Projet SAR

RAPPORT

Recherche de filtres de Bloom similaires
Application à la recherche par mots clés basée sur une DHT

Réalisé par **NDOMBI TSHISUNGU** Christian & **DOAN** Cao Sang
Encadrant : M. **MAKPANGOU** Mesaac, Regal

26 Mars 2015

Table des matières

Chapitre 1

Introduction

*D*ans ce sujet, on abordera sur les recherches par mots clés. Une fois, on tape un mot ou une chaine de mots, l'application cherche dans la base de données pour trouver la liste des documents qui contient les descriptions correspondantes avec ces mots clés.

Dans la base de données qui contient des grosses masses de donées, les index sont basés sur les descriptions des documents. Chaque mot est une clé, chaque clé contient une liste des documents ayant ce mot. Au contraire à la liste normale, le mot clé est le nom du document et il contient la liste des mots qui se trouvent dans la description de ce document. Par exemple, le mot *PSAR : le projet pour les étudiants du M1*, normalement, le mot clé est *PSAR*, donc il contient sa description *le projet pour les étudiants du M1*, mais dans cette liste, les mots clés sont *le, projet, pour, les, étudiants, du, M1* et chaque mot contient le mot *PSAR*. On s'appelle la *liste inversée*.

Du coup, on utilise les tables de hachage distribuées (DHT) pour stocker les listes inversées. L'avantage d'utilisation de ces tables est que la complexité est égale à $\theta(1)$ car on trouve directement l'élément que l'on recherche. La statégie de distribution des listes inversées la plus utilisée (appelée partionnement vertical) consiste à associer à chaque terme une clé qui désigne le pair qu stocke la liste inversée associée à ce terme. Une fois que l'on lance la recherche sur les documents qui sont satisfaits à l'ensemble des mots que l'on a saisi, le système doit chercher pour chaque mot dans cet ensemble la liste des documents qui le contiennent, et puis fait l'intersection des listes inversées retrouvées.

Dans les systèmes pair-à-pair (P2P), chaque nœud contient un ou plusieurs mots clés, une fois qu'on cherche un mot, le système va demander le nœud où se trouve ces mots clés. Le problème est que si l'on cherche un ensemble de mots, il doit obligatoirement envoyer plusieurs requêtes à plusieurs nœuds pour récupérer les listes des documents et les faire l'intersection. Si on a un grand nombre de machines qui cherchent une quantité importante de mots clés, le système sera saturé. Et aussi, le problème se pose sur un ou

quelques nœuds précises, et les autres sont libres dans la plupart de temps. Donc, le coût de la recherche est d'autant plus élevé que la requête est précise et le déséquilibrage de charges des noeuds d'indexation du fait de la non-uniformité de la popularité des termes, ce sont des problèmes qu'on va rechercher la solution dans ce projet.

Chapitre 2

Projet

2.1 Marché

Il existe actuellement plusieurs moteurs de recherche par mots clés sur le marché. Il est toujours nécessaire d'améliorer les recherches en minimisant le temps de réponse et le coût de recherche, d'autant plus que les besoins de recherche deviennent très importants lorsqu'on exploite des bases de données de plus en plus grande et complexe. L'application que nous développerons doit répondre explicitement à ces exigences.

2.2 Contexte et objectifs

Comme l'application FreeCore[1], qui utilise le filtre de Bloom pour faciliter la recherche par mot clé, le but de notre projet est d'explorer d'autres solutions basées sur une recherche de filtre de Bloom similaires. Chaque filtre de Bloom sera vu comme un point dans un espace à n dimensions plutôt qu'une concaténation de n mots binaires. On détermine ensuite une relation de proximité et on exploite les algorithmes de recherche des filtres similaires pour réduire l'ensemble de filtres à examiner.

2.3 Enoncé du besoin

Comme FreeCore, l'application exploitera les propriétés des filtres de Bloom. Il permettra d'une part, de stocker les publications dans un fichier VA_file (put). Et d'autre part, d'effectuer des recherches de contenus par mots clés (search).

