FreeCore : Un substrat d'indexation des filtres de Bloom fragmentés pour la recherche par mots clés

Mesaac Makpangou, Bassirou Ngom, Samba Ndiaye

ComPAS'2014 : Parallélisme/ Architecture / Système
Neuchâtel, Suisse, du 23 au 25 avril 2014

FreeCore : Un substrat d'indexation des filtres de Bloom fragmentés pour la recherche par mots clés

Mesaac Makpangou [1], Bassirou Ngom [1,2], Samba Ndiaye [2]

1 : Equipe REGAL/Laboratoire d'Informatique de paris 6
Université Pierre et Marie Curie
Paris, FRANCE
prenom.nom@lip6.fr
2 : Département Mathématiques-Informatique
Université Cheikh Anta Diop
Dakar, SENEGAL
{bassirou83.ngom, samba.ndiaye}@ucad.edu.sn

Résumé
Le support efficace de la recherche par mots clés est essentiel pour une bonne exploitation des réseaux de stockage pair-à-pair structurés. Un nombre important de solutions existent dans la littérature, toutefois elles sont confrontées aux problèmes de performance inhérents au schéma d'indexation mis en œuvre. Ce papier présente *FreeCore*, un substrat d'indexation de filtres de Bloom fragmentés et de recherche par mots clés. Les contributions de ce travail sont au nombre de trois. La réalisation d'un système de stockage offrant une interface qui permet d'associer une description à chaque contenu. La clé de stockage d'un contenu est déterminée à partir du filtre de Bloom de sa description. Cette facilité permet de ramener la recherche par mots clés au problème de recherche des clés de stockage qui matchent un filtre de Bloom. En second lieu, la construction d'un index distribué dont le coût de maintenance est indépendant du nombre de mots clés fournis lors d'une publication. Enfin, une méthode de recherche d'information à base de mots clés dont le coût est indépendant du nombre de mots clés fournis. Les propriétés découlant des choix opérés et les résultats des évaluations font de *FreeCore* une brique de base pour des applications et systèmes désirant un support efficace de la recherche par mots clés.

Mots-clés : Recherche d'information, table de hachage distribuée, filtre de Bloom, indexation

---

1. Introduction

La recherche d'information par mots clés est essentielle pour une bonne exploitation de certains réseaux de stockage pair-à-pair structurés. Or, les systèmes pair-à-pair structurés sont optimisés pour des recherches exactes. Au cours de la décennie écoulée, plusieurs systèmes de recherche par mots clés des contenus des réseaux pair-à-pairs structurés ont été proposés [1, 13, 7, 15, 12, 20, 5, 16]. Ces systèmes exploitent chacun un index inversé distribué qui permet, étant donné un ensemble de termes (ou mots clés), de localiser les documents qui contiennent cet ensemble. Nous distinguons deux types d'indexes selon que l'on maintient une liste inversée par terme ou par ensemble de termes.

Les systèmes d'indexation maintenant une liste inversée par terme sont confrontés au problème du coût de la recherche avec plusieurs mots clés. Pour effectuer une recherche concernant plusieurs mots clés, on doit retrouver les listes inversées associées à ces mots clés et faire ensuite l'intersection de ces listes. La solution intuitive consiste à visiter successivement tous les pairs en transmettant à chacun l'intersection des listes stockées par les pairs déjà visités. Les travaux [8, 15] ont démontré que ce protocole de base avait un coût très élevé en terme de bande passante.
Plutôt que de transmettre les listes inversées, Reynolds et al [12] proposent de transmettre dans un premier temps les filtres de Bloom représentant l'intersection des listes ; le dernier pair visité retourne à son prédécesseur les éléments qu'il considère dans l'intersection, puis chaque pair retraversé élimine les références qui ne sont pas dans sa liste et envoie le reste à son prédécesseur. Au final, les listes transmises sur le réseau sont plus courtes, mais la latence de la recherche est 2 fois plus importante que lorsqu'on n'utilise pas les filtres de Bloom. Proof [20] maintient les résumés des documents contenant un mot clé dans la liste inversée associée à ce terme. Pour réaliser l'intersection des listes, Proof calcule et transmet les intersections de pair en pair ; toutefois, la référence d'un document est incluse dans la liste à transmettre au prochain pair que si les mots clés de la requête sont dans le résumé de ce document. Cette technique permet de réduire les listes transportées à travers le réseau, au prix d'un index plus volumineux.
Malgré l'existence de plusieurs techniques permettant de réduire la bande passante consommée pour réaliser l'intersection des listes inversées, les systèmes d'index maintenant une liste inversée par mot clé restent confronter à des problèmes inhérents à leur nature. En particulier, le coût de maintenance de l'index distribué augmente avec le nombre de mots clés par document ; le coût de la recherche est d'autant plus élevé que le nombre de mots clés de la requête est élevé ; enfin, le déséquilibrage de charge des nœuds d'indexation du fait de la non-uniformité de la popularité des termes. Pour remédier à ces problèmes, des systèmes d'index associant des listes à des ensembles de termes ont été proposés. Ces systèmes permettent de retrouver directement la liste associée à un ensemble de mots clés. Joung et al [5] utilisent un hyper cube pour distribuer les descriptions des documents à travers un réseau de serveurs d'index. Pour cela, ils définissent r fonctions de hachage qui permettent d'associer à chaque ensemble de mots clés un vecteur de r bits ; chaque vecteur de r bits identifie un sommet de l'hyper cube. Ensuite, ils définissent aussi une fonction qui associe un identifiant de serveur d'index à chaque sommet de l'hyper cube. Chaque description est stockée par le serveur d'index associé au sommet de l'hyper cube dont l'identifiant est le vecteur associé à cet ensemble de mots clés. Pour retrouver les références des documents correspondant à un ensemble de mots clés, il suffit de calculer le vecteur de r bits correspondant à cet ensemble et d'accéder au serveur d'index correspondant au sommet identifié par le vecteur associé à la requête. Pour retrouver tous les documents contenant les mots clés fournis, on doit visiter tous les serveurs d'index qui stockent des sur-ensembles de l'ensemble de mots clés fournis dans la requête. Ainsi, plus r est grand, plus le coût de la recherche de sur-ensembles est élevé. Pour que ce coût n'explose pas, les auteurs suggèrent d'utiliser des valeurs de r inférieures à 10. Ceci est approprié si on souhaite distribuer les descriptions sur un petit groupe de serveurs.
Ce papier présente *FreeCore*, un substrat d'indexation et de recherche de contenus par mots clés. *FreeCore* permet, d'une part de publier des contenus et leurs descriptions, d'autre part de découvrir des ressources publiées satisfaisant des mots clés. *FreeCore* stocke les publications (c'est-à-dire les descriptions et contenus décrits) dans une table de hachage distribuée. *FreeCore* utilise le filtre de Bloom représentant une description comme clé de stockage de cette description et du contenu qu'elle décrit. *FreeCore* maintient aussi un index de filtres de Bloom

