM document
                                                           WHERE name = ?)
                             ");
   $response->execute(array($file));
   $bf = $response->fetch()[0];
   $log = (count($arrayPos)/$bf)*log(3102/$numberOfFile);
   $response->closeCursor();


   if(array_key_exists($file, $res)) {
     $res[$file] += $log*10 + (similarity($oldArrayPos[$file], $arrayPos));
   } else {
     $res[$file] = $log;
   }
   $oldArrayPos[$file] = $arrayPos;
 }
}
```

Après implémentation de ces solutions, nous avons fait le choix d'effectuer les traitements via PHP, nous pouvons ainsi avoir des réponses plus rapides.

2.3.3 Pagination

Dans un soucis de performance et d'ergonomie, nous avons décidé d'intégrer la notion de pagination. En effet l'affichage de la totalité des solutions étant extrêmement long, nous avons établi un affichage maximum de 10 résultats simultanés. La liste de tous les résultats est découpée en paquet de 10, sérialisée, puis stockée dans une variable de session. Ainsi, le parcours de l'index et les traitements ne sont effectués qu'une seule fois. Le changement de page se fait donc de façon presque instantané, puisque qu'il sagira seulement d'une lecture dans un tableau.

2.4 Interface de recherche

Le projet a été conçu de façon à proposer une interface conviviale. Elle permet de faire une recherche rapide, et d'accéder aux résultats tout aussi rapidement. À tout moment, l'utilisateur peut choisir de faire une nouvelle indexation ou d'afficher le pdf du rapport. Il n'a qu'à cliquer sur l'un des 2 liens proposés. Pour les autres requêtes demandées, le nombre de documents dans lesquels les mots recherchés apparaissent, est défini, et la liste de ces documents est affichée à la suite.

En cliquant sur le titre de l'un d'eux, le contenu du document est affiché, et l'on retrouve le nombre de fois que les mots de la requête apparaissent, et sont dans une couleur différente au texte. Les KeyWords sont aussi colorés car ils appartiennent bien à la requête.

3 Conclusion

Ce projet est pour le moins une réussite. En effet, nous sommes partis sur du Php afin de proposer une interface conviviale avec une utilisation facile, sans nous préoccuper de l'optimisation. Avec persévérence et beaucoup de patience, nous avons cherché à optimiser le plus possible l'indexation. Nous avons découvert la classe `"QuickHashStringInt"` qui permet de gérer les différentes données à partir d'une table de Hachage. Cela a eu pour conséquence de diminuer amplement le temps d'indexation. La recherche se fait donc de façon simple et rapide.