2.3.1 Stockage et indexation des publications

L'application permet de rechercher des publications qui contiennent tous les mots clés de la requête de façon optimale en utilisant le filtre de Bloom.

Elle doit :

— créer un filtre de Bloom correspondant à la description de la publication,
— créer un vecteur de n dimensions.
— créer les lieux où on stocke les différents documents,
— à partir d'un fichier de test, classer et indexer les documents dans les lieux correspondants,
— ajouter un nouveau document à partir d'un filtre de Bloom,

2.3.2 Recherche de contenus

Pour la recherche l'application créera un Filtre de Bloom des mots clés, représentants notre critère de recherche, c'est-à-dire les mots clés et devra :

— recherche un document à partir du filtre de Bloom,
— utilise le vecteur approximatif pour indexer et rechercher.

En outre, l'application doit assurer les services suivants :

— afficher les messages d'erreurs, s'il en existe,
— afficher les états de l'application,
— interagir avec l'utilisateur,
— faciliter les tests en utilisant les fichiers de test ou en utilisant les entrées saisit par l'utilisateur.

Note : Nous ne traiterons pas les erreurs possibles obtenues en cas de faute de frappe ou de faute d'orthographe, ni la différence de genre et de nombre des mots clés. Qui pourront faire objet d'une suite de ce travail.

Chapitre 3

Filtre de Bloom

Un filtre de Bloom est une structure de données probabiliste compacte inventée par Burton Howard Bloom en 1970 [1]. L'avantage d'utilisation de filtre de Bloom est que cette technique nous permet de savoir avec certitude que l'élément n'est pas présenté dans l'ensemble d'élément, c'est-à-dire il ne faut pas y avoir de faux négatif mais il peut y avoir des faux positifs. On ne peut savoir qu'avec une certaine probabilité, l'élément peut être présent dans l'ensemble. Ce qui réduit d'une manière considérable les entrées lorsqu'on fait une recherche dans une masse de données. En plus, la taille de filtre est fixe et indépendante du nombre d'éléments contenus, par contre, plus déléments plus de faux positifs.

En réalité, le filtre de Bloom a un structure très simple, un tableau B de m bits associé à i fonctions de hachage h_i, $0 \leq i \leq m-1$ permettant de mapper tout élément de l'ensemble à une des m cases du tableau. Ces fonctions de hachage ont une répartition uniforme des éléments de l'ensemble sur le tableau, et évidemment, doivent avoir une répartition différente. Au départ, le filtre représente un ensemble vide, et toutes les cases sont à 0.

Pour chaque clé k à ajouter à B, au lieu de se contenter de mettre à vrai la case B. $h(k)$ avec une seule fonction de hachage comme on le fait classiquement, on va mettre à vrai les m cases B. $h_i(k)$. Le principe étant que la probabilité que deux clés différentes aient les mêmes m valeurs pour leurs fonctions de hachage est faible.

Pour savoir si une clé est présente, on s'assura que les m cases de la table B correspondant aux valeurs des m fonctions de hachage sont posisitionnées à 1. Ce filtre est utile pour déterminer si un élément ne fait pas partie d'un ensemble, afin par exemple pour définir rapidement d'un traitement lourd lors de vérifier qu'une personne ne fasse pas partie d'une liste noire : d'abord, une vérification rapide avec le filtre de Bloom, puis en cas de potentiel posistif, un vérification plus certaine avec la comparaison dans la base de données.

1. Wikipédia

Le filtre de Bloom a quelques désavantages comme par exemple pour supprimer un élément dans l'ensemble de données, il nous faut reconstruire le filtre, en plus les faux posisitifs augmentent avec le nombre d'éléments présents dans l'ensemble. Nombreuses solutions utilisent des techniques probabilistes pour réduire le traitement d'information et leur coût.

Le filtre de Bloom et leurs variantes sont largement utilisés dans divers systèmes distribués. Plusiseurs recherches récentes et beaucoup de nouveaux algorithmes ont été proposés pour les systèmes distribués qui sont directement ou indirectement basés sur Bloom filtres[2].