fragmentés correspondant aux publications. Intuitivement, un filtre de Bloom fragmenté (FBF) d'une publication est un ensemble de mots clés binaires de même taille constituant une partition du filtre de Bloom associé à cette publication, plus la spécification des positions occupées par chaque mot clé binaire et une estampille identifiant de façon unique la publication concernée. Toutefois, cet index maintient pour chaque mot clé binaire une liste de fragments (plutôt que de filtres de Bloom fragmentés) contenant ce mot binaire.
Grâce d'une part à la représentation des descriptions par des filtres de Bloom et d'autre part à la fragmentation de ces filtres en un nombre fixe de vecteurs de bits, les coûts de maintenance de l'index et de résolution des requêtes de recherche sont indépendants du nombre de mots clés dans les descriptions des documents et dans les requêtes de recherche. Les propriétés des fonctions de hachage utilisées d'une part pour construire les filtres de Bloom et d'autre part pour mapper des clés de stockage à des identifiants des pairs garantissent que la charge d'indexation est uniformément répartie sur les différents pairs. Comparée à la solution décrite dans [5] les publications peuvent être distribuées sur un très grand nombre de serveurs sans nuire à la performance des recherches globales car ceci dépend uniquement de la taille des mots binaires et non de celle de clés de stockage.
Le reste du papier est organisé comme suit. La section 2 présente le système *FreeCore*. Nous y détaillons le système d'indexation, puis décrivons les algorithmes de recherche mis en œuvre par *FreeCore*. La section 3 décrit les travaux comparables. Dans la section 4, nous présentons l'évaluation du système, puis la section 5 tire quelques conclusions.

2. FreeCore

FreeCore est un substrat d'indexation et de recherche par mots clés. D'une part, *FreeCore* permet de publier des contenus et leurs descriptions. Un contenu est une suite de bits de taille quelconque. En pratique, un contenu peut être un lien vers une ressource Web, l'adresse de stockage d'un objet, ou un objet quelconque. Une description est un ensemble de chaînes de caractères (ou mots clés). Une description est de taille quelconque. D'autre part, *FreeCore* permet aussi d'effectuer des recherches de contenus par mots clés. Nous distinguons deux types de recherche : *recherche exacte* et *recherche globale*. La recherche exacte (resp. globale) retourne les références des contenus dont les descriptions sont égales à (resp. sont des sur-ensembles de) l'ensemble des mots clés de la requête.
La section 2.1 détaille comment *FreeCore* stocke et indexe les publications fournies par ses utilisateurs. La section 2.2 montre ensuite comment *FreeCore* exploite son index pour réaliser les deux types de recherche.

2.1. Stockage et indexation des publications

D'un point de vue système, *FreeCore* est un système pair-à-pair de découverte de contenus. *FreeCore* permet à ses utilisateurs d'associer des descriptions à des contenus. Toutefois, *FreeCore* ne garantit ni la pertinence des descriptions, ni l'unicité de la description d'un contenu. Par conséquent, des descriptions différentes peuvent être associées à un même contenu ou une même description peut être associée à des contenus différents.

2.1.1. Stockage des publications

Nous désignons par publication tout triplet comprenant : un contenu, un ensemble de mots clés, et une estampille qui identifie de façon unique cette publication parmi toutes les autres publications.

FreeCore exploite une table de hachage distribuée (DHT) pour stocker les publications. *FreeCore* considère des chaînes de bits comme clés de stockage et définit une fonction de hachage qui permet d'associer une entrée de la DHT à chaque clé de stockage.
La clé de stockage d'une publication est égale au filtre de Bloom des mots clés de sa description. Les filtres de Bloom représentant les différentes descriptions ont la même taille et sont construits à l'aide des mêmes fonctions de hachage. Dans le reste du document, t_{fb} désigne la taille des filtres de Bloom représentant les descriptions des contenus.
Comme la plupart des systèmes de stockage pair-à-pair qui reposent sur une table de hachage distribuée, *FreeCore* définit aussi une fonction qui associe à chaque entrée de la table de hachage un identificateur de serveur responsable du stockage des publications correspondant à cette entrée. Ainsi, toutes les publications ayant la même description correspondent à la même entrée de la DHT et sont stockés par un même serveur.