Algorithme 1 *Insertion dans le filtre de Bloom*

IN : x objet à insérer dans le filtre de Bloom B
FUNCTION : *insert(x)*
OUT : $\emptyset$

---

for $i = 0...k-1$ **do**
    $j \leftarrow h_i(x)$
    if $B_j == 0$ **then**
        $B_j \leftarrow 1$
    end
end

Par exemple, supposons que nous souhaitions ajouter la clé "computer" dans la table B de taille 16 bits, que nous ayons 4 fonctions de hachage $h_i, 0 \leq i < 4$ et que h_0("*computer*") = 3, h_1("*computer*") = 8, h_2("*computer*") = 15, h_3("*computer*") = 10, h_4("*computer*") = 11. Donc, l'état de la table B après l'insertion sera :

15															0
1	0	0	0	1	1	0	1	0	0	0	0	1	0	0	0

TABLE 3.1 – Exemple filtre de Bloom

Algorithme 2 *Test d'appartenance d'un élément dans le filtre*

IN : x objet à tester dans le filtre de Bloom B
FUNCTION : *ismember(x)*
OUT : *bool*

$m \leftarrow true$
$i \leftarrow 0$
while m && $i \leq k - 1$ **do**
    $j \leftarrow h_i(x)$
    if $B_j == 0$ **then**
        $m \leftarrow false$
    end
    $i \leftarrow i + 1$
end return m

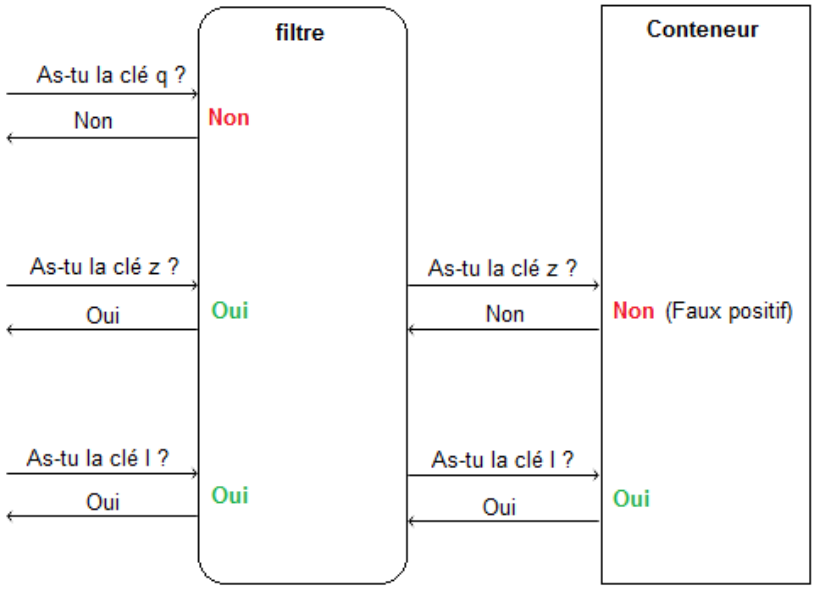

FIGURE 3.1 – isMember ?

Chapitre 4

Réalisation du travail

4.1 Notre solution

Nous avons choisi le filtre de Bloom de taille fixe de 512 bits. A partir des descriptions en entrée, ce programme transforme chaque mot clé en 1 bit avec la position précise dans notre filtre. Nous avons utilisé une seul fonction de hachage comme SHA 256 bits, cette fonction encode le mot clé en une chaine hexadécimale de 256 bits. Avec un simple calcul, le filtre est bien rempli. Nous avons choisi seulement une fonction de hachage car si on utilise plus de 2 fonctions, avec un seul mot, elle génère 2 bits différents, donc, si on a 200 mots clés, ce filtre devient être tout rempli par 1. Ce problème nous rend l'imposibité de réduire l'ensemble des données à recherche. Et pourquoi doit-on ne pas augmenter la taille de filtre ? Mais quelle taille suffit-il pour 200 descriptions si on a 3 fonctions de hachage ? Après plusieurs tests et discussions ensemble, nous avons pris une seule fonction de hachage très forte pour avoir moins de chance qu'il y a 2 mots clés différents qui ont la même valeur de hachage. En réalité, sur l'ensemble 107 459 documents, il existe au plus un document pour chaque filtre de Bloom.