2.1.2. Indexation des publications

Le principe est de construire un index qui associe à chaque mot clé les publications qui contiennent ce mot clé. Pour satisfaire ce principe tout en rendant les coûts de maintenance (resp. recherche) indépendants du nombre de mots clés de la description (resp. recherche), *FreeCore* assimile chaque publication à son estampille et au filtre de Bloom de la description qui lui est associée.
FreeCore fragmente chaque filtre de Bloom représentant la description associée à une publication en m morceaux de même taille que nous appelons des chunks ; chaque chunk est un vecteur de c bits, avec $c = t_{fb}/m$. Chaque chunk est caractérisé par sa valeur (un mot binaire de c bits) et son rang (un numéro compris entre 1 et m). Nous convenons que le chunk de rang 1 correspond aux bits de poids fort du filtre de Bloom qui le contient. Nous appelons fragment de filtre de Bloom de publication (ou simplement fragment de publication) tout triplet (chunk, rang, estampille) tel que : $chunk$ est un vecteur de c bits ; $rang$ désigne une position entre 1 et m ; $estampille$ correspond à une estampille de publication. Nous désignons par *filtre de Bloom fragmenté*, l'ensemble comprenant les m fragments distincts d'une même publication.
Plutôt que de maintenir un index qui associe à chaque mot clé une liste (inversés) des publications le contenant, *FreeCore* associe à chaque vecteur de c bits la liste de fragments de publications le contenant. Ici, les mots clés sont les vecteurs de c bits et la liste inversée contient les fragments. Un identifiant de serveur d'index est associé à chaque vecteur de c bits. La liste de fragments correspondant à un vecteur de c bits est assignée au serveur d'index associé à ce vecteur.

2.1.3. Algorithme de publication

L'algorithme 1 décrit le traitement d'une publication par *FreeCore*. Ce traitement comprend 2 parties : le stockage de la publication et l'ajout des fragments de cette publication dans l'index. Les lignes 1 à 4 correspondent à la partie stockage de la publication alors que les lignes 5 à 9 créent les fragments de cette publication et les ajoutent dans l'index. On notera que la clé de stockage de chaque fragment est le chunk (vecteur de c bits) associé à ce fragment.

Algorithm 1 stockage et indexation d'une publication

```
Input: contenu, keywords
1: EstampillePublication ts ⟵ calculEstampille() ;
2: BloomFilter summary ⟵ calculFiltreBloom(keywords)
3: Publication publication = creerPublication(ts, contenu, keywords) ;
4: put(summary, publication) ;
5: for i = 0 → m do
6:     chunks[i] = bf.substring(i ∗ c, c ∗ (i + 1) − 1) ;
7:     Fragment frag = créerFragment(chunks[i], i+1, ts) ;
8:     put(chunk[i], frag) ;
9: end for
```

2.2. Recherche de contenus

Etant donné un ensemble de mots clés, nous recherchons les données associées à cet ensemble (recherche exacte), ou à tout sur-ensemble de cet ensemble (recherche globale). La résolution de cette requête se fait en deux étapes : d'abord la localisation des publications susceptibles de contenir les mots clés de la requête, puis le classement des descriptions localisées par rapport à la requête. Les données associées aux *top-k* descriptions sont retournées au demandeur.

2.2.1. Localisation des publications

La localisation d'une publication consiste à déterminer la clé de stockage de cette publication.

Pour commencer, *FreeCore* calcule $query$, le filtre de Bloom représentant les mots clés contenus dans la requête. Pour ce calcul, il utilise la même taille de filtre de Bloom et les mêmes fonctions de hachage que pour le calcul des filtres de Bloom associés aux descriptions contenues dans les publications.

Par construction des clés de stockage des publications, $query$ est la clé de stockage des publications ayant comme description l'ensemble des mots clés de la requête. Ainsi pour une recherche exacte, la localisation des publications revient simplement à calculer le filtre de Bloom qui contient tous les mots clés de la requête.

Soient deux chaînes de t bits, u et v, nous disons que v *matche* u *(ou que* u *décrit* v*) si et seulement si* $\forall i, 0 \leq i < t, u[i] = 1 \Rightarrow v[i] = 1$. Pour une recherche globale, le problème de localisation des publications susceptibles de contenir les mots clés de la requête se ramène à trouver les publications qui ont chacune une clé de stockage qui matche le filtre de Bloom contenant ces mots clés. Du fait de la nature de son index, *FreeCore* procède en deux étapes : d'abord la localisation des fragments, puis le réassemblage des chunks des filtres de Bloom fragmentés.

Pour cela, *FreeCore* fragmente $query$ en m chunks $q_1, ..., q_m$ comprenant chacun c bits. La numérotation des chunks de $query$ suit la même logique que celle des chunks des filtres de Bloom des descriptions. Soit un chunk ch de $query$, $matched(ch)$ désigne l'ensemble des vecteurs de c bits qui matchent ch, c'est-à-dire que $v \in matched(ch)$ si et seulement si v est un vecteur de c bits et $\forall i, 0 < i < c, ch[i] = 1 \Rightarrow v[i] = 1$. Etant donné un coupe (ch, pos) où $0 < pos \leq m$, nous désignons par *étape de traitement* associée à (ch, pos) la recherche de tous les fragments associés à un élément de $matched(ch)$ et dont la position est égale à pos.