Dans le cadre de notre projet, un document est une adresse URL décrit par l'ensemble de mots clés, ces mots clés sont écrits en codage UTF-8, sont basés sur plusieurs langues et contiennent les caractères spéciales. Donc, pour chaque filtre de Bloom, nous avons crée le fichier qui contient seulement cette adresse URL. En utilisant encore une fois la fonction de hachage SHA 256 bits, elle nous rend le nom unique pour chaque document et les stocke sur le disque.

Dans le cas le plus simple, pour ajouter un nouveau document, nous appliquons cette fonction de hachage pour chaque mot clé. En appliquant encore une fois cette fonction sur le filtre de Bloom généré dans le premier temps, nous avons le fichier qui contient ce document qui correspond avec ses mots clés. Ensuite, la recherche de document, facilement, on ré-applique cette fonction de hachage sur l'ensemble de mots clés pour trouver le filtre, à

partir de ce filtre, on peut trouver le nom du fichier qui contient ce document. Mais, cette solution marche si et seulement si on a exactement tous les mots clés dans la description d'un document, sinon cela ne marche jamais. Avec le filtre de Bloom, on peut savoir si ces mots clés existent dans l'ensemble des mots clés d'un document. Donc, pour chercher les documents qui contiennent ces mots clés, il faut regarder tous les filtres qui contiennent le filtre de la requête. Mais le parcours séquentiel est très lent sur le grand nombre de documents.

4.2 Index

Nous avons utilisé l'index pour indexer les filtres. Nous avons divisé chaque filtre de 512 bits en 64 morceaux de 8 bits. Pour chaque morceau, nous avons pris 4 bits, donc, à la fin, nous avons un nouveau filtre de taille 256 bits, la taille de ce nouveau filtre est égale à la moitié de l'ancien filtre. On applique cette façon pour tous les filtres existants, si deux filtres de taille 512 bits qui génèrent le même filtre de taille 256 bits, ces deux filtres ont même index. Pour chaque index, on crée un fichier qui correspond avec cet index (nous utilisons la fonction SHA 256 aussi pour générer le nom de ce fichier), contient les filtres de taille 512 bits qui ont cet index. Après la création des index, on trouve le nombre des fichiers index sont beaucoup moins que celui des fichiers de document. On a 2^{256} valeurs possibles pour les index au lieu de 2^{512} valeurs différents avec le filtre de 512 bits.

2^{256} est un nombre très grand, mais heureusement, un très grand nombre d'index ne est pas distribué. Pour éviter la recherche inutile sur les index inexistants, nous avons utilisé un tableau qui contient tous les index occupés. Ce tableau se situe dans le fichier VA_file. La recherche devient plus facile et plus rapide car on calcule le filtre de taille 256 bits à partir de celui de taille 512 bits, ensuite, on cherche dans le fichier VA_file les index qui contient l'index de la requête, grâce à l'ensemble des index trouvés, on peut trouver les fichiers qui correspondent avec ces index pour trouver les filtres de taille 512 bits qui contiennent notre filtre de 512 bits. Si oui, on trouve l'ensemble des documents qui contient la requête. On note le filtre de Bloom de taille k B^k, le filtre de la requête de taille k B^k_{req}.