Pour trouver toutes les localisations des publications satisfaisant une recherche globale, *FreeCore* effectue les m étapes correspondant aux différents couples $(chunk, position)$ résultant de la fragmentation de $query$. Une fois que tous les fragments associés à ces couples sont collectés, FreeCore procède au réassemblage des chunks appartenant au même filtre de Bloom. Rappelons que par construction, tous les fragments d'une même publication ont en commun la même estampille de publication. Ainsi, si on dispose de m fragments ayant la même estampille, on reconstitue le filtre de Bloom (et donc la clé de stockage) de la publication concernée

par concaténations successives des chunks contenus dans les fragments, et ce, dans l'ordre de leurs positions.

2.2.2. Collecte de fragments

Pour collecter tous les fragments pertinents, *FreeCore* implante un protocole similaire à celui proposé par Joung et al [5]. L'algorithme 2 décrit succinctement le protocole de collecte de tous les fragments des publications susceptibles de satisfaire une recherche globale.

Algorithm 2 Collecte des listes inversées de fragments pertinents

Input: keywords
1: BloomFilter query ⟵ calculFiltreBloom(keywords)
2: List<Fragment> conteneur ⟵ ∅
3: List<NodeId> visited ⟵ ∅
4: Chunk[ ] chunks⟵ fragment(query)
5: int[ ] step⟵ ∅
6: NodeId coordinateur = getLocalNodeId() ;
7: PlanExeReq plan ⟵ new PlanExeReq(conteneur, visited, chunks, step) *//Création du plan d'exécution.*
8: Chunk c⟵ getMaxchunkBF(plan) *//on récupère le chunk avec le plus grand nombre de bits à 1.*
9: SearchFragmentMessage message⟵ new SearchFragmentMessage(coordinator, c, plan)
10: route(c, message)

Une fois que *FreeCore* dispose de l'ensemble de chunks du filtre de Bloom associé à la requête, il lui associe un plan d'exécution. Le plan d'exécution d'une requête encapsule les informations caractérisant cette requête ainsi que l'état d'avancement de son traitement. Le plan d'exécution contient notamment : la liste de toutes les étapes de traitement, l'identification de l'étape en cours de traitement, la liste des étapes terminées, le conteneur des fragments pertinents déjà collectés, et la liste pairs visités. Initialement le conteneur de fragments est vide, de même que la liste de pairs déjà visités (voir la ligne 7).

Après avoir initialisé le plan d'exécution, le coordinateur choisit le chunk ayant le plus de bits à 1 et initialise l'étape courante avec sa valeur, puis démarre le parcours en envoyant une demande de recherche de fragments au pair qui gère la liste (inversée) de fragments associée au chunk identifiant l'étape courante. La demande contient le plan d'exécution, le chunk recherché et l'identificateur du coordinateur.

Chaque serveur d'index qui reçoit une demande d'exécution de la recherche de fragments ajoute dans le conteneur la liste de fragments associée à un chunk du filtre de la requête. Une fois le traitement local de la demande terminé, le pair courant ajoute son identifiant dans la liste de pairs visités et détermine le pair suivant qui doit traiter la requête et lui transmet l'exécution de la requête.

Pour déterminer le pair suivant, on commence par chercher s'il existe un pair non encore visité, stockant des fragments qui matchent le chunk courant. S'il en existe on choisit le plus proche du pair courant et on lui transmet l'exécution de la requête. Lorsque tous les pairs stockant des fragments qui matchent le chunk en cours de traitement ont été visités, on enregistre ce chunk dans la liste des étapes terminées, puis on choisit parmi les chunks non encore traités, celui qui a le plus de bits à 1. On met à jour l'étape courante et ensuite on transmet l'exécution au pair qui gère le chunk choisi. Le processus continue jusqu'à ce que toutes les étapes soient terminées.

A la fin du processus, *FreeCore* supprime du conteneur de fragments tous les filtres de Bloom fragmentés qui sont incomplets. Tout au long du parcours, un fragment peut être supprimé

de ce conteneur dès lors qu'on sait qu'il manquera des fragments du même groupe à la fin du parcours.

2.2.3. Classement des réponses

Compte tenu des propriétés des filtres de Bloom, il y a une probabilité non nulle d'obtenir des faux positifs. Ici, un faux positif est une publication dont le filtre de Bloom de sa description matche le filtre de Bloom de la requête, mais cette description ne contient pas tous les mots clés de la requête.
L'objectif ici est d'attribuer un score à chaque publication localisée, permettant ensuite de classer les publications. *Freecore* définit le score d'une publication par rapport à une requête comme étant le ratio nombre de mots clés de la requête appartenant à cette publication, par nombre total de mots clés de la publication. On note que plus le sore est élevé, plus la publication est considérée comme pertinente. En particulier, si la participation matche exactement une requête, son score est de 1 ; si au contraire les mots recherchés sont contenus dans une publication ayant d'autres mots clés, le score de cette publication sera d'autant plus faible que le cardinal de la description sera élevé.
Le calcul du score d'une publication nécessite d'accéder à cette publication. L'algorithme 3 décrit les principales actions pour noter et classer des publications. Le calcul du score est fait sur le pair où la publication est stockée. Si le score atteint un certain seuil, la réponse contient le score et la référence du contenu, sinon le serveur retourne uniquement le score. L'intérêt de cette anticipation est d'éviter un accès supplémentaire lors de la fabrication de la réponse à retourner à l'utilisateur.
Une fois que les scores des différentes publications précédemment localisées sont calculés, cette procédure retourne le classement à l'appelant.

Algorithm 3 classement des publications localisées

Input: clefs keywords
1: **for** $i = 0 \rightarrow clefs.length$ **do**
2: score[i] = getIfScoreIsSatisfied(k, keywords, thresold)
3: **end for**
4: Key[] classement = rank(clefs,score) ;
5: Retourner le classement

3. Etat de l'art

Plusieurs travaux s'intéressent à la recherche de documents XML contenus dans un réseau pair-à-pair. XP2P [2] construit un index qui contient des fragments de chemin xpath, ainsi que des informations sur les fragments parents et fils. Koloniari et al [6] utilise des structures de données spécialisées dérivées des filtres de Bloom pour optimiser la solution XP2P [2]. Jamard et al [4] proposent des filtres de Bloom distribués et les utilisent pour indexer à la fois les informations sur la structure et les informations sur les contenus textuels. Ces filtres sont ensuite utilisés comme un index réparti sur plusieurs pairs dans le réseau. Ces travaux exploitent la structure des documents XML [19] et la syntaxe des requêtes XPATH [3].
De nombreux systèmes d'index ont été proposés afin de supporter des requêtes complexes sur les DHT. PHT[11] est une structure d'index qui utilise un arbre de préfixes construit sur une DHT. Les nœuds sont identifiés par un préfixe de $D - bits$. PHT stocke les documents dans les nœuds feuilles de l'arbre. La structure arborescente est répartie sur le réseau P2P structuré

contenant les données indexées. Tang et al proposent LIGHT[17] pour améliorer les performances de PHT. LIGHT est aussi basée sur les arbres préfixés et se compose d'une structure arborescente, d'une structure de données appelée "leaf bucket" et une fonction de nommage. Ces systèmes ne sont pas destinés à la recherche d'information à base de mots clés.

Au cours de la décennie écoulée, plusieurs systèmes de recherche par mots clés des contenus des réseaux pair-à-pairs structurés ont été proposés [1, 13, 7, 15, 18]. Reynolds et al.[12] adoptent le partitionnement vertical et proposent un index distribué et utilisent des filtres de Bloom pour réduire la bande passante consommée pour les échanges des listes inversées entre pairs : au lieu d'envoyer la liste complète des documents, le filtre de bloom du nœud ayant le plus grand nombre de documents est envoyé. L'autre nœud effectue alors une intersection du filtre reçu et des documents qu'il stocke puis retourne le résultat. *FreeCore* utilise aussi les filtres de Bloom, mais pour fabriquer des listes inversées compactes. Dans Proof [20], l'index global associe à chaque terme les descripteurs des documents qui le contiennent ; chaque descripteur contient entre autres l'identifiant et le résumé (sous forme de filtre de Bloom) du document. Notre solution diffère de Proof notamment par le fait que Proof maintient une liste inversée par mot clé.

Wang et al. [9] proposent des structures de données dérivées des filtres de Bloom avec compteurs pour représenter les contenus des pairs de façon efficace, en prenant en compte le caractère dynamique des réseaux pair-à-pair. Ces structures sont utilisées pour le routage des requêtes.

Joung et al [5] et Szekeres et al [16] utilisent un hyper cube comme structure d'index. Ils associent à tout objet un vecteur de r-bits obtenu à partir des mots clés qui décrivent cet objet ; une fonction de correspondance associe un identifiant de pair à tout vecteur de r-bits. Le couple (description, référence) de chaque objet est stocké sur le pair correspondant au r-bits associé aux mots clés de la description. Pour retrouver les objets caractérisés par des mots clés données, le système calcule le vecteur r-bits correspondant à ces mots clés. Pour une recherche exacte le système accède au pair correspondant au vecteur calculé, puis retire les références associées à l'ensemble de mots clés de la requête. S'il s'agit d'une recherche globale, le système parcours les nœuds du sous-hyper cube dont la racine est le nœud qui correspond au vecteur caractérisant la requête et collecte les références associées aux sur-ensembles de la requête. *FreeCore* utilise la même approche que [5] mais diffère sur la nature de l'index. Dans [5], les descriptions de documents sont distribuées sur 2^r pairs (e.g., les super- nœuds) alors que dans *FreeCore* nous distribuons les publications sur l'ensemble des pairs, mais maintenons un index des fragments de filtres de Bloom permettant de localiser efficacement les contenus associés aux descriptions.

4. Validation expérimentale

Nous avons réalisé un prototype de *FreeCore* au-dessus de Chord [14]. Nous utilisons le simulateur peersim [10]. Nous considérons la simulation événementielle qui permet à un noeud d'exécuter un comportement suite à l'arrivée d'un événement parmi un ensemble planifié. Pour la génération des clés de noeuds et de documents, nous avons utilisé la fonction de hachage SHA-1.

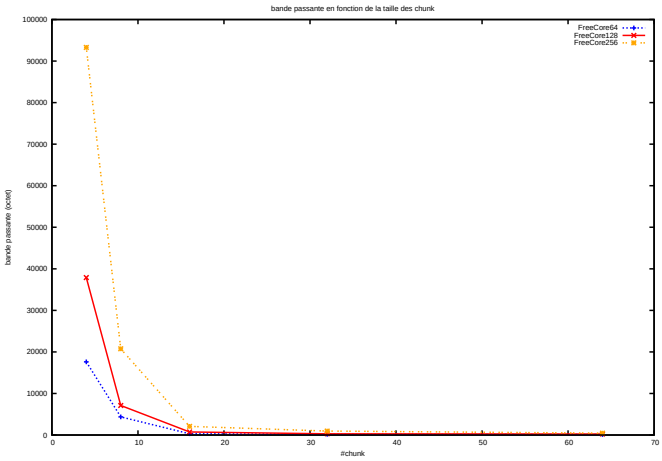
Les simulations ont été réalisées sur une machine possédant un processeur Intel Pentium de CPU B960 2.20GHz*2. Le système d'exploitation installé est Ubuntu 12.04 LTS version 64 bits.