Le problème est que le fichier VA_file sera explosé s'il y a des milliers d'index. Dans ce cas, pour résoudre le problème et pour ne pas faire agrandir le nombre de fichier à gérer, nous utilisons un nombre limite d'index pour chaque niveau d'index. Ce nombre est fixé à 1024 index. Une fois, le nombre d'index stocké dans le VA_file dépasse ce seuil, l'application crée un nouveau niveau d'index. Ce nouveau d'index est basé sur l'index plus bas. Par exemple, un filtre de Bloom de taille 256 bits B_1^{256} est l'index d'un filtre de Bloom de taille 512 bits B_0^{512}. Donc, pour chaque niveau i, le filtre créé doit satisfaire la formule suivant : $B_i^{\frac{512}{2^i}}$. Le système regarde le fichier VA_file

pour trouver les index existants et les transforme en nouveau niveau d'index. Chaque nouveau d'index correspond avec un fichier qui contient les index de niveau plus bas qui ont le même nouveau d'index. Le système va refaire chaque fois que le seuil est dépassé.

Est-ce que le niveau d'index est illimité ? La réponse est non. Pour le filtre de Bloom que nous avons choisi 512 bits, nous avons maximum 6 niveaux, B_6^8. Du coup, la taille de filtre de Bloom au niveau le plus haut est de 8 bits. A ce niveau, on a au maximum 256 valeurs d'index possibles dans le fichier VA_file, un nombre très petit par rapport au quantité de données.

La technique d'augmentation le niveau dynamique est très efficace si on a un nombre de données n'est pas important. Si on a 100 000 documents à ajouter, la création des fichiers nécessaires nous prend plusieurs heures pour les créer. Donc, si on connaît le nombre de données, on peut fixer combien de niveaux d'index nécessaire. Selon notre résultat sur un disque dur SSD, nous avons fixé 6 niveaux d'index, la création nous coûte seulement 15 minuites. Avec la curiosité, nous avons essayé de réaliser sur un disque dur 80Go utilisé l'interface SATA II, il prend plus de deux heures pour faire le même travail. Du coup, nous avons laissé à côté la technique de création dynamique, et fixé 6 niveaux d'index utilisés.

4.3 Recherche de données

Puisque l'on a fixé le nombre de niveau, on connaît la taille du filtre de Bloom contenu dans le fichier VA_file, sinon on peut lire ce fichier pour trouver quel niveau où on est. D'abord, le système crée le filtre de Bloom de taille 512 bits B_{req}^{512} pour la requête selon l'ensemble de mots clés fourni par l'utilisateur. A partir de ce filtre, le système calcul l'index de niveau le plus haut $B_{6_{req}}^8$ et lit le fichier VA_file pour trouver l'ensemble des index qui contiennent $B_{6_{req}}^8$. Une fois on a trouvé, le résultat est stocké dans un fichier res_6 qui correspond à ce niveau, ici c'est niveau 6. Ensuite, le système lit ce fichier de résultat pour trouver le nom des fichiers d'index à niveau 6 B_6^8, il accéde à ces fichiers et compare avec le filtre de la requête au niveau 5 $B_{5_{req}}^{16}$ avec les filtres trouvés. Car le fichier VA_file contient les index de niveau 6, chaque fichier qui correspond avec un index à niveau 6, contient les index de niveau 5, etc... Le résultat trouvé est stocké dans un fichier res_5.

Le système utilise la même façon en descendant vers le niveau 1. Le fichier res_1 contient les filtres de taille 256 bits B_1^{256}, en regardant les fichiers de B_1^{256}, on trouve les filtres de taille 512 bits B_0^{512} et les compare avec le filtre B_{req}^{512} de la requête, une fois que le filtre existant contient celui de la requête, on peut trouver facilement le nom du fichier qui contient le document cherché. Après la recherche, on trouve un ensemble de documents qui contient les mots clés de la requête. On ne peut éviter les faux positifs, on fait confiance à la puisance de fonction de hachage. Mais avec certitude, on a trouvé tous

les documents dont la description contiennent la requête.