Nous considérons trois métriques : la bande passante consommée pour transporter les listes inversées et la précision des résultats, la latence exprimée en terme de nombre de sauts (hops), et la précision. Nous n'avons pas considéré le taux de rappel qui est l'autre métrique de la qualité de la recherche car les propriétés des filtres de Bloom garantissent qu'il n'y a pas de

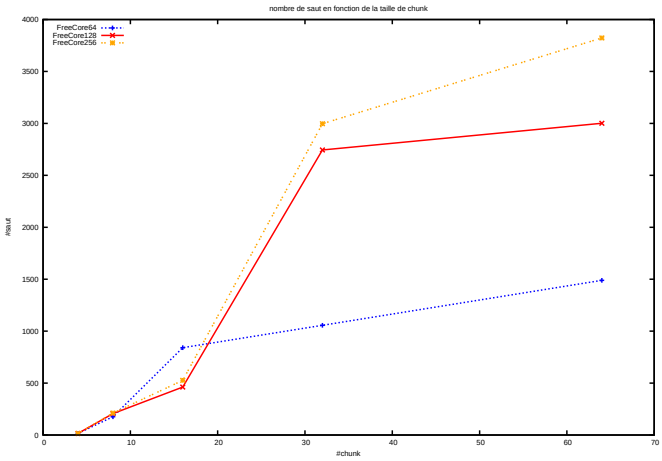
vrais négatifs.
Pour touts les mesures, nous considérons un dataset comprenant 240 descriptions différentes ayant entre 14 et 20 mots clés. Au total, le dataset comprend 800 mots clés uniques. Un test consiste en l'exécution de 30 requêtes fournissant chacune entre 5 et 10 mots clés.

4.1. Evaluation de différentes configurations de FreeCore

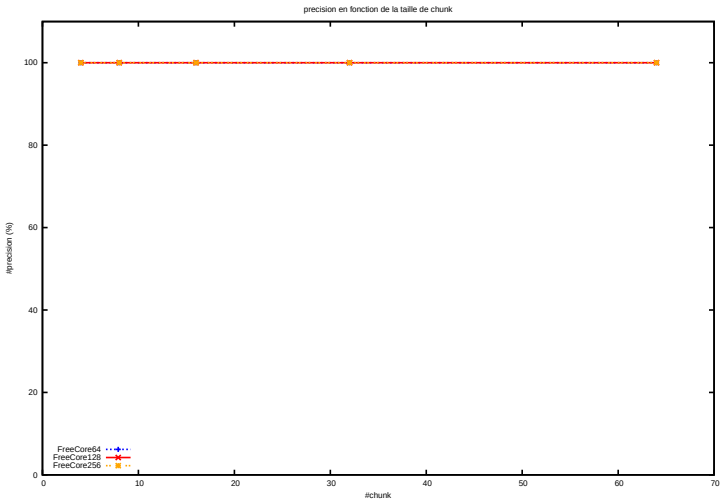
FreeCore a deux paramètres qui sont au coeur de son fonctionnement : la taille des filtres de Bloom utilisés pour représenter aussi bien les descriptions publiées que les requêtes, et la taille des fragments. Nous évaluons les performances de *FreeCore* pour différentes configurations.
Nous avons fixé le nombre de fonctions de hachage ; nous considérons trois tailles pour les filtres de Bloom : 64, 128, 256. Pour chaque taille de filtre, nous mesurons la métrique (bande passante, latence, précision, et rappel) pour différentes tailles de chunk : 4, 8, 16, 32 et 64.
Les figures 1a, 1b, et 1c. synthétisent les résultats de cette évaluation.