Chapitre 5

Algorithme des fonctions

5.1 CREATE_FILTER

Algorithme 3 *Création d'un filtre de Bloom à partir d'un ensemble des descriptions*

IN : $\sum desc$
FUNCTION : *create_filter*($\sum desc$)
OUT : B^{512}

---

init(B^{512})
$x \leftarrow$ FIRST($\sum desc$)
while $x \neq \emptyset$ **do**
    $i \leftarrow$ SHA_256(x)
    $j \leftarrow i \ mod \ 512$
    $B^{512}[j] \leftarrow 1$
    $x \leftarrow$ NEXT($\sum desc$)
return B^{512}

5.2 PUT

Algorithme 4 *Ajout d'un filtre dans le système*

IN : filtre de Bloom de taille 512 bits B^{512}
FUNCTION : *put*(B^{512})
OUT : $\emptyset$

$i \leftarrow$ MAX_LEVEL
$vector_i \leftarrow$ CREATE_VECTOR(B^{512}, i)
$x \leftarrow$ FIRST(*VA_file*)
while $x \neq \emptyset$ **do**
    if $vector_i = x$ **then**
        BREAK
    end
    $x \leftarrow$ NEXT(*VA_file*)
end
if $vector_i \neq x$ **then**
    VA_file $\leftarrow$ ADD($vector_i$)
end
for $i = MAX\_LEVEL$ ... 1 **do**
    if $i = 1$ **then**
        $vector_i \leftarrow$ CREATE_VECTOR(B^{512}, i)
        CREATE_FILE($vector_i$, B^{512})
    else
        $vector_i \leftarrow$ CREATE_VECTOR(B^{512}, i)
        CREATE_FILE($vector_i$, CREATE_VECTOR(B^{512}, *i - 1*))
end
return $\emptyset$

5.3 SEARCH

Algorithme 5 *Recherche d'un document à partir d'un filtre de Bloom*

IN : filtre de Bloom de taille 512 bits B^{512}
FUNCTION : *search*(B^{512})
OUT : $\sum doc$

$i \leftarrow$ MAX_LEVEL
$vector_i \leftarrow$ CREATE_VECTOR(B^{512}, i)
$tmp \leftarrow$ CREATE_FILE(i)
$x \leftarrow$ FIRST(*VA_file*)
while $x \neq \emptyset$ **do**
    if $vector_i \subseteq x$ then
        tmp $\leftarrow$ ADD(x)
    end
    $x \leftarrow$ NEXT(*VA_file*)
end
for $i = MAX\_LEVEL - 1$ ... 1 **do**
    $vector_i \leftarrow$ CREATE_VECTOR(B^{512}, i)
    $x \leftarrow$ FIRST(FILE(i + *1*))
    $tmp \leftarrow$ CREATE_FILE(i)
    while $x \neq \emptyset$ **do**
        if $vector_i \subseteq x$ **then**
            tmp $\leftarrow$ ADD(x)
        end
        $x \leftarrow$ NEXT(FILE(i + *1*))
    end
end
$x \leftarrow$ FIRST(FILE(*1*))
while $x \neq \emptyset$ **do**
    $y \leftarrow$ FIRST(FILE(x))
    while $y \neq \emptyset$ **do**
        if $B^{512} \subseteq y$ **then**
            $\sum doc \leftarrow$ FIRST(FILE(y))
        end
        $y \leftarrow$ NEXT(FILE(x))
    end
    $x \leftarrow$ NEXT(FILE(*1*))
end
return $\sum doc$

Chapitre 6

Résultat obtenu

6.1 Données

Nous avons testé sur 107 459 documents, cette application a généré 320 000 fichiers nécessaires. 107 459 documents sont sous forme un lien URL. Au début, ces lien sont stockés dans un fichier de test. Nous avons testé le programme sur le disque dur SATA II et un SSD. La création des fichiers nécessaires nous coûte plus de deux heures sur un disque dur traditionnel au lieu de 15 minuites sur un SSD. Du coup, nous avons choisi d'utiliser le SSD dans tous nos tests. Avant de chaque test, les fichiers temporaires se sont vidés.

6.2 Recherche aléatoire

Nous réalisons la requête avec les mots clés choisis aléatoirement, la quantité de mots clés augmente chaque fois nous relançons la requête. Nous constatons que, plus de mots clés, moins de temps d'attente la réponse. Dans la figure 6.1, nous testons avec un à 20 mots clés.

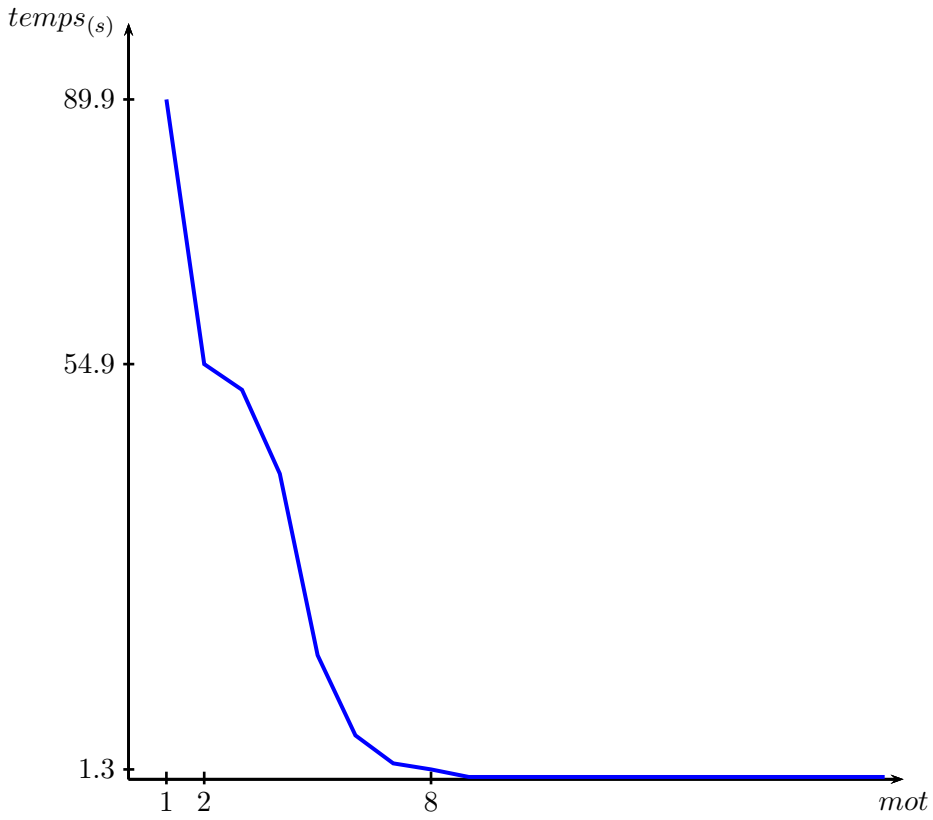

FIGURE 6.1 – Recherche aléatoire

6.3 Recherche sélective

Pendant les tests, nous voyons que avec la même quantité de mots clés dans la même description d'un document quelconque, le temps nécessaire pour avoir la réponse varie. Nous essayons plusieurs test avec la quantité de mots clés fixe, nous changons seulement les mots clés qui appartiennent dans la même description. Le résultat est intéressant.

Dans le premier test, nous choissons l'ensemble des mots clés qui appartiennent à plusieurs descriptions, ces mots clés sont appelés "populaire". Le deuxième test, nous prennons que les mots clés spécifiques qui sont caractéristiques pour chaque document dans la description d'un document quelconque. La description d'un document dans ces deux tests reste identique, ce qui change est les mots clés utilisés. Nous commençons par 1 mot et chaque fois nous relançons la requête, nous augmentons le nombre de mots clés. Dans le premier test, le nombre de mots clés augmenté ne signifie que le temps va être réduit.Dans le graphe, nous montrons que si on ajoute un mot "très populaire", ce mot nous coûte plus de temps. Dans le deuxième test, si on ajoute un mot "populaire", ce mot casse l'advantage de la recherche sélective. Donc, pour mieux rechercher un document, il faut que nous cherchions plusieurs mots clés qui caractérisent le document. Dans la figure 6.2, nous testons sur les requête d'un à sept mots clés.