(a) Bande passante consommée



(b) Nombre de hops



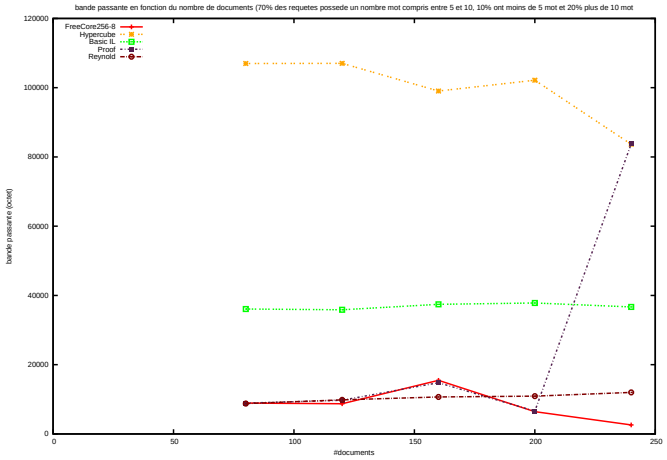
(c) Précision des résultats

FIGURE 1 – Bande passante consommée, latence et précision

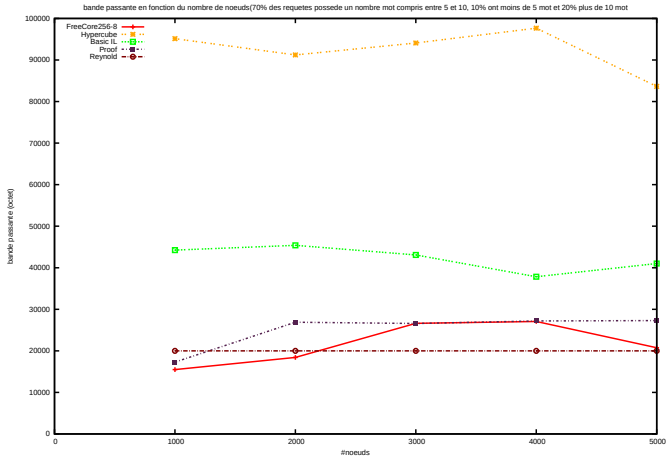
La figure 1a montre que, quel que soit la taille des filtres, plus la taille de fragments est petite, plus la consommation en bande passante est importante. La figure 1b montre que la performance en terme de nombre de sauts croit en fonction de la taille de fragment. Concernant la précision, le résultat biaisé par notre dataset. Nous avons construit des ensembles de mots clés sont soit 2 à 2 disjoints, soit l'un est inclus dans l'autre.

4.2. Comparaison avec d'autres systèmes d'index

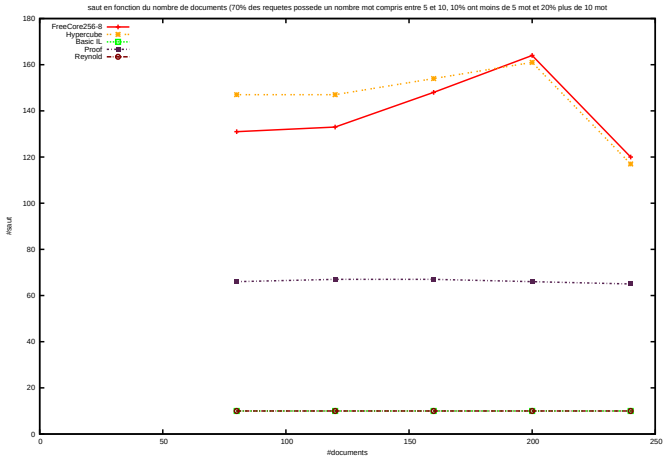
Nous comparons les performances de *FreeCore* à celles de quatre autres systèmes : un système maintenant un index distribué par mot clé (*IL de base*), Proof [20], le système présenté dans [12], et le système basé sur hyper cube. les performances de *IL de base* constituent le pire cas. Les 3 autres systèmes sont les plus comparables à *FreeCore* et représentent l'état de l'art. Pour *FreeCore*, nous utilisons des filtres de Bloom de 256 bits et des chunk de 8 bits. Nous considérons deux métriques : la bande passante et la latence.



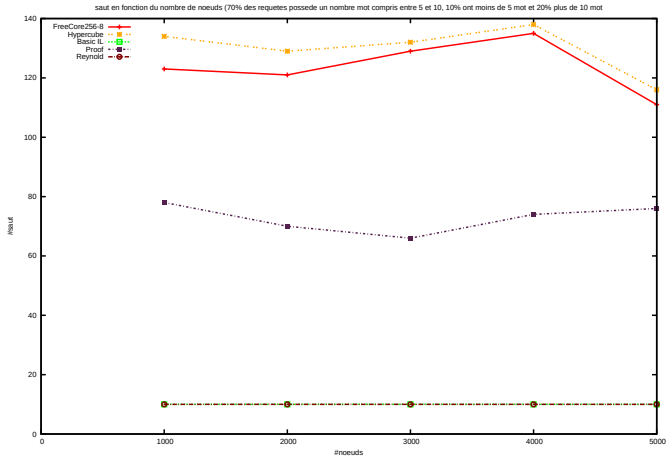
(a) bande passante en fonction du nombre de documents



(b) bande passante en fonction de la taille du réseau



(c) Nombre de sauts en fonction du nombre de documents



(d) Nombre de sauts en fonction de la taille du réseau

FIGURE 2 – Comparaisons des consommations de bande passante et de nombre de sauts

Les figures 2a et 2b représentent les consommations de bande passante des systèmes considérés. Bien que nous ayons choisi une des configurations les moins favorables pour *FreeCore*, les performances de *FreeCore* sont comparables à celle de Reynolds et de Proof, et sont meilleures que celle d'hyper cube.
Les figures 2c et 2d représentent le nombre de sauts réalisés lors de chaque test pour des différents systèmes considérés. Les performances de *FreeCore* ne sont pas bonnes. Lorsque les filtres sont fragmentés en des morceaux très petits comparé à la taille des filtres de Bloom, *FreeCore* est moins performant que les solutions optimisées à base de liste inversées par mot clé telles que Proof et Reynolds. Ce comportement a deux causes : (i) le nombre de morceaux du filtre de Bloom représentant chaque requête est plus grand que le nombre moyen de mots clés par requête ; (ii) les listes inversées associées aux différents morceaux du filtre de Bloom de la requête ne contient pas que des fragments réassemblés.

5. Conclusion

Nous avons présenté *FreeCore*, un système pair-à-pair structuré qui permet de publier des contenus en associant une description à chacun. Nous avons décrit la construction d'un index distribué sur les pairs qui associe à chaque vecteur de c bits la liste des fragments de filtres de Bloom fragmentés associés à ce vecteur. Nous avons montré que grâce à cette adaptation du partitionnement vertical qui associe une liste inversée à chaque vecteur de c bits plutôt qu'à chaque mot clé, les coûts de maintenance et de recherche deviennent indépendants du nombre de mots clés fournis. Les premières évaluations menées ont montré que les performances de *FreeCore* dépendent des valeurs des paramètres t_{fb} et c. Toutefois, *FreeCore* a des meilleures performances que les systèmes existants comparables.

Bibliographie

1. Bender (M.), Michel (S.), Triantafillou (P.), Weikum (G.) et Zimmer (C.). – Minerva : Collaborative p2p search. – In *Proceedings of the 31st International Conference on Very Large Data Bases*, *VLDB '05*, VLDB '05, pp. 1263–1266. VLDB Endowment, 2005.
2. Bonifati (A.), Matrangolo (U.), Cuzzocrea (A.) et Jain (M.). – Xpath lookup queries in p2p networks. – In *Proceedings of the 6th annual ACM international workshop on Web information and data management*, *WIDM '04*, WIDM '04, pp. 48–55, New York, NY, USA, 2004. ACM.
3. Groups (W. X. Q. W.). – *The XPath 2.0 Standard*, 2007.
4. Jamard (C.), Gardarin (G.) et Yeh (L.). – Indexing textual xml in p2p networks using distributed bloom filters. – In *Proceedings of the 12th international conference on Database systems for advanced applications*, *DASFAA'07*, DASFAA'07, pp. 1007–1012, Berlin, Heidelberg, 2007. Springer-Verlag.
5. Joung (Y.-J.), Fang (C.-T.) et Yang (L.-W.). – Keyword search in dht-based peer-to-peer networks. – In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, *ICDCS '05*, ICDCS '05, pp. 339–348, Washington, DC, USA, 2005. IEEE Computer Society.
6. Koloniari (G.) et Pitoura. (E.). – Content-based routing of path queries in peer-to-peer systems. – In *Proceedings of the 9th International Conference Extending Database Technology (EDBT'04)*.
7. Lele (N.), Wu (L.-S.), Akavipat (R.) et Menczer (F.). – Sixearch.org 2.0 peer application for collaborative web search. – In *Proceedings of the 20th ACM Conference on Hypertext and Hypermedia*, *HT '09*, HT '09, pp. 333–334, New York, NY, USA, 2009. ACM.

8. Li (J.), Loo (B. T.), Hellerstein (J. M.), Kaashoek (M. F.) et et al. – On the feasibility of peer-to-peer web indexing and search. – In *IN IPTPS ?03*, pp. 207–215, 2003.
9. Meng-Fan (W.), Da-Fang (Z.), Xiao-Mei (T.), xia an (B.) et Bin (Z.). – Multi-keyword search over p2p based on counting bloom filter. – In *2011 2nd International Conference on Networking and Information Technology IPCSIT*.
10. Montresor (A.) et Jelasity (M.). – PeerSim : A scalable P2P simulator. – In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pp. 99–100, Seattle, WA, septembre 2009.
11. Ramabhadran (S.), Ratnasamy (S.), Hellerstein (J. M.) et Shenker (S.). – Brief announcement : Prefix hash tree. – In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, St. John's, Newfoundland, Canada, july 2004.
12. Reynolds (P.) et Vahdat (A.). – Efficient peer-to-peer keyword searching. – In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, *Middleware '03*, Middleware '03, pp. 21–40, New York, NY, USA, 2003. Springer-Verlag New York, Inc.
13. Rosenfeld (A.), Goldman (C. V.), Kaminka (G. A.) et Kraus (S.). – Phirst : A distributed architecture for p2p information retrieval. *Inf. Syst.*, vol. 34, n2, avril 2009, pp. 290–303.
14. Stoica (I.), Morris (R.), Karger (D.), Kaashoek (M. F.) et Balakrishnan (H.). – Chord : A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, vol. 31, n4, août 2001, pp. 149–160.
15. Suel (T.), Mathur (C.), wen Wu (J.), Zhang (J.), Delis (A.), Kharrazi (M.), Long (X.) et Shanmugasundaram (K.). – Odissea : A peer-to-peer architecture for scalable web search and information retrieval. – In Christophides (V.) et Freire (J.) (édité par), *WebDB*, pp. 67–72, 2003.
16. Szekeres (A.), Baranga (S. H.), Dobre (C.) et Cristea (V.). – A keyword search algorithm for structured peer-to-peer networks. *Int. J. Grid Util. Comput.*, vol. 2, August 2011, pp. 204–214.
17. Tang (Y.), Zhou (S.) et Xu (J.). – Light : A query-efficient yet low-maintenance indexing scheme over dhts. *IEEE Trans. on Knowl. and Data Eng.*, vol. 22, January 2010, pp. 59–75.
18. Tigelaar (A. S.), Hiemstra (D.) et Trieschnigg (D.). – Peer-to-peer information retrieval : An overview. *ACM Trans. Inf. Syst.*, vol. 30, n2, mai 2012, pp. 9 :1–9 :34.
19. toExcel. – *Extensible Markup Language (Xml) 1.0 Specifications : From the W3C Recommendations*. – iUniverse, Incorporated, 2000.
20. Yang (K.-H.) et Ho (J.-M.). – Proof : A dht-based peer-to-peer search engine. – In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence*, *WI '06*, WI '06, pp. 702–708, Washington, DC, USA, 2006. IEEE Computer Society.