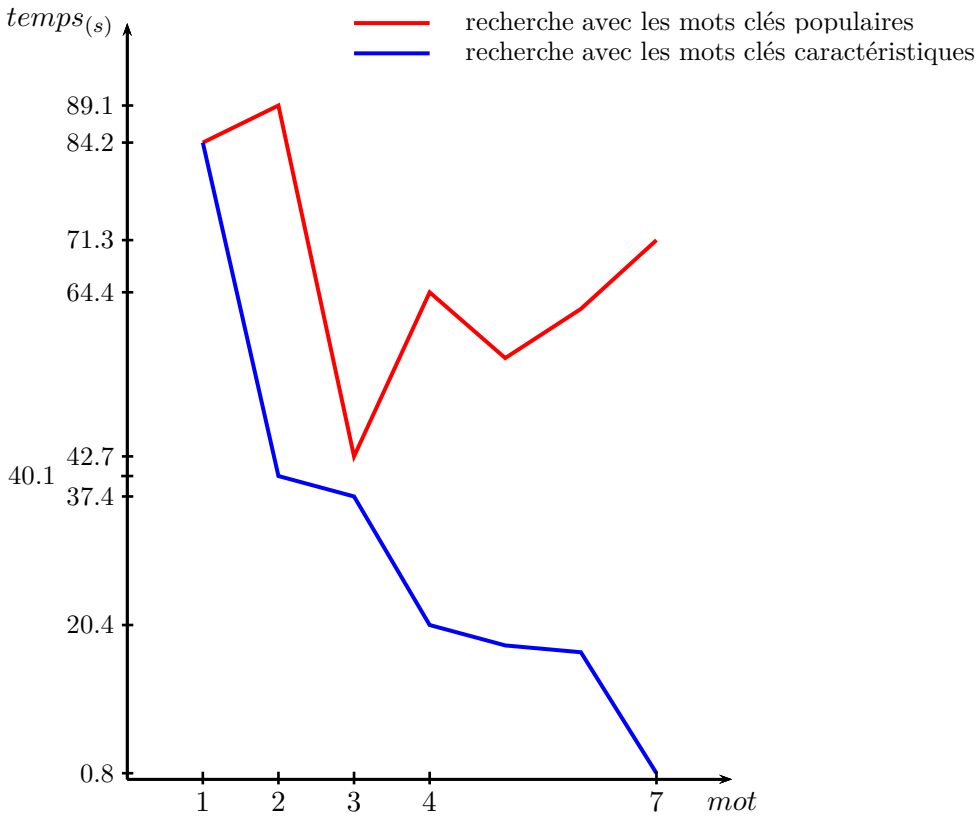

FIGURE 6.2 – Recherche sélective

Chapitre 7

Conclusion

*A*près ce projet, notre technique de recherche a besoin d'améliorer, le temps de réponse est très important. Car a création d'un vecteur est une sorte de vecteur approximatif, mais ce vecteur ne caractérise pas forcément les caractéristiques du filtre donné. En plus, notre programme est écrit en C, donc la mémoire louée pour chaque programme est limité, nous ne pouvons stocker toutes les données intermédiaire dans la mémoire. Du coup, nous travaillons sur le disque dont le temps d'accès de disque nous coûte cher. Chaque fois nous obtenons un résultat intermédiaire, nous devons écrire sur le disque sous forme d'un fichier.

Bibliographie

[1] Mesaac Makpangou, Bassirou Ngom, Samba Ndiaye : ***Freecore : Un substrat d'indexation des filtres de Bloom fragmentés pour la recherche par mots clés***. ComPAS'2014, Apr 2014, Neuchâtel, Switzerland

[2] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz : ***Theory and Practice of Bloom Filters for Distributed Systems***. Communications Surveys & Tutorials, IEEE. pp. 131 – 155. Fevrier. 